INPE-16678-PUD/218

NATURAL LANGUAGE REQUIREMENTS: AUTOMATING MODEL-BASED TESTING AND ANALYSIS OF DEFECTS

Valdivino Alexandre de Santiago Júnior

INPE
São José dos Campos
2010

PUBLISHED BY:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3945-6911/6923
Fax: (012) 3945-6919
E-mail: pubtc@sid.inpe.br

INPE-16678-PUD/218

NATURAL LANGUAGE REQUIREMENTS: AUTOMATING MODEL-BASED TESTING AND ANALYSIS OF DEFECTS

Valdivino Alexandre de Santiago Júnior

INPE
São José dos Campos
2010

ABSTRACT

At present, there are several techniques to elaborate software requirements specifications. However, the simplest way for stakeholders to elaborate software requirements is still Natural Language. Moreover, Natural Language may be associated to requirements modeling methods, like use case models where a textual description exists in order to describe behavior through a sequence of actor-system interactions. But, Natural Language specifications suffer from problems like inconsistency, incompleteness, and ambiguity. Some authors argue that trying to develop complete and consistent requirements models is not interesting, but the goal shall be to analyze and resolve conflicting requirements, and to reason with models that contain inconsistencies. However, in order to follow these guidelines, the aforementioned issues, inconsistency and incompleteness, shall be first properly detected. Requirements also serve as a starting point to develop models for system and accepance test case generation. However, both activities, system and acceptance test case generation, and analysis of Natural Language requirements to detect defects, are usually very time-consuming specially if one considers complex systems. This PhD proposal presents a methodology, as well as the main characteristics of the tool that will support it, named *Automatizando TesStes BasEados em Modelos e Análise de DeFeitos considerAndo Requisitos em Linguagem NAtural* (SEMAFALA - Automating Model-Based Testing and Analysis of Defects considering Natural Language Requirements). The goals of the methodology are exactly the automated translation of Natural Language requirements into behavioral models to support system and acceptance Model-Based Testing, and to also detect automatically inconsistency and incompleteness in such requirements.

REQUISITOS ELABORADOS EM LINGUAGEM NATURAL: AUTOMATIZANDO TESTES BASEADOS EM MODELOS E ANÁLISE DE DEFEITOS

RESUMO

Atualmente, existem diversas técnicas para elaborar especificações de requisitos de software. Entretanto, o modo mais simples para "stakeholders" elaborarem requisitos de software é ainda a Linguagem Natural. Além disso, Linguagem Natural pode estar associada a métodos de modelagem de requisitos, tais como modelos de caso de uso onde uma descrição textual existe para descrever comportamento por meio de uma seqüência de interações ator-sistema. Mas, especificações elaboradas em Linguagem Natural possuem problemas como inconsistência, não completude e ambiguidade. Alguns autores argumentam que tentar desenvolver modelos de requisitos completos e consistentes não é interessante, mas o objetivo deve ser analisar e resolver conflitos em requisitos, e raciocinar com modelos que possuem inconsistência. Porém, para seguir essas diretivas, os problemas mencionados anteriormente, inconsistência e não completude, devem ser primeiramnte detectados adequadamente. Requisitos também servem de ponto de partida para desenvolver modelos para a geração de casos de teste de sistema e aceitação. Entretanto, ambas as atividades, geração de casos de teste de sistema e aceitação e análise de requisitos elaborados em Linguagem Natural para detectar defeitos, usualmente consomem bastante tempo, especialmente se forem considerados sistemas complexos. Esta proposta de doutorado apresenta uma metodologia, assim como as principais características da ferramenta que dará suporte a tal metodologia, denominada *Automatizando TesStes BasEados em Modelos e Análise de DeFeitos considerAndo Requisitos em Linguagem NAtural* (SEMAFALA). Os objetivos da metodologia são exatamente a tradução automática de requisitos elaborados em Linguagem Natural em modelos comportamentais visando Testes Baseados em Modelos de sistema e aceitação, e também detectar automaticamente inconsistência e não completude em tais tipos de requisitos.

CONTENTS

Pág.

LIST OF FIGURES

Pág.

LIST OF TABLES

1 INTRODUCTION

One desirable property related to every single product is quality. Despite the domain, organizations want to assure they build high-quality products. In software context it is not different. Nowadays, several software development organizations put lot of emphasis in the quality mechanisms for achieving products with high levels of user satisfaction.

Quality is very difficult to define. Different specialists provide distinct points of views regarding this concept. However, it is possible to identify elements of quality definitions, such as defect level, defect origins, product complexity, conformance to requirements, user satisfaction and robustness (GODBOLE, 2006). Regardless of the perspective, practitioners agree that quality is a major business factor and achieving it requires huge effort from the organizations.

Software Quality Assurance involves several activities like planning, measurement, configuration management, walkthroughs, inspection and testing. The level of such activities depends on the project and on the organization. Thus testing a software product is only a facet to get quality. However, the role of testing is undoubtedly important and industry and academia have been paying attention to this task for several years.

One issue is that testing activities accounts for 30% to 40% of total software development efforts (PRESSMAN, 2001), or even higher depending on the criticality of the application. Thus, test automation appeared as an attempt to reduce the costs of testing, increase fault detection and shorten testing cycles. Although test automation is not a silver bullet to solve all testing problems, if properly planned and implemented, it can help to achieve cost-effectiveness of test process activities during the software development lifecycle (SANTIAGO et al., 2008a).

In spite of the fact that several commercial, proprietary, or open source tools are available supporting the automation of testing, the human factor is still very present within a test process. For instance, a tool may help to automatically generate test cases, but a test designer is necessary to evaluate if all test cases are feasible to apply given constraints like lack of time and test cases redundancies. Or an enviroment is able to create test cases automatically, but one shall derive a behavioral model to be the basis for such enviroment to do its job. A framework may support execution and assertion of verdicts of test cases automatically, but a testing professional shall develop lots of test scripts. The point is that there are several ways to improve even more the testing automation and try to provide confidence that the software product has intrinsic quality.

Dynamic Analysis is the process of evaluating a system or component based on its behavior during execution (IEEE, 1990). The bottom line is that testing can be considered a type of dynamic analysis technique. There are many other dynamic analysis techniques such as: Ernst et al. (2001) which used dynamic techniques for discovering invariants from execution traces to support software evolution; Ammons et al. (2002) which proposed a technique to produce specifications based on program execution and also enable such artifacts to be used by formal verification tools to find faults; and Lorenzoli et al. (2008) which addressed the generation of models of relations between data values and component interactions based on *GK-tail*, a technique to automatically generate Extended Finite State Machines (EFSMs) from interaction traces.

On the other hand, Static Analysis is the process of evaluating a system or component based on its form, structure, content, or documentation (IEEE, 1990). Hence, program execution is not required in static analysis. Inspection is an example of a classic static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems (IEEE, 1990). Examples of deliverables examined within an inspection are software requirements specifications, design documents and source code. Inspection is a review method with a related process. An inspection meeting is held with various participants playing distinct roles like moderator, author of the inspected deliverables and reviewer. The general idea is to add value to the artifacts developed by the author giving hints to improve them.

Software Systems Requirements Engineering (RE) also contributes to software quality. A simple definition states that RE is the process of discovering the purpose of a software system. Nuseibeh and Easterbrook define 5 core RE activities: elicitation, modeling and analysis, communication, agreement, and evolution (NUSEIBEH; EASTERBROOK, 2000). Elicitation is the process of gathering requirements information, and it is closely related to other RE activities. Modeling refers to the development of abstract descriptions that are amenable to interpretation. Several analysis techniques exist once requirements are modeled. Requirements shall be effectively communicated among the diferent stakeholders, and the way in which requirements are documented plays an important role regarding this. As the name implies, agreement refers to the fact that stakeholders get consensus about the requirements. Inspection is one technique that may be applied within this activity. Software systems are dynamic, i.e. they evolve. Stakeholders requirements also change. Thus, evolution is a natural characteristic of requirements. Some studies revealed that more than 50% of software defects are attributed to requirements problems and more than 80% of rework effort is spent on requirements-related defects (FERGUSON; LAMI, 2005). This fact proves how important RE is in the context of software development.

Software requirements specifications may be elaborated according to several approaches. Requirements may be elicited and modeled based on scenario-based methods, such as use case models (OMG, 2007), and goal-oriented methods, such as Tropos (BRESCIANI et al., 2004). Class and object hierarchies may be the choice to model data. Formal methods (models and languages) may be used to model behavior, but they require high expertise for that. Thus, formal methods are not very common in industrial practice. The conclusion is that Natural Language (NL) is still the most used to elaborate requirements specifications (MICH et al., 2004) provided it is the simplest way for stakeholders. Moreover, NL may be associated to requirements modeling methods, like use case models where a textual description exists in order to narrate the behavior through a sequence of actor-system interactions (SINHA et al., 2007).

Unfortunately, serious shortcomings exist if a specification is elaborated in NL. Most notably, ambiguity, incompleteness and inconsistency make a document unclear and this may impact on the next artifacts produced within the software development lifecycle. However, Nuseibeih and Easterbrook argue that “The idea that the attempt to build consistent and complete requirements models is futile, and that RE has to take seriously the need to analyse and resolve conflicting requirements, to support stakeholder negotiation, and to reason with models that contain inconsistencies” (NUSEIBEH; EASTERBROOK, 2000). However, in order to follow these guidelines, the aforementioned issues, inconsistency and incompleteness, shall be first properly detected. For instance, if an inconsistency is detected in a software requirements specification, such inconsistency may be tolerated, and resolved at a later stage. This is known as the *tolerating inconsistency* approach (BALZER, 1991). Requirements can continue evolving, and inconsistency is nonblocking in this proposal (GERVASI; ZOWGHI, 2005). But, in order to adopt the *tolerating inconsistency* approach, the inconsistency itself shall be first detected. To accomplish this in very complex NL software requirements specifications automatically is very challenging but, at the same time, relevant in order to decrease the time required to perform such analysis.

This document presents the proposal regarding this PhD Thesis. The goals of this proposal are two-fold. First, the automated translation of NL requirements into behavioral models addressing system and acceptance test case generation. This is very challenging given that system and acceptance test cases are generated considering the entire software product. In a Model-Based Testing (MBT) approach, a test designer should identify scenarios and later develop models to cover them, and this demands huge effort from the testing professional because the number of scenarios may be very large. Besides, the idea is to develop a methodology with tool support so that the requirements expressed in NL can be converted automatically to the behavioral models. The second goal is to addresss automatically incompleteness and inconsistency in NL software requirements specifications with the help of the tool that will be developed. Thus, such tool will support the RE modeling and analysis activity (NUSEIBEH; EASTERBROOK, 2000).

This work is organized as follows. Chapter 2 will provide an overview of some approaches dealing with the previously cited issues in NL requirements. Chapter 3 will discuss briefly software testing with emphasis in MBT. Chapter 4 will present the proposal itself. Conclusions, future work, and schedule of this proposal are in Chapter 5.

2 NATURAL LANGUAGE REQUIREMENTS

Natural Language Processing (NLP) is a field of computer science and linguistics dealing with the problem of computers to process and understand human languages. NLP has significant overlap with the field of computational linguistics, and some authors consider it a sub-field of Artificial Intelligence (AI) (RUSSELL; NORVIG, 1995). There are several domains in which NLP may be applied such as spelling correction, grammar checking, search engines, information extraction, information retrieval, speech recognition, and so on.

Jurafsky and Martin distinguish into six categories the knowledge of language needed to engage in complex language behavior (JURAFSKY; MARTIN, 2000): Phonetics and Phonology, Morphology, Syntax, Semantics, Pragmatics, and Discourse. The field of NLP is vastly supported by several concepts like finite automata, N-grams, Hidden Markov Models, Part-Of-Speech (POS) tagging, lexicalized and probabilistic parsing, Chomsky hierarchy, pumping lemma, first order predicate calculus, and many others.

The following section will present some works addressing the analysis of defects in NL requirements. These approaches make use of some categories and concepts mentioned above.

2.1 Analysis of Defects in NL Requirements

Published literature has been addressing the analysis of software requirements to detect defects such as ambiguity, incompleteness, and inconsistency. Formal specification languages have drawn attention to specify software requirements because they add formality and remove ambiguities, usually found in NL requirements specifications. However, they are difficult to understand by ordinary professionals in industry and this limits their applicability to some restricted domains. Requirements modeling methods with an associated graphical notation, such as Unified Modeling Language (UML) (OMG, 2007) use cases and goal-oriented approaches, have become popular in academia and also in some industry domains. However, it is usual that requirements expressed in NL are the basis for deriving high level design documents containing UML class, sequence and state machines models. The point is by no means to say that UML-based software development is worthless but to stress that NL is the most natural way to describe requirements and, hence, it is still the most common approach to elaborate software requirements in practice (GNESI et al., 2005) (MICH et al., 2004).

The aforementioned defects (ambiguity, incompleteness, and inconsistency) usually present in NL requirements specifications demand the development of methodologies and supporting tools to try to improve the specification quality. This section will present some significant approaches with respect to this issue. The *Quality Analyzer for Requirements Specification* (QuARS) is a tool that enables the user to analyze NL requirements automatically (LAMI; TRENTANNI, 2004) (GNESI et al., 2005) (FERGUSON; LAMI, 2005) (BUCCHIARONE et al., 2008). Three categories of quality properties should be accounted for when analyzing NL specifications: expressiveness (mainly ambiguity and poor readability), consistency, and completeness. A quality model for the expressiveness property was defined in a previous work (FABBRINI et al., 2001) and QuARS was developed based on such model to automate NL requirements analysis. QuARS makes use of dictionaries to support the requirements analysis.

QuARS is able to partially support inconsistency and incompleteness analysis by clustering the requirements, also known as View Derivation, according to specific topics like security. However, such analysis is not accomplished automatically. Moreover, the analysis QuARS performs is limited to syntax-related issues of a NL requirements document addressing ambiguity. Matching words in NL requirements with those stored in repositories cannot remove ambiguity completely: a thorough analysis is necessary for that purpose (BERRY et al., 2003).

CIRCE is an environment that supports modeling and analysis of requirements elaborated in NL (AMBRIOLA; GERVASI, 2006) (AMBRIOLA; GERVASI, 1997). The modeling and analysis activity is accomplished by an expert system based on modular agents. The tool parses and transforms NL requirements into a forest of parse trees. To do that, CIRCE uses a domain-based parser called CICO. A requirements specification D is considered a triple $\langle G, F, R \rangle$, where G is a set of *designations* (also called a glossary), F is a set of *definitions*, and R is a set of *requirements*. Another element of CIRCE are the *modelers*. In particular, the *validation modelers* aim at identifying inconsistent, ambiguous, and incomplete requirements. By defining a few requirements in accordance with the formal model embedded in CIRCE, i.e. by means of designations and definitions, the tool can generate models like state transition diagrams allowing the user to analyze problems in requirements.

CIRCE seems to be a remarkable tool. However, the issue is how easy for a user to express the domain in order to use CIRCE. In other words, application domain must be expressed by a user by means of designations and definitions which, in turn, must be written using a formal syntax. A Requirements Engineer must declare designations using lots of tags and he/she must perform a deep analysis of the NL requirements to accomplish that. The authors reported that their tool has been applied successfully in three informal pilot studies with industrial partners, and a more controlled study in cooperation with the National Aeronautics and Space Administration (NASA) Independent Verification and Validation (IV&V) facility. Although many benefits are pointed out, the cost of introducing CIRCE in the work environment shall not be neglected. The authors asserted that, in the NASA case study: "... pre-existing requirements can be analyzed by CIRCE as they are, thus imposing no additional burden on the requirements writer." However, it was not that simple because it was necessary to write *Model, Action, Substitution* (MAS) rules which are rules that drive the CICO's parsing algorithm. One of the authors, an expert in the conception of the tool, took 2 days to write such formal rules. Even though it is a one-time operation, depending on the domain it will be necessary to write these rules formally besides designations and definitions. One may wonder how easy is for a CIRCE non-specialized professional to write MAS rules.

Both forms to elaborate NL requirements, unrestricted and restricted (controlled) approaches, have supporters. In a recent survey among 142 software development organizations, Mich et al. (2004) asserted: "... we find that in a majority of cases it is necessary to use NL Processing systems capable of analysing documents in full natural language". In a previous work, Mich (MICH, 1996) presented the *Natural Language - Object Oriented Production System* (NL-OOPS). The NL-OOPS tool supports analysis of unrestricted NL requirements by extracting the classes and their associations for use in creating class models. The unrestricted NL analysis is obtained using as a core the NL processing system *Large-scale, Object-based, Linguistic Interactor, Translator, and Analyser* (LOLITA) (MORGAN et al., 1995). LOLITA is built around a large graph called SemNet, a particular form of conceptual graph, which holds knowledge that can be accessed, modified or expanded using NL input. NL-OOPS allows detection of ambiguities in the text but there is no evidence that it supports automated detection of incompleteness and inconsistency.

However, Ambriola and Gervasi (AMBRIOLA; GERVASI, 2006) mentioned that the lack of domain knowledge limits the applicability of systems based on unrestricted NL requirements. They asserted that "... assuming that the user should provide no further information than the requirements themselves, these systems have to resort to heuristics to identify the proper objects, or rely on domain-specific knowledge bases". In the first case, they mentioned unsatisfactory results when heuristic algorithms may need to be tuned, as reported in Mich et al. (2002). Moreover, there are also reports of bad performance regarding the evaluation of information extraction by using the LOLITA system (MORGAN et al., 1995) (CALLAGHAN, 1998). In the second case, a negative aspect is the effort to build sufficiently large domain knowledge bases which may be impractical.

Gervasi and Zowghi proposed a formal framework for identifying, analyzing, and managing inconsistency in requirements derived from multiple stakeholders and expressed in NL (GERVASI; ZOWGHI, 2005). A prototype tool, CARL, was developed incorporating all the techniques described in the paper. Their formal model relies on two different AI techniques, *default reasoning* and *belief revision*. They focus on a particular kind of inconsistency, *logical contradiction*, and the authors claim that the framework supports the detection of both explicit and hidden inconsistencies (the cases in which the inconsistency occurs due to the consequences of some requirements, rather than the requirements themselves). They adopted the *tolerating inconsistency* approach (BALZER, 1991). For dealing formally with inconsistency, first requirements expressed in controlled NL are automatically parsed and translated into propositional logic formulae. This process involves morphosyntactic analysis and the previously mentioned domain-based parser CICO. Once the specification is represented as sets of propositional logic formulae, a theorem prover and a "model checker"[1] are used aiming at detecting inconsistencies, the latter addressing the discovery of the hidden ones.

Despite these interesting features, and as well as CIRCE (AMBRIOLA; GERVASI, 2006), CARL suffers from the same problem regarding the likely need to write new MAS rules depending on the domain. Moreover, scalability is still an issue because the example shown in the paper is too simple as the specification has very few requirements.

[1]Actually, CARL does not perform "true" model checking according to its most used definition (CLARKE; LERDA, 2007).

Hunter and Nuseibeh proposed a formal approach to reason, analyze, and accomplish actions in inconsistent specifications (HUNTER; NUSEIBEH, 1998). It seems that such work influenced in some ways the CARL development. Hunter and Nuseibeh adopted the tolerating inconsistency approach, they dealt with one kind of inconsistency, *logical contradiction*, and multiple stakeholder development was accounted for. They presented an adaptaion of classical logic, Quasi-Classical (QC) logic, that allows continued reasoning in the presence of inconsistency. According to them, their approach is also able to identify the likely sources of inconsistencies, and use this to suggest actions. Despite the remarkable work, no tool was developed to automate the processes of reasoning, analysis, and action, and the case study presented was too much simple (by the way it is the same case study used in Gervasi and Zowghi (GERVASI; ZOWGHI, 2005)).

Kim and Sheldon presented a method that models and evaluates NL software requirements specifications using the Z formal language and Statecharts (KIM; SHELDON, 2004). Their method transforms a NL specification into a Z specification which in turn derives the Statecharts models (actually, State/Activity charts). The case study used in their work was the NASA Guidance and Control Software (GCS) developed for the Viking Mars Lander. The goal was to analyze the integrity of the GCS specification in terms of completeness, consistency, and fault-tolerance. Their work presented some interesting results but the transformations proposed were heavily dependent on human skill, and there is no evidence that a tool was developed to automate the defects detection.

Java Requirement Analyzer (J-RAn) is a tool that implements a Content Analysis technique to support the analysis of inconsistency and incompleteness in NL requirements specifications (FANTECHI; SPINICCI, 2005). Based on the NL document, this technique exploits the extraction of the interactions between the entities described in the specification as Subject-Action-Object (SAO) triads. These SAO triads are obtained with the help of the *Link Grammar Parser* (SLEATOR; TEMPERLEY, 1993), a syntactic parser of English based on *link grammar*, a formal grammatical system. J-RAn was used in a very simplified case study and, even so, a significant number of SAOs were incorrectly extracted (21%) and missing extractions were also huge (16%). The tool helps in the analysis of inconsistency and incompleteness by providing Content Analysis charts (graphs) to a requirements analyst. However, the analysis itself is not automated but manually carried out by the analyst.

Text Analyzer is a tool designed and implemented to generate test cases based on NL requirements (SNEED, 2007). Thus, it supports black box testing and it is intended to be used for system and acceptance testing. *Text Analyzer* needs heavy user intervention in order to define the application domain. The tool first scans the text in order to identify all nouns. These nouns are displayed to the test designer who decides which ones are considered pertinent objects of the Implementation Under Test (IUT), i.e. the software product that is the target of testing. Such objects are in turn the elements which the test cases relate to. This task can be very time-consuming depending on the complexity of the requirements specification. The user must also identify *keywords* used in the requirements text (e.g. *INPT = this word indicates a system input*). This is another activity that demands time and probably makes the approach less attractive specially if one considers complex NL requirements documents. One last remark is that the author did not mention how to handle the previously explained problems regarding NL requirements (ambiguity, inconsistency, incompleteness).

Table 2.1 summarizes the characteristics of the tools/approaches discussed in this subsection. It is a hard task to realize all the properties of a tool without using it. As all these tools are not available for free use, indeed some of them will probably be delivered as commercial products, it was not possible to use them and perceive the real benefits and shortcomngs in practice. In Table 2.1, a question mark (?) means that it is not totally clear that the feature is provided by the tool. It happens once within the CIRCE tool because it is not evident that the state transition diagrams it generates are true Statecharts or FSMs. *Partial* refers to the cases where the characteristic is partially covered because it is not clear to which extent the feature is completely addressed by the tool/approach. *Manual* implies either that a user must observe some sort of information on the interface of the tool in order to wonder about the characteristic or that there is no tool developed at all. An X implies the tool/approach has such feature and an empty box means the opposite. Besides, the **K-S** column refers to the Kim-Sheldon approach (KIM; SHELDON, 2004), and **Text An** is the *Text Analyzer* tool (SNEED, 2007).

Table 2.1 shows that none of the tools/approaches address the problem of translating automatically NL requirements into behavioral models (e.g Statecharts and/or FSMs) for system and acceptance MBT. *Text Analyzer* can be used for

Table 2.1 - Characterisitics of tools/approaches for analysis of NL requirements.

Characteristics	**CIRCE**	**QuARS**	**NL-OOPS**	**CARL**	**K-S**	**J-RAn**	**Text An**
Ambiguity (automatically)	Partial	Partial	Partial				
Inconsistency (automatically)	Partial	Manual		Partial	Manual	Manual	
Incompleteness (automatically)	Partial	Manual			Manual	Manual	
Solution Suggestion	X			X			
Clustering (to improve doc structure)	X	X					
Transformation to Statecharts or FSMs	?				Manual		
Transformation to other structural and behavioral models (including UML use case, class, sequence, collaboration models)	X		X (class model)				
Restricted NL	X	X		X	X		X
Unrestricted NL			X			X	
Scalability		X					X
Supported by Formal Method	X			X	X		
Supported by AI	X		X	X			
NL usage for Testing							X
NL usage for Model-Based Testing							

system and acceptance test case generation but it does not make use of formal techniques (models) and their benefits. Automated derivation of UML models from NL specifications is a desired and helpful feature which may support not only software development but also the testing process. CIRCE and, although only class models, NL-OOPS(LOLITA) allow such translation but deriving behavioral models automatically (like Statecharts or FSMs) taken into account system and acceptance testing is very challenging, because a test designer should usually derive several models according to functionalities, gathered from complex NL requirements specification, that an IUT must provide. In other words, it is used to adopt a scenario-based approach. This kind of automation, addressing testing of the entire product, is considerably difficult to be achieved. These facts explain why none of the above tools/aproaches supports Model-based testing having as starting point NL requirements.

The divide and conquer philosophy comes to help due to the fact that it is extremely difficult to obtain Model-based test cases automatically if one considers the modeling of the entire software. Test case explosion is likely to occur when trying to obtain test cases from the model of the entire system. Hence, after the scenarios had been defined and the models to cover them developed, a Model-based tool can be used for test case generation. Moreover, despite the remarkable work of the tools/approaches presented, it seems that novel methodologies/tools are necessary in order to truly automate the detection of defects within NL requirements specifications.

One last work related to NL will be mentioned. Konrad and Cheng presented a process that supports the specification and analysis of UML models with respect to behavioral properties specified in NL (KONRAD; CHENG, 2006). This process has been implemented in the *SPIDER* tool suite. This approach is a model checking of UML models against NL properties. Specifically, the process is configured to read UML 1.4 models and generate the formal specification language PROMELA for the model checker SPIN (HOLZMANN, 2003). NL properties are derived using a previously developed grammar (KONRAD; CHENG, 2005) that supports the specification patterns proposed by Dwyer et al. (1999). The grammar enables the NL representation of these specification patterns, and it is used to specify Linear-Time Temporal logic (LTL) properties, the property description language of the SPIN model checker. Hence, the goal is to analyze UML models against the NL properties and not to detect the issues of the NL properties themselves.

3 SOFTWARE TESTING

Many definitions of software quality have been proposed in the literature. Pressman defined this branch of Software Engineering as (PRESSMAN, 2001): “Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software”. This definition emphasizes, among other points, that software requirements are the basis from which quality is measured. Lack of conformance to requirements is lack of quality. However, one problem arises when software requirements specifications are poorly elaborated and, therefore, good requirements specifications are a valuable starting point towards high-quality software.

Verification and Validation are two important concepts related to software quality. In this work, these terms will be defined in accordance with the IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology* (IEEE, 1990):

a) Verification: the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the beginning of that phase;

b) Validation: the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

In summary, what Verification highlights is whether the outputs of each phase of the software development lifecycle are according to what has been specified during planning. But, Verification does not guarantee that such outputs are necessarily correct. In turn, Validation tries to confirm that specifications of one phase or even the entire system is adequate and consistent with customer requirements. Verification and Validation encompass a wide array of Sofware Quality Assurance activities among others formal technical reviews, quality and configuration audits, documentation review, feasibility study and all sort of testing (PRESSMAN, 2001).

Nevertheless, there is another definiton for Verification as stated in the IEEE Std 610.12-1990 (IEEE, 1990): Verification is formal proof of program correcteness. Formal verification constructs mathematical proofs about the behavior of computer hardware and software. As a heavy user of mathematical logic, it has strong connections with theoretical computer science. Thus, the aim of formal verification methods is to prove that a given system satisfies its specification by formal means, e.g. mathematical proofs (CLARKE; LERDA, 2007). Basically, any formal method consists of three elements:

a) Specification: expresses what the system ought to do;

b) Modeling: indicates what the system actually does;

c) Verification: process that checks whether the model satisfies the specification.

At present, one popular formal method is **temporal logic model checking** (CLARKE; EMERSON, 2008) (QUEILLE; SIFAKIS, 1982) (CLARKE; LERDA, 2007). Model checking was originally conceived for verifying finite state systems such as sequential circuit designs and communication protocols. In model checking, the specification is expressed using temporal logic, an extension of propositional logic that allows reasoning about the relative timing of events. On the other hand, the semantics of a system is usually given by means of a **Kripke structure**, a type of computational model.

One huge limitation of model checking is related to state explosion. If the system to be verified is too large or the specification is too complex, model checking might not terminate due to insufficient resources, e.g. running time and/or available memory. The state explosion problem can be tackled by symbolic model checking, a technique that uses Binary Decision Diagrams (BDDs) to represent sets of states and transitions (BRYANT, 1986). Model checking is then performed directly on the BDD representations. However, BDD-based symbolic model checking still presents some problems when trying to solve the state explosion issue (CLARKE; LERDA, 2007). Despite these drawbacks, model checking has been in use by many semiconductor manufacturers for hardware design. With respect to software, there are tools available for accomplishing formal verification using model checking, such as SPIN (HOLZMANN, 2003), Symbolic Model Verifier (SMV/NuSMV) (McMILLAN, 1993), and MAGIC (CHAKI et al., 2004).

Three important concepts are very related to software testing: *fault*, *error*, and *failure*. In this work, fault and error will be defined according to the IEEE Std 610.12-

1990 (IEEE, 1990) and failure in accordance with Laprie and Kanoun (LAPRIE; KANOUN, 1996):

a) Fault: an incorrect step, process, or data definition. For example, an incorrect instruction in a computer program;

b) Error: the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result;

c) Failure: when an error passes through the system-user interface and affects the service delivered by the system.

The term *defect* will be used as a synonym for fault. By reasoning about the above definitions, one can assert that a fault may or may not lead to an error which in turn may or may not lead to a failure. Hence, not always a fault provokes an error because it is likely that a certain part of the source code has never been exercised neither during the testing activities nor after the product was delivered to the customer. Likewise, an error may occur but the user may not perceive the problem and then a failure is not identified.

The simplest definition of testing was given by Myers (MYERS, 2004): "Testing is the process of executing a program with the intent of finding faults[1]". Thus, a good test strategy is the one that finds faults in the IUT. Myers presented a very interesting perspective regarding the psychology of testing and enunciated ten vital software testing principles. For instance, he argues that "a programmer should avoid attempting to test his or her own program". The idea is that a programmer knows exactly what his/her program is supposed to do and may not realize that some faults exist. Moreover, in general, no one wants to find faults in one's own product. It is a destructive feeling.

From previous explanation, software testing relates to Verification and Validation. Besides, a process must be defined for software testing to be effective. Such process comprises a set of activities and organizations can adopt different solutions.

[1]Myers indeed used *errors* over *faults*. However, according to the definition given earlier, it is more adequate to use fault because the main intention of software testing is to find bugs in the source code, i.e. the existing faults.

Typical activities include *Plan Test* (IEEE, 1998), *Generate/Select Test Cases* (MATHUR, 2008), *Execute Test Cases* (SANTIAGO et al., 2008a), *Evaluate Test Results* (the oracle problem) (WEYUKER, 1982) (BINDER, 1999), and *Select Test Cases for Regression Testing* (MATHUR, 2008). This work will detail only the activity *Generate/Select Test Cases* because is the one most related to it. Only a remark concerning with the *Execute Test Cases* activity is the *Qualidade do Software Embarcado em Aplicações Espaciais* (QSEE - Quality of Space Application Embedded Software) research project (SANTIAGO et al., 2007).

In the context of QSEE project, a tool was designed and implemented aiming to automate both test case execution and test process documentation generation (SILVA et al., 2006) (SILVA et al., 2007) (SILVA, 2008). The *QSEE-Teste Automatizado de Software* (QSEE-TAS - QSEE-Software Automated Testing) tool has, among others, the following characteristics (SANTIAGO et al., 2008a): support of functional and robustness (fault injection) testing for embedded software, multiple IUTs testing by means of communication channels using RS-232 and/or USB interface standards, automated test case execution, and automated test process documentation generation in an XML output file. In an empirical evaluation comparing QSEE-TAS with a test execution tool previously developed at *Instituto Nacional de Pesquisas Espaciais* (INPE - National Institute for Space Research), which had little support for automation, and using test suites created for three space-related IUTs, QSEE-TAS decreased in 52.5% regression testing execution time of the same test suite for one IUT compared with the previous tool (SANTIAGO et al., 2008a).

3.1 Test Case Generation

Applying exhaustive testing is not feasible. Thus, one must find a way to select a set of inputs from the input domain of a Program P so that this set can find the maximum number of faults. Test case generation/selection is exactly about this issue.

A *test case* comprises the *test input data* and the corresponding *expected result*. The expected result of a test case is the outcome that is expected to occur when the IUT is stimulated by the test input data. A set of test cases is a *test suite*. The Test Case Specification deliverable then specifies the test input data, the expected results, and a set of execution conditions for a test item (IEEE, 1998).

Generating test cases is probably the most studied testing activity by researchers. A very brief discussion of the types of existing testing techniques related to this activity follows. This discussion is based on the recent book of Mathur who defines five classifiers each of which maps from a set of features to a set of testing techniques (MATHUR, 2008). Two of such classifiers will be addressed: *source of test generation* and *lifecycle phase in which testing takes place.*

Table 3.1 is an adaptation with some slight modifications from Mathur's proposal for the classifier *source of test generation.* Table 3.2 is an adaptation regarding the classifier *lifecycle phase in which testing takes place.*

Table 3.1 - An adaptation from Mathur's definition for the classifier *source of test generation.*

Artifact	**Technique**	**Example**
Requirements (Informal)	Black Box	- Ad hoc testing - Boundary-value analysis - Equivalence partitioning - Category-partition method - Classification trees - Random testing
Requirements (Formal Model)	Model-based Specification	- Statechart testing - Finite State Machine testing - B testing - Z testing - Pairwise testing
Design Documents	Model-based Document	- Unified Modeling Language testing (Sequence, Collaboration, ... Diagrams)
Source Code	White Box	- Control Flow testing - Data Flow testing - Mutation testing

Table 3.1 shows that test cases can be generated from informally or formally specified requirements and without the aid of the source code. This technique is known as *black box testing.* Examples of black box testing techniques based on informal requirements include ad hoc testing and some heuristics like equivalence partitioning, boundary-value analysis and the Category-Partition method (OSTRAND; BALCER, 1988).

As many other Software Enginering research areas, there are different perspectives

Table 3.2 - An adaptation from Mathur's definition for the classifier *lifecycle phase in which testing takes place.*

Phase	**Technique**
Coding	Unit testing
Integration	Integration testing
System Integration	System testing
Prerelease	Acceptance testing
Maintenance	Regression testing

with respect to Model-Based Testing (MBT). Mathur considers that MBT refers to the situations where requirements are formally specified (MATHUR, 2008). However, other authors assert that the testing community tends to consider MBT as a type of testing in which tests are derived from software behavioral models (EL-FAR; WHITTAKER, 2001). The latter definition includes formal model/language specifications and other notations, like UML[2] models (OMG, 2007). This work follows the latter definition and, therefore, the second and third rows in Table 3.1 are considered examples of MBT. MBT is also a form of black box testing. When test cases are derived from design documents, some authors define the technique as *grey box* testing (ABDURAZIK; OFFUTT, 2000). Subsection 3.1.1 will further describe MBT.

Table 3.2 takes into account the fact that testing activities occur during the entire software development lifecycle. When a developer is coding, unit testing can be applied. As a system usually is decomposed into several units, integration testing takes place when such units are integrated. When the entire system has been developed, system testing is applied. A customer can generate his/her own test suite in order to accept the product developed by a supplier. Actually, an independent organization may be in charge to create such tests and apply them. Thus, acceptance testing comes into picture. Regression testing is useful, and mandatory, whenever a new version of a product is created by modifying an existing version. It is impotant to mention that black box techniques can be used for all phases of software lifecycle. However, white box testing is suitable for unit, integration and regression testing but not for system and acceptance testing, because it is difficult in practice to derive tests cases based on source code when the entire system is considered.

[2]Some authors do not consider UML a formal language but rather a semi-formal language.

3.1.1 Model-Based Testing

MBT has drawn lot of attention in both industrial and academic areas in which several models have been used in order to guide test process activities like test case generation and test results evaluation (SANTIAGO et al., 2008b). This subsection will provide an overview of works concerned with MBT.

Briand and Labiche presented the *Testing Object-orienTed systEms with the unified Modeling language* (TOTEM) approach based on UML diagrams addresing functional system testing (BRIAND; LABICHE, 2002). Test requirements are derived from Use Case diagrams, Use Case descriptions, Interaction Diagrams (Sequence or Collaboration) associated with each Use Case, and Class Diagrams (composed of application domain classes and their contracts). In TOTEM, Activity Diagrams can be used to capture sequential dependencies among Use Cases with the aid of application domain experts. Based on these sequential dependencies, legal sequences of Use Cases are built for test case generation.

An approach to system testing based on UML activity diagrams was proposed by Hartmann et al. (2005). The approach is based on the transformation of existing textual use case specifications into UML activity diagrams. Test generation is accomplished using the Category-Partition method (OSTRAND; BALCER, 1988). Despite the automated features, the test designer must manually annotate with stereotypes the resulting UML activity diagrams in order to indicate whether an activity pertains to a user or to the system.

Finite State Machines (FSMs) (LEE; YANNAKAKIS, 1996), Statecharts (HAREL, 1987) (HAREL et al., 1987) and Specification and Description Language (SDL) (DOLDI, 2003) are a few examples of modeling techniques commonly user for testing. At this point, it is important to formally define what is an FSM in the context of this work. Different authors have distinct definitions about FSM and in many cases the term FSM is used as a class of models which encompasses many other state-transition models. In this work, an FSM is a deterministic Mealy machine which can be formally defined as follows (PETRENKO; YEVTUSHENKO, 2005):

Definition 1: An FSM A is a 7-tuple $(S, s_0, X, Y, D_A, \delta, \lambda)$, where:
- S is a finite set of states with the initial state s_0,
- X is a finite set of inputs,

- Y is a finite set of outputs,
- $D_A \subseteq S$ x X is a specification domain,
- δ is a transition function $\delta : D_A \to S$, and
- λ is an output function $\lambda : D_A \to Y$.

Note that there is no set of final states in the above definition. Thus, a Deterministic Finite State Automaton (DFA) basically differs from an FSM because a DFA has a set of final states but it has neither a finite set of outputs (Y) nor an output function (λ). A DFA is used as a regular language acceptor. An FSM is desired when it is necessary to model the dynamics of input/output of a system, although it is possible to use it as an acceptor too. Hopcroft and Ullman define a Determininistic Mealy machine, like the one defined above, as a DFA with output (HOPCROFT; ULLMAN, 1979). However, there are also Finite State Transducers (FSTs) which have not only a set of final states but also a finite set of outputs and a relation which encompasses the roles of both the transition and output functions.

Simplicity is one of the key advantages in using FSM and this technique has been in use for modeling reactive systems and protocol implementations for a long time. Once an IUT is modeled as a state-transition diagram representing an FSM, several test criteria[3] like Transition Tour (TT), Distinguishing Sequence (DS), Unique Input/Output (UIO) (SIDHU; LEUNG, 1989), W (CHOW, 1978), switch cover (1-switch) (PIMONT; RAULT, 1976) and state counting (PETRENKO; YEVTUSHENKO, 2005) can be used to generate test cases. Also, a comparison of fault detection effectiveness of some of these criteria was made in (SIDHU; LEUNG, 1989).

Sinha et al. (2007) demonstrated how a combination of UML use case and class diagrams can be converted to an EFSM. The transformation algorithm translates different use case specific constructs such as *included use cases*, *extension points*, *conditional statements* by accounting for their associated semantics. One issue of their work is that there is no more than one state in the EFSM representing use cases. For testing, it is hard to think of an EFSM with only one single state and a lot of self-loop transitions as proposed in their approach. It is not very clear, and probably not suitable, how to get test cases with a model like this.

A Model-based approach to generate a set of conformance test cases for interactive

[3]Some authors prefer the term *method* rather than *criterion*. However, this work will adopt the latter.

systems, i.e. those which react to operations invoked by external environment, was proposed by Paradkar (PARADKAR, 2003). The approach presents extensions to both the Category-Partition method (OSTRAND; BALCER, 1988) and the *Test Specification Language* (TSL) (BALCER et al., 1989). Test case generation is based on the extraction of a Finite State Automaton (FSA) from a specification written in an extended version of TSL, known as *Specification and Abstraction Language for Testing* (SALT). An environment was developed to support the approach but even the author mentioned that the method was not applied in industrial context with more complex applications.

An algorithm that generates a partition of the input domain from a Z specification has been introduced by Hierons (HIERONS, 1997). This partition can be used both for test case generation and for the production of an FSA. This FSA can then be used to control the testing process. This method generates a large FSA making this approach difficult for test case generation addressing large software systems (PARADKAR, 2003). Singh et al. (1997) proposed an approach for generating test cases from formal specifications written in Z language by combining the classification-tree method for partition testing with the Disjunctive Normal Form (DNF) (DICK; FAIVRE, 1993) approach. Their technique first derives a classification tree describing high level test cases from the Z formal specification of the IUT. Then, the high level test cases are further refined by generating a DNF for them.

Complex software usually presents features like parallel activities and hierarchy. These features are very hard to represent using FSMs, so this leads to considering higher-level techniques as Statecharts. Several approaches have been proposed to generate test cases from Statecharts models. Hong et al. (2000) provides a way to derive EFSMs from Statecharts to devise test criteria based on control and data flow analysis. Binder adapted the W criterion to a UML context and named it round-trip path testing, in which flattening a Statechart is a pre-requisite before using the criterion itself (BINDER, 1999). Santiago et al. (2006) proposed a methodology to transform hierarchical and concurrent Statecharts into FSMs with the support of the PerformCharts tool (VIJAYKUMAR et al., 2006).

Souza proposed a family of testing coverage criteria, the *Statechart Coverage Criteria Family* (SCCF), for models in Statecharts (SOUZA, 2000). Test requirements established by the SCCF criteria are obtained from the Statecharts reachability tree (MASIERO et al., 1994). Antoniol et al. (2002) presented a study whose main goal was to analyze cost and efficiency of the Binder's round-trip path criterion. Briand et al. (2004) showed a simulation and a procedure to analyze cost and efficiency of three criteria proposed by Offutt and Abdurazik (OFFUTT; ABDURAZIK, 1999) and the very same round-trip path.

A system testing approach to coverage of *elementary transition paths* was proposed by Sarma and Mall (SARMA; MALL, 2009). The technique relies on the derivation of a System State Graph (SSG) based on UML 2.0 use case, sequence, and Statecharts diagrams. The test criterion which their method aims to satisfy is *transition path coverage* which states that each elementary transition path p of the SSG must be exercised at least once by a test suite T. Sarma-Mall's work presents some limitations but the most severe one concerns with the combined fragment *loop* (like control structures *while* or *for*) of sequence diagrams. A loop is either not executed at all or it is executed only once. In other words, a loop is reduced to an option (*opt* - like control structure *if*) combined fragment. Thus, the authors did not address one of the major problems in path testing. Howden (1976) stated that, in general, a program containing loops will have an infinite or undetermined number of paths. Hence, testing all the paths of a program is unfeasible. As UML 2.0 sequence diagrams allow to model loops, this fact restricts their approach considerably and the problem of testing applications in the presence of huge number of paths is not properly addressed.

Fröhlich and Link presented a system testing method based on textual descriptions of UML use cases (FROHLICH; LINK, 2000). They translated a use case description into a UML Statechart (UML state machine) and, after that, they applied AI planning techniques to derive test suites satisfying the coverage testing criterion which states that all transitions of the UML state machine must be traversed at least once. Their method sounds interesting but they considered a weak testing criterion. Covering all transitions of an FSM (state model) is one of the weakest test criterion in terms of fault detection effectiveness (CHOW, 1978) (SIDHU; LEUNG, 1989).

Hartmann et al. (2000) presented an integration testing approach based on Statecharts which were used to model components and their interfaces. They defined a strategy to compose Statecharts that model software components as well as an algorithm to reduce the size of the composed Statecharts. Test generation is accomplished using the Category-Partition method (OSTRAND; BALCER, 1988). A TSL (BALCER et al., 1989) test design is created from the Global Behavioral Model, a Statechart obtained after integration of state models representing components. Such Global Behavioral Model is similar to the flat FSM generated by the *Geração Automática de Casos de Teste Baseada em Statecharts* (GTSC - Automated Test Case Generation based on Statecharts) environment (SANTIAGO et al., 2008b) having global states (configurations) formed by active BASIC states of each orthogonal Statecharts component. A limitation of their approach is the fact of not supporting concurrent Statecharts to model each component, a common feature in present day software.

Software modeling for testing purposes may be adapted from non-formal requirements specifications and design documents. Naturally, the greatest benefit occurs when test case generation is accomplished automatically by means, for instance, of environments that support behavioral models. As a result of a cooperation between *Ciências Espaciais e Atmosféricas* (CEA - Atmospheric and Space Sciences) and *Laboratório Associado de Computação e Matemática Aplicada* (LAC - Associate Laboratory of Computing and Applied Mathematics) at INPE, GTSC is an environment that allows test designers to model software behavior using Statecharts and/or FSMs in order to generate test cases automatically based on some test criteria for FSM and some for Statecharts (SANTIAGO et al., 2008b). At present, GTSC has implemented switch cover, UIO and DS test criteria for FSM models and two test criteria from SCCF, all-transitions and all-simple-paths, targeting Statecharts models. In other words, test criteria define the rules that drive test case generation in GTSC.

Figure 3.1 shows the GTSC architecture. Note that besides the architectural elements, it also shows external elements (Reachability Tree, Test Cases, ...) built during the use of GTSC by a test designer.

In order to use the system, a user shall translate the Statecharts behavioral model into an XML-based language named *PerformCharts Markup Language* (PcML) (SANTIAGO et al., 2006). Based on a PcML document, a flat FSM is generated by the Statecharts Flattening element. A flat FSM is a model where all hierarchical and orthogonal features of a Statecharts model were removed. PerformCharts tool (VIJAYKUMAR et al., 2006), one of the components of the Statecharts Flattening element, is responsible for that.

The flat FSM is actually the basis for test case generation. Hence, a test designer

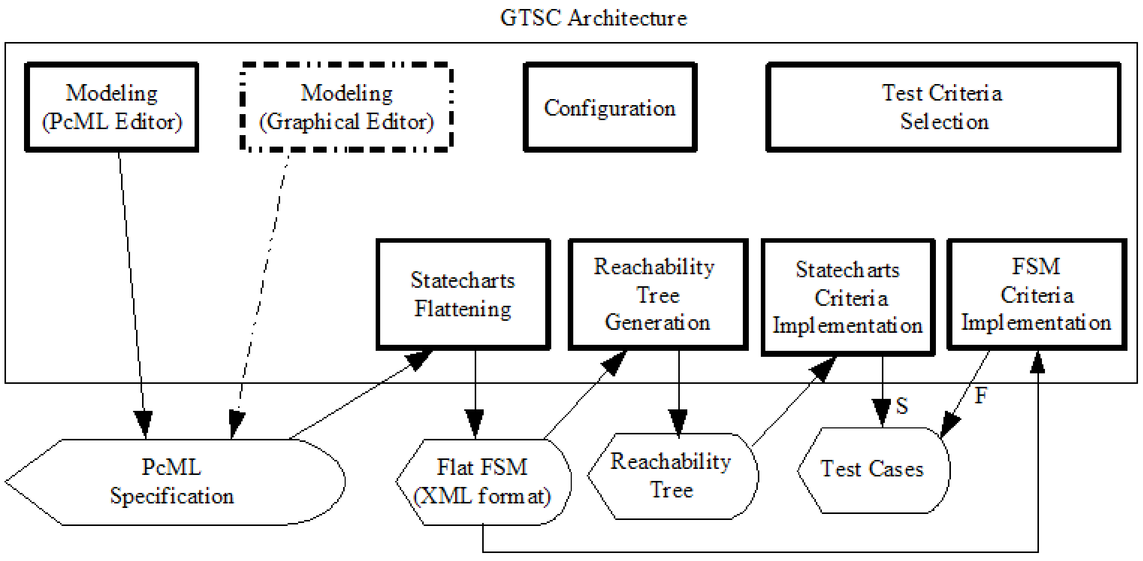


Figure 3.1 - The GTSC Architecture.

may follow two approaches. If a SCCF test criterion will derive test cases, GTSC must adapt the flat FSM to resemble a reachability tree (MASIERO et al., 1994). A reachability tree is only a behavioral representation of Statecharts, showing the possible configurations and paths (sequence of configurations) that the system can reach. SCCF requires a reachability tree in order to generate, at the end of the process, test cases. Thus, based on the selected test criterion of SCCF and on the this tree, test cases are created.

If an FSM test criterion is the option, it is only necessary for the user to choose among the available criteria for FSM and instruct the environment to generate test cases, based on the flat FSM. A final observation is in order to mention that a Statecharts without hierarchy, parallelism, synchronization is nothing more than an FSM (or Extended FSM if guard conditions are used). Hence, it is perfectly possible to provide as input to GTSC an FSM in PcML syntax and use the same FSM test criteria to derive test cases.

GTSC was able to generate flat FSMs with as many as 40 states (configurations) and more than 300 transitions, and test suites with up to 265 optimized test cases, showing its potential scalability for dealing with modeling of complex systems (SANTIAGO et al., 2008b).

3.1.2 Combinatorial Designs

Combinatorial designs are a set of techniques for test case generation which allow the selection of a small set of test cases even when the input domain, and the number of subdomains in its partition, is large and complex (MATHUR, 2008). These techniques have been found to be effective in the discovery of faults due to the interaction of various input variables. Depending on the context, several other names may be found in the literature such as orthogonal designs and pairwise testing. Only the techniques most closely related to this PhD proposal will be shortly described.

Some basic definitions follow. Consider a program P that takes k inputs corresponding to variables $X_1, X_2, ..., X_k$. These input variables are known as *factors*. Each value assignable to a factor is known as a *level*. A set of levels, one for each factor, is a *factor combination* or *run*.

The first technique described is **Orthogonal Arrays**. An Orthogonal Array (OA) is an $N \times k$ matrix in which the entries are from a finite set L of l levels such that any $N \times t$ subarray contains each t-tuple exactly the same number of times. Such an array is denoted by $OA(N,\, k,\, l,\, t)$, where N is the number of *runs*, k is the number of factors, l is the number of levels, and t is the strength of the OA. Pairwise design occurs when $t = 2$.

Orthogonal Arrays assume that each factor, f_i, will be assigned a value from the same set of l levels. This is not realistic and, thus, **Mixed-Level Orthogonal Arrays** (MOAs) come into picture for situations where factors may be assigned values from different sets. Such an array is denoted by $MOA(N,\, {l_1}^{k_1}\, {l_2}^{k_2} \ldots {l_p}^{k_p},\, t)$, indicating N *runs* where k_1 factors are at l_1 levels, ..., k_p factors are at l_p levels. As before, t is the strength.

Both OA and MOA are examples of balanced designs, i.e. any $N \times t$ subarray contains each t-tuple exactly the same number of times. In software testing, the balance requirement is not always essential. Mathur (2008) mentions that if an IUT has been tested once for a given pair of levels, there is usually no need for testing it again for the same pair unless the IUT is known to behave nondeterministically. For deterministic applications, the balance requirement may be relaxed and unbalanced designs are adequate choices.

A **Covering Array** (CA) is an example of unbalanced design. A Covering Array

$CA(N,\ k,\ l,\ t)$ is an $N \times k$ matrix in which the entries are from a finite set L of l levels such that each $N \times t$ subarray contains each possible t-tuple **at least** a certain number of times. In other words, the number of times the different t-tuples occur in the $N \times t$ subarrays may vary. This difference often leads to combinatorial designs smaller in size than orthogonal arrays.

A **Mixed-Level Covering Array**, $MCA(N,\ {l_1}^{k_1}\, {l_2}^{k_2} \ldots {l_p}^{k_p},\ t)$, is analogous to an MOA in that both allow factors to assume levels from different sets. MCA is also an instance of unbalanced designs. MCAs seem to be the most popular choice of combinatorial designs among software testers, because they are generally smaller than MOAs and more adequate for testing.

4 PROPOSAL FOR AUTOMATING MBT AND ANALYSIS OF DEFECTS CONSIDERING NL REQUIREMENTS

This PhD Thesis proposal consists of a methodology with tool support aiming to address the objectives stated in Chapter 1: automated translation of NL requirements into behavioral models to support system and acceptance Model-Based Testing, and to also detect automatically inconsistency and incompleteness in NL requirements. Within this chapter, the SWPDC software product, developed in the context of the QSEE project (SANTIAGO et al., 2007), will be used as a case study. In QSEE project, there were the following computing units: Payload Data Handling Computer (PDC), Event Pre-Processors (EPPs), and On-Board Data Handling (OBDH) Computer.

The methodology as well as its supporting tool are named *Automatizando TesS**tes** BasE**ados em*** ***M**odelos e **A**nálise de DeF**eitos considerA**ndo Requisitos em L**inguagem NA**tural* (SEMAFALA - Automating Model-Based Testing and Analysis of Defects considering Natural Language Requirements). SEMAFALA is summarized in Figure 4.1. The dashed rectangles in Figure 4.1 indicate physical entities that will exist in the context of the methodology. The bold rectangles represent tools that will be developed/adapted and the thin rectangles represent activities that the user shall accomplish to support the methodology. An environment will be developed incorporating all the tools shown in the **Context Independent** part. *CSAO Generator*, *CSAO-to-Statecharts Translator*, and *Defects Analyzer* are components that will be developed. CSAO stands for Control-Subject-Action-Object and it is an extension of the SAO triads mentioned in the work of Fantechi and Spinicci (FANTECHI; SPINICCI, 2005). *Link Grammar Parser* (SLEATOR; TEMPERLEY, 1993), GTSC (SANTIAGO et al., 2008b), and QSEE-TAS (SILVA, 2008) already exist and they will be adapted to compose the environment.

SEMAFALA requires that a user refines the NL requirements specifications. This refinement activity involves checking spelling and grammatical errors, identifying the scenarios for testing, and the sequence of requirements that compose each scenario. In system and acceptance testing, where testing is applied considering the entire software product, a scenario-based approach is recommended . Thus, a test designer usually breaks down the entire system based on functionalities it must provide, and then models are derived to address each functionality. However, a complete usage scenario is usually defined by several requirements which together characterize a

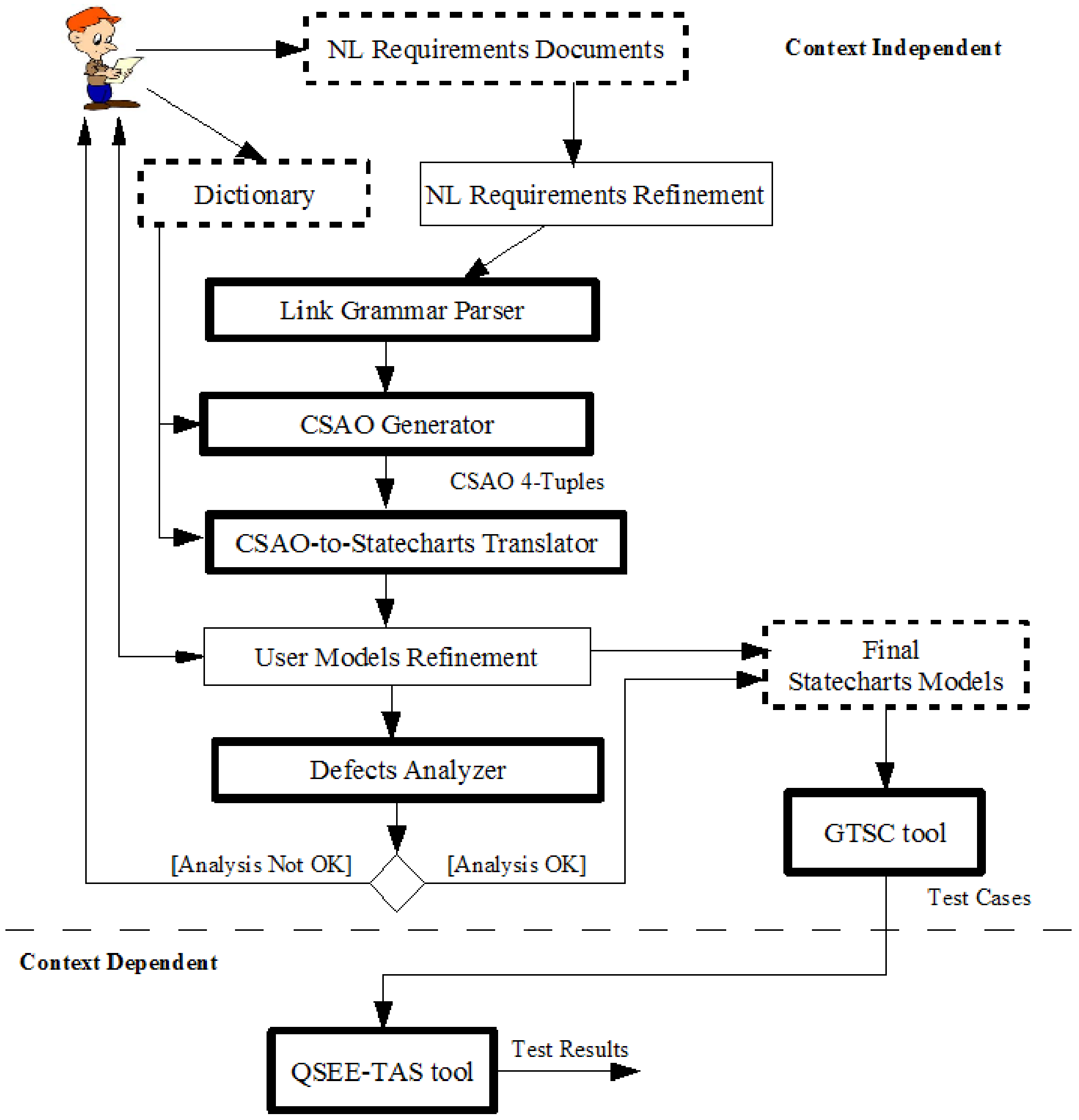


Figure 4.1 - The SEMAFALA Methodology.

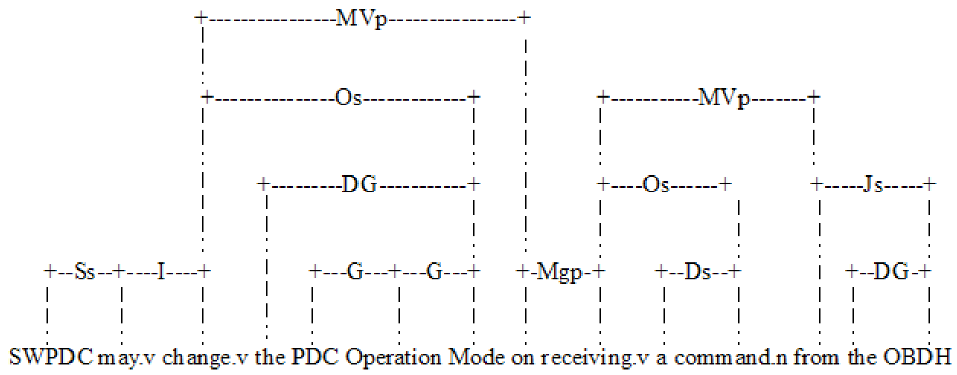
way to stimulate the system. Thus, it is necessary that the environment somehow knows which are the requirements that compose a particular scenario. These are the reasons behind the identification of scenarios and their respective requirements.

The Dictionary defines the application domain and it is considered a triple $\langle N, R, C\rangle$, where N is a set of Names that characterize the domain, R is a set of input/output pairs that represent the Reactiveness of the system, and C is a set of words that characterize specific Control behaviors when automatically translating NL requirements into models (Statecharts) for testing. The user shall define the Names and Reactiveness elements of the Dictionary via a Graphical User Interface (GUI). The Control element will be already within the environment. It is worth to be mentioned that the Reactiveness feature of the Dictionary comes into picture due to the choice of Statecharts as the computational model. However, almost any system can be modeled as a reactive system. It is a matter of perspective. Events shall be sensed by a system so that it can act according to such inputs.

The *Link Grammar Parser* can then generate the set of labeled links connecting pairs of words. Consider the following requirement adapted from the specification of the SWPDC software product:

[TS018] SWPDC may change the PDC Operation Mode on receiving a command from the OBDH.

The output produced by *Link Grammar* is shown in Figure 4.2. Note the link types between pair of words. For instance, "SWPDC" is the subject noun (link type "S") of the verb "may" (the index .v in the sentence means it is a verb). Another example, the verb "change" is connected to the object (link type "O") "Mode".



```
            +--------------MVp--------------+
            |                               |
            +------------Os------------+    |    +-----------MVp-----------+
            |                          |    |    |                         |
            |        +-------DG--------+    |    +------Os-------+         +---Js---+
            |        |                 |    |    |               |         |        |
+--Ss-+--I--+        |   +-G-+----G----+    +-Mgp+           +-Ds+         |    +-DG+
|     |     |        |   |   |         |    |    |           |   |         |    |   |
SWPDC may.v change.v the PDC Operation Mode on   receiving.v a   command.n from the OBDH
```

Figure 4.2 - Output of the *Link Grammar Parser* for requirement [TS018].

In J-RAn tool (FANTECHI; SPINICCI, 2005), the output of *Link Grammar* is accounted for in order to identify SAO triads from the NL sentences. To accomplish this, J-RAn searches for verbs (.v) within the sentence. Once the verb has been detected, it is possible to identify the subject if a link type "S" is detected, and objects if a link type “O” is matched. To finish the process, other words may be added after link types like determiners (“D”). However, such tool did not present a very good performance as mentioned in section 2.1.

In this PhD work, the intention is to implement the extraction mechanism presented above but with several modifications. The first modification is the inclusion of Control features (“C”) in the SAO triad so that it will be transformed into a CSAO 4-tuple. The reasoning behind this relies on the fact that words like *if ...* and sentence of words like *in the case ...* determine a particular behavior in the model generated. For instance, finding an if-then-else situation in one or several NL requirements (e.g. in one requirement: “If the system ...”; in the same or in the next requirement: “On the other hand, ...”) may imply that the behavioral model will have a state with two outgoing transitions each one representing the possible outcome of the if-then-else situation.

The second modification is related to the object identification. A sentence is delimited by a period. A requirement may have more than one sentence producing more than one CSAO 4-tuple per requirement. Actually, even one sentence may have more than one 4-tuple, if there are more than one subject within such sentence. The idea is to use not only the link type “O” but also “OT” and “OX” to identify objects. Besides, other link types regarding determiners (“DD”, “DG”, “DT”, “AL”) will be accounted for. Last but not least, it is possible that there is no explicit object generated by *Link Grammar* depending on the NL requirement. Consider another requirement adapted from the specification of SWPDC:

[TS021] Housekeeping data shall be generated automatically and continually by SWPDC at every 600 s.

By using *Link Grammar* and the guidelines described above, there is no object because none of the link types regarding object (“O”, “OT”, ...) appears in the parser output. However, there are link types such as “MV” which means the connection of verbs and adjectives to modifying phrases that follow (like adverbs), prepositional phrases and other features, and “J” which represents the connection of prepositions to objects. Hence, if there is a subject but no link type representing objects, however, if an “MV” and/or “J” link types exist, an object is identified from the NL sentence. Thus, the requirement [TS021] will have: C = empty; S = “Housekeeping data”; A = “shall be generated”; O = “SWPDC”.

Algorithm 1 CSAO Generator.

```
CSAO_Generator(LinkGrammarOutput, Dictionary){
1  INITIALIZE CSAO to empty
2  FOR each sentence in LinkGrammarOutput
3    WHILE NOT end of sentence
4      SEARCH for an "C" in the labeled links
5      IF "C" is found
6        SET CSAO.Control to words in the sentence corresponding to
           "C" matching Dictionary.Control
7      ELSE
8        SET CSAO.Control to empty
9      END IF
10     SEARCH for an "S" in the labeled links
11     IF "S" is found
12       SET CSAO.Subject to the the word corresponding to "S"
13       SET CSAO.Subject to CSAO.Subject + previous words until a
           determiner ("D","DD","DG","DT","AL") or first word of the
           sentence is encountered
14       FIND for the next ".v" in the sentence after the matched "S"
15       SET CSAO.Action to the word corresponding to ".v"
16       SET CSAO.Action to CSAO.Action + posterior verbs and negation
           adverb (not) but discarding verb-modifying adverbs ("E")
17       SEARCH for the next "O" OR "OT" OR "OX" in the labeled links
18       IF "O" OR "OT" OR "OX" are found
19         SET CSAO.Object to the word corresponding to "O"/"OT"/"OX"
20         SET CSAO.Object to CSAO.Object + previous words until a
             verb (.v) is encountered, discarding determiners ("D",
             "DD","DG","DT","AL") and/or prepositions ("J")
21       ELSE
22         IF "MV" or "J" are found
23           SET CSAO.Object to the words corresponding the most
               enclosing "MV" or "J", discarding determiners ("D",
               "DD","DG","DT","AL") and/or prepositions ("J")
24         END IF
25       END IF
26     ELSE
27       SET CSAO.Control to empty
28       SET CSAO.Subject to empty
29       SET CSAO.Action to empty
30       SET CSAO.Object to empty
31     END IF
32   END WHILE
33 END FOR
34 RETURN(CSAO)}
```

The mechanisms of extracting CSAO 4-tuples from the output of the *Link Grammar Parser* are shown in **Algorithm 1** CSAO Generator. Note that for each sentence in the *Link Grammar* output, the algorithm first searches for a "C" (conjunction) in order to realize whether a control pattern is found in the NL sentence, and sets the Control of the CSAO 4-tuple (lines 4-6). Otherwise, the CSAO Control is set to empty (lines 7-8). After, the algorithm searches for subjects ("S") (line 10) and if it finds, CSAO Subject, Action and Object fields are set (lines 11-23). Note that determiners, prepositions and verb-modifying adverbs are discarded when identifying the elements of the CSAO 4-tuples. If no subject is matched, the tuple is set to empty (lines 26-30).

The *CSAO-to-Statecharts Translator* will take the CSAO 4-tuples as input and will generate the Statecharts models. **Algorithm 2** presents the main ideas regarding this tool. The model generated is a list of structures composed of the following fields: State; TrInput (the input event related to the incoming transition of the State); and TrOutput (the output related to the outgoing transition of the State). The initial idea is to denote the states of the model with the Subject element of the CSAO 4-tuple. This is clearly shown in lines 9, 14 and 18. However, a call to the *Check_Subject_Name* method is made prior to the assignment (lines 8 and 17) in order to realize whether there is already a state in the model with the same name of the current CSAO Subject. The *Check_Subject_Name* method, shown in **Algorithm 3**, will return a new name for the subject just adding an underscore followed by an incrementing number after the subject, if there is already a coinciding state name in the model; otherwise, it will return the same CSAO Subject.

The Reactiveness set of the Dictionary plays an important role on defining the TrInput and TrOutput values. As shown in lines 21 to 26 of **Algorithm 2**, if the Object of the CSAO 4-tuple exists in the input element of the Reactiveness set, then TrInput will be assigned to such value and TrOutput will have the value of the corresponding output element of the set. This is because in reactive systems, the input/output is usually well defined in terms of entities, for instance, communication protocols. However, if the tuple Object does not match any value of the input element of the Reactiveness set, TrInput will be formed by a combination of Action_Object of the CSAO 4-tuple, and TrOutput will be empty. Another remark of **Algorithm 2** is to mention that the if-then-else situation in NL sentences is addressed from lines 5 to 15. Hence, more than one transition may be leaving the same state in the resulting model.

Algorithm 2 CSAO-to-Statecharts Translator.

```
CSAO_to_Statecharts_Translator(CSAO tuples, Dictionary){
INITIALIZE cont to FALSE
INITIALIZE lastcstate, exist_cont, exist_subj to empty
INITIALIZE Model to empty
FOR each tuple in CSAO tuples
  IF tuple.Control exists in Dictionary.Control AND cont = FALSE THEN
    SET exist_cont to tuple.Control
    SET exist_subj to tuple.Subject
    CALL Check_Subject_Name(tuple.Subject, Model)
    SET Model.State to newtuple.Subject
    SET lastcstate to newtuple.Subject
    SET cont to TRUE
  ELSE
    IF tuple.Control = exist_cont AND tuple.Subject = exist_subj THEN
     SET Model.State to lastcstate
      SET cont to FALSE
    ELSE
      CALL Check_Subject_Name(tuple.Subject, Model)
      SET Model.State to newtuple.Subject
    END IF
  END IF
  IF tuple.Object exists in Dictionary.Reactiveness.Input THEN
    SET Model.TrInput to the matched Dictionary.Reactiveness.Input
    SET Model.TrOutput to the matched Dictionary.Reactiveness.Output
  ELSE
    SET Model.TrInput to tuple.Action_Object
    SET Model.TrOutput to empty
  END IF
END FOR
CALL Refine_Model(Model, Dictionary)
CALL Draw_Model(PcMLModel)}
```

After all tuples were translated, the *Refine_Model* method (**Algorithm 4**) is called within the *CSAO-to-Statecharts Translator* (**Algorithm 2**). *Refine_Model* is not the User Models Refinement activity shown in Figure 4.1. It is an automated refinement aiming to eliminate unnecessary states and transitions of the model, to rename certain states of the model, and to transform the 3-field structures into 4-field ones

Algorithm 3 Check_Subject_Name.

```
Check_Subject_Name(name, model){
1  INITIALIZE counter to 1
2  IF name already exists in model.State
3    IF name has an underscore
4      GET number after underscore
5      SET counter to number + 1
6    ELSE
7      INCREMENT counter
8    END IF
9    RETURN(name_counter)
10 ELSE
11   RETURN(name)
12 END IF}
```

by including the destination state of each transition. As shown in lines 3 to 5 of **Algorithm 4**, the method removes a state and its respective input event (TrInput) if the state does not exist in the Name set of the Dictionary, and the input event does not exist in such set, and the input event does not exist either in the input set of the Reactiveness element of the Dictionary. This is done because not all NL sentences contain relevant information to justify the creation of a state and its transition.

However, the name of a state may be changed if TrInput exists in the Dictionary Name set and, at the same time, it does not exist in the input set of Reactiveness (see lines 7 to 13 of **Algorithm 4**). This is explained due to the fact that the subjects, which in turn first generated the name of states in the model, in NL requirements are usually a few names like system, the name of a computer or a software product, and so on. This implies that the name of the states would basically be limited to those names added by a counter (see the *Check_Subject_Name* algorithm) such as system, system_1, system_2, and so on. In order to avoid this situation and to provide more meaningful names for states, the new name of the state is changed to a word or sentence of words that are in the Name set of the Dictionary. Hence, this set will provide the name of all the states within the refined model. Also note that what is searched in the Dictionary Name set is only the Object part of TrInput (see line 25 of **Algorithm 2**); the Action part is not considered.

Algorithm 4 also transforms the 3-field structures into 4-field ones (trmodel) by

Algorithm 4 Refine_Model.

```
Refine_Model(model, dictionary){
1  INITIALIZE trmodel to empty
2  FOR each node n in model
3    IF model.State(n) does not exist in dictionary.Name AND
       model.TrInput(n) does not exist in dictionary.Name AND
       model.TrInput(n) does not exist in dictionary.Reactiveness.Input
4      REMOVE model.State(n)
5      REMOVE model.TrInput(n)
6    ELSE
7      IF model.TrInput(n) matches at least one entry of the
         dictionary.Name AND model.TrInput(n) does not exist in
         dictionary.Reactiveness.Input
8        IF exactly one entry of the dictionary.Name is matched
9          SET model.State(n) to the matched entry of the
             dictionary.Name
10       ELSE
11         SET model.State(n) to matched entries of the
             dictionary.Name separated by underscore
12       END IF
13       REMOVE underscore and following words from model.TrInput(n)
14     END IF
15     SET trmodel.Srcstate(n) to model.State(n)
16     SET trmodel.TrInput(n) to model.TrInput(n)
17     SET trmodel.TrOutput(n) to model.TrOutput(n)
18     IF n is the last node of model
19       SET trmodel.Deststate(n) to model.State(0)
20     ELSE
21       SET trmodel.Deststate(n) to model.State(n+1)
22     END IF
23   END IF
24 END FOR
25 TRANSLATE trmodel to PcML
26 RETURN(PcMLtrmodel)}
```

adding the destination state of a transition (see lines 15 to 21). This is to simplyfy the translation of the refined model to PcML which is the input language of the GTSC environment (see section 3.1.1). One last remark of the *Refine_Model* method refers to the identification of the destination state (Deststate) of the transition. Note that when reaching the last node of model (line 18), the destination state of trmodel is set to the first (initial) state. The FSM test criteria implemented in GTSC, DS, UIO and switch cover, demand that the Flat FSM is initially connected. In such a kind of machine it is possible to go back to the initial state from any state of the FSM. This explains line 18. Even though the Statecharts criteria implemented in GTSC, all-transitions and all-simple-paths, do not have such a demand, this action turns the model translated from NL requirements more generic in the sense that a test designer may choose any of the five GTSC test criteria to generate the test suite.

The *Draw_Model* method (**Algorithm 2**) will show in a GUI the model returned from *Refine_Model*. *Draw_Model* will do it just adding states, incoming and outgoing transitions following the sequence of nodes in the refined model. A user can then make a second refinement in the model probably changing the name of input events of transitions and even the name of states. Hence, the Final Statecharts models can be input to the GTSC environment for test case generation if the user does not want to make any analysis related to defects of NL requirements. In the **Context Dependent** part, the QSEE-TAS tool can be the one to automatically execute the test cases and provide preliminary test results. It is important to mention that the test cases generated by GTSC can be used to several other test case execution tools.

4.1 Defects Analyzer

Before generating test cases, the user may think it is important to analyze whether the NL requirements have issues like inconsistency and incompleteness. The *Defects Analyzer* component then takes place, and its main features are described as follows.

Instead of translating NL requirements into some sort of logic, like the propositional one (GERVASI; ZOWGHI, 2005), SEMAFALA will handle inconsistency by examining the Statecharts models generated. Hence, two types of inconsistencies are envisaged:

a) logical contradiction. In this case, the model will be examined to realize whether there is an external event (input event), e, within a transition and its negation, $\neg e$, leaving from the same source state and reaching the same destination state;

b) non-determinism inconsistency. Suppose a state s_i and also that the same external event, e, within a transition goes from s_i to two different states, s_j and s_k. Hence, a non-determinism inconsistency is characterized.

In terms of automated detection of inconsistencies, CARL (GERVASI; ZOWGHI, 2005) addresses logical contradiction but not non-determinism inconsistency. CIRCE (AMBRIOLA; GERVASI, 2006) (AMBRIOLA; GERVASI, 1997) deals with inconsistencies by examining, for instance, Data Flow Diagrams (DFDs). The tool identifies conflicts in requirements but it is not evident if such conflicts fall into the two categories proposed above. Besides, the authors even mentioned that CIRCE discovers only a limited class of conflicts (AMBRIOLA; GERVASI, 1997). Kim and Sheldon (KIM; SHELDON, 2004) deals with non-determinism inconsistency by examining Statecharts and Activity Charts but they do not mention logical contradiction. Furthermore, their approach is entirely manual. The conclusion is that SEMAFALA will make a contribution due to the fact of detecting both types of inconsistencies and in an automated manner.

Incompleteness will be approached in SEMAFALA using a different perspective from the works presented in section 2.1. CIRCE detects incompleteness by searching for unsued data (variables that are never modified), and data coming from nowhere in DFDs. Kim and Sheldon propose to search for absorbing states/activities in Statecharts/Activity Charts. These are valuable insights towards incompleteness detection. However, the problem of incompleteness is much more complex so that analyzing visual diagrams is not enough. A different method is required.

In SEMAFALA, combinatorial designs (MATHUR, 2008) will help to address the incompleteness issue. The user is required to define the factors and their corresponding levels. Then, *Mixed-Level Covering Arrays* (MCAs) will be determined in accordance with the information the user provided. In order to generate MCAs, two choices exist. The first option is to reuse tools that are open source such as the *Test Configuration Generator* (TConfig) (UNIVERSITY OF OTTAWA, 2008). TConfig has implemented *In-Parameter-Order* (IPO), a procedure that can generate MCAs. Hence, the SEMAFALA tool might incorporate TConfig as one of its component. Alternatively, Mathur proposes an IPO procedure for the same purpose, but considering only pairwise designs (MATHUR, 2008). This procedure might be implemented within SEMAFALA.

Basically, each *run* (*factor combination*) of the generated MCA will drive an algorithm in SEMAFALA. The idea is that if a *run* is "matched" within a set of NL requirements, then it is said that the NL requirements are complete with respect to this *run*. Otherwise, they are incomplete. Let $f_i, 1 \leq i \leq k$ be a set of factors, and $l_{ij}, 1 \leq j \leq n$ be the set of levels for each factor f_i, where n may vary depending on i. The algorithm will realize whether there are words, or sentences of words, in the set of NL requirements which match the levels l_{ij} of each *run*. Besides, there will be a priority factor, f_1, and corresponding levels, l_{1j}, based on which the algorithm will evaluate if the other levels of the *run*, $l_{ij}, 2 \leq i \leq k$, are satisfied within the NL requirements in a pairwise manner. Hence, the strength, t, chosen is 2 (i.e. pairwise design).

In order to exemplify how this approach works, consider Table 4.1 where an instance of factors and levels for the SWPDC product is shown. The levels' names were abbreviated as follows (OpMode stands for Operation Mode, and Cmd for Commands):

a) Initiation = "Initiation Operation Mode" $\vee$ "Initiation Mode", Safety = "Safety Operation Mode" $\vee$ "Safety Mode", Nominal = "Nominal Operation Mode" $\vee$ "Nominal Mode", Diagnosis = "Diagnosis Operation Mode" $\vee$ "Diagnosis Mode";

b) Tx Sci Data = "transmit scientific data" $\vee$ "TX_DATA-SCIENTIFIC", Tx Hk Data = "transmit housekeeping data" $\vee$ "TX_DATA-HOUSEKEEPING", Load Prog = "load program" $\vee$ "LOAD_DATA", Ver Mode = "verify operation mode" $\vee$ "VER_OP_MODE";

c) Memory Mgmt = "memory managament";

d) Simple = "simple error", Double = "double error", Watchdog = "watchdog error".

Notice the disjunction ($\vee$) in the description above. This implies that the same level may be identified by different terms, e.g. "transmit scientific data" or the mnemonics of the command, "TX_DATA-SCIENTIFIC". Moreover, it is also possible to include conjunction ($\wedge$) when defining a level, and even to mix it with disjunction resulting

Table 4.1 - An example of factors and levels for the SWPDC case study.

Factors	**Levels**			
OpMode	Initiation	Safety	Nominal	Diagnosis
Cmd	Tx Sci Data	Tx Hk Data	Load Prog	Ver Mode
Storage	Memory Mgmt	No		
Error	Simple	Double	Watchdog	

in a complex formula. A GUI in SEMAFALA will enable the user to define a level such as:

$time \wedge housekeeeping \wedge minimum \wedge (millisecond \vee milliseconds \vee second \vee seconds \vee minute \vee minutes)$.

One possible *run* of the MCA is {Safety, Ver Mode, No, Simple}. The translation of this *run* to the algorithm is:

- having the NL requirement generated at least one CSAO 4-tuple, and if such requirement matches the expression “Safety Operation Mode” or “Safety Mode”, does it also match the expression “verify operation mode” or “VER_OP_MODE”?

- having the NL requirement generated at least one CSAO 4-tuple, if an NL requirement matches the expression “Safety Operation Mode” or “Safety Mode”, does it NOT match the expression “memory management”?

- having the NL requirement generated at least one CSAO 4-tuple, if an NL requirement matches the expression “Safety Operation Mode” or “Safety Mode”, does it also match the expression “simple error”?

Demanding the existence of at least one CSAO tuple aims to better elaborate the search mechanism. Rather than simply searching for a pattern of words in a requirement, the algorithm in SEMAFALA will verify whether a CSAO tuple can de derived from the requirement. This prevents the algorithm to consider poorly elaborated requirements, i.e. requirements without even a subject. Hence, the incompleteness algorithm of the *Defects Analyzer* component will make use of the *Link Grammar Parser* and also the CSAO Generator (**Algorithm 1**). To avoid confusion, these linkages were omitted in Figure 4.1.

Note that the priority factor is OpMode. To be considered complete against this *run*, all three conditions must be satisfied. If any of the conditions is not matched, SEMAFALA will report to the user the incompleteness. Furthermore, the need to search for the priority level, i.e. “Safety Operation Mode” or “Safety Mode”, within the NL requirements may be optional. It depends on the structure of the NL specification. For instance, in SWPDC specification, there is a subsection “PDC Requirements of the Safety Operation Mode”. Hence, all requirements regarding this subsection obviously refer to the Safety Operation Mode.

Two reasons motivated the choice of combinatorial designs for dealing with incompleteness. First, to decrease the number of combinations which is the main goal of combinatorial designs, and even doing so covering the interaction of factors (in SEMAFALA, pairwise interaction). In Table 4.1, there are 96 *factor combinations* if an exhaustive approach is used. However, there are only 17 *factor combinations* by applying the IPO implemented in TConfig (UNIVERSITY OF OTTAWA, 2008), i.e. it is an $MCA(17,\ 2^1\ 3^1\ 4^2,\ 2)$.

The second reason is to provide hints to the test designer to identify the scenarios for system and acceptance test case generation. Thus, the pairwise design philosophy implemented in SEMAFALA will help the user by considering each *run* as a scenario. In the example above, the *run* {Safety, Ver Mode, No, Simple} directs the user to define a scenario, and to identify the corresponding requirements, which covers the Safety Operation Mode where the command to verify the current operation mode of the computer is present, and to realize if there is any kind of memory management accomplished in such operation mode, and finally to wonder whether the system is capable to handle the occurence of simple errors in the very same mode. Naturally, the user may even decompose the *run* into several scenarios by combining the levels. In other words, one scenario might cover “Safety Operation Mode” and “verify operation mode”, another “Safety Operation Mode” and “simple error”, and so on.

This second reason derives an alternate workflow within SEMAFALA. In the basic workflow (Figure 4.1), the sequence of actions performed is: models generation, inconsistency, and incompleteness analysis. On the other hand, the alternate workflow presents the following sequence: incompleteness analysis, models generation, and inconsistency analysis.

SEMAFALA will adopt the *tolerating inconsistency* (BALZER, 1991) and also the *tolerating incompleteness* approaches. Hence, requirements can evolve and the user will have the opportunity to resolve such issues when it is more adequate within the software development lifecycle.

4.2 Preliminary Results

This section presents the preliminary results by the application of the methodology proposed in this PhD Thesis. Considering the SWPDC software product, it is assumed that the user refined the NL requirements shown in Figure 4.3, i.e. grammatical errors were checked and such requirements form one scenario for MBT. These requirements were refined from two deliverables developed in the context of the QSEE project (SANTIAGO et al., 2007): Technical Specification (TS) and Requirements Baseline (RB).

```
[TS025] PDC shall be powered on by the Power Conditioning Unit
via an OBDH command.

[TS003] After being powered on, the PDC will be in the Initiation
Operation Mode. SWPDC shall then accomplish a POST. If PDC
presents any irrecoverable problem, this computer shall remain
in the Initiation Operation Mode and such a problem shall not
be propagated to the OBDH.

[TS012] If PDC does not present any irrecoverable problem,
after the initiation process, SWPDC shall automatically change
to Safety the PDC Operation Mode.

[RB005] The OBDH shall send a VER_OP_MODE command to PDC.

[RB064] The PDC shall be in the Safety Operation Mode in order
to be switched off.
```

Figure 4.3 - SWPDC NL requirements characterizing a scenario.

Figure 4.4 shows a piece of the Dictionary concerned with the SWPDC case study. Recall that the sets Name and Reactiveness will be provided by the test designer/requirements analyst while Control is already defined within the tool

```
<Name> = PDC, SWPDC, Initiation Operation Mode, Initiation Mode,
 Initiation , Safety Operation Mode, Safety Mode, Safety , Nominal
 Operation Mode, Nominal Mode, Nominal, Diagnosis Operation Mode,
 Diagnosis Mode, Diagnosis , EPP, EPPs, EPPH1, EPPH2, EPP1, EPP2,
 ...

<Reactiveness> = HANDLE_HW_RST_PROC / CMD_REC,
                 SND_CLOCK / CLOCK_DATA,
                 VER_OP_MODE / INFO_OP_MODE,
                 TX_DATA_SCIENTIFIC / SCIENTIFIC_DATA,
                 TX_DATA_HOUSEKEEPING / HOUSEKEEPING_DATA,
                  ...

<Control> = if , in the case , ...
```

Figure 4.4 - A piece of the Dictionary for the SWPDC case study.

supporting the methodology. In Reactiveness, the term on the left side of / is the input and the one on the right side of / is the output. These are commands (inputs) the OBDH can send to PDC and the corresponding expected responses (outputs) the PDC would send back. This command/response behavior is completely described in a communication protocol that shall be implemented in both computers.

Following the methodolody, *Link Grammar Parser* and *CSAO Generator* together will produce the CSAO 4-tuples. Figure 4.5 shows the tuples derived according to the requirements in Figure 4.3. Overall, 10 CSAO tuples were generated. As previously mentioned, one requirement may derive more than one tuple like requirements [TS003] and [TS012] which were responsible for 5 and 2 tuples respectively. Furthermore, even one sentence may generate more than one tuple. For instance, the last 3 tuples of requirement [TS003] were due to its last sentence: *If PDC presents any irrecoverable problem, this computer shall remain in the Initiation Operation Mode and such a problem shall not be propagated to the OBDH.*

Having obtained the CSAO tuples, the *CSAO-to-Statecharts Translator* can generate the Statecharts models. The results by applying the algorithms presented earlier in this chapter are presented in Figure 4.6 and Figure 4.7. Figure 4.6 shows the raw transformation of the CSAO tuples to the model, i.e. before the *Refine_Model* method is called. Note that some states (PDC, PDC_2, PDC_3, ...) differ their names only due to the application of the *Check_Subject_Name* method (**Algorithm 3**).

```
[TS025]
C = −; S = PDC; A = shall be powered; O = Power Conditioning
                                                        Unit

[TS003]
C = −; S = PDC; A = will be; O = Initiation Operation Mode
C = −; S = SWPDC; A = shall accomplish; O = POST
C = if; S = PDC; A = presents; O = irrecoverable problem
C = −; S = computer; A = shall remain; O = Initiation Operation
                                                         Mode
C = −; S = problem; A = shall not be propagated; O = OBDH

[TS012]
C = if; S = PDC; A = does not present; O = irrecoverable problem
C = −; S = SWPDC; A = shall change; O = Safety PDC Operation
                                                      Mode

[RB005]
C = −; S = OBDH; A = shall send; O = VER_OP_MODE command

[RB064]
C = −; S = PDC; A = shall be; O = Safety Operation Mode
```

Figure 4.5 - CSAO 4-tuples generated according to the NL requirements in Figure 4.3.

Besides, PDC_3 occurs twice in the model. This is because the if-then-else situation encountered in sentences of requirements [TS003] and [TS012]. Thus, in order to represent two transitions leaving from the same state, it is necessary to repeat the name of PDC_3 in the model created by *CSAO-to-Statecharts Translator*. Besides, only one transition has input and output in accordance with the Reactiveness element of the Dictionary (VER_OP_MODE / INFO_OP_MODE). This happens because there is only one tuple whose Object matches an input of the Dictionary Reactiveness set (see tuple of [RB005] in Figure 4.5). The remaining transitions have only input events formed by the concatenation of tuple's Action_Object; they have no outputs.

The model in Figure 4.7 is derived after the application of the *Refine_Model* method (**Algorithm 4**). Some states in the original model (Figure 4.6) have their name changed because their input event matched some entry of the Name set of the Dictionary. Precisely, it is not the entire name of the input event that is evaluated in **Algorithm 4** but only the name after the underscore, which corresponds to the

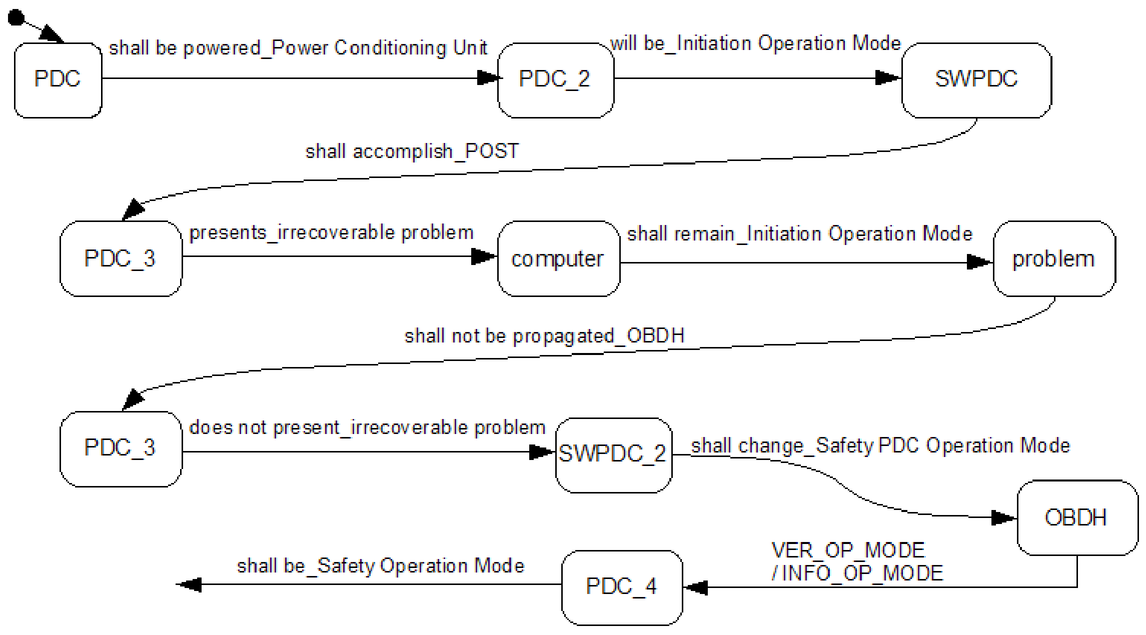

shall be powered_Power Conditioning Unit
will be_Initiation Operation Mode
PDC
PDC_2
SWPDC
shall accomplish_POST
PDC_3
presents_irrecoverable problem
computer
shall remain_Initiation Operation Mode
problem
shall not be propagated_OBDH
PDC_3
does not present_irrecoverable problem
SWPDC_2
shall change_Safety PDC Operation Mode
OBDH
shall be_Safety Operation Mode
PDC_4
VER_OP_MODE
/ INFO_OP_MODE

Figure 4.6 - Statechart model before calling the *Refine_Model* method.

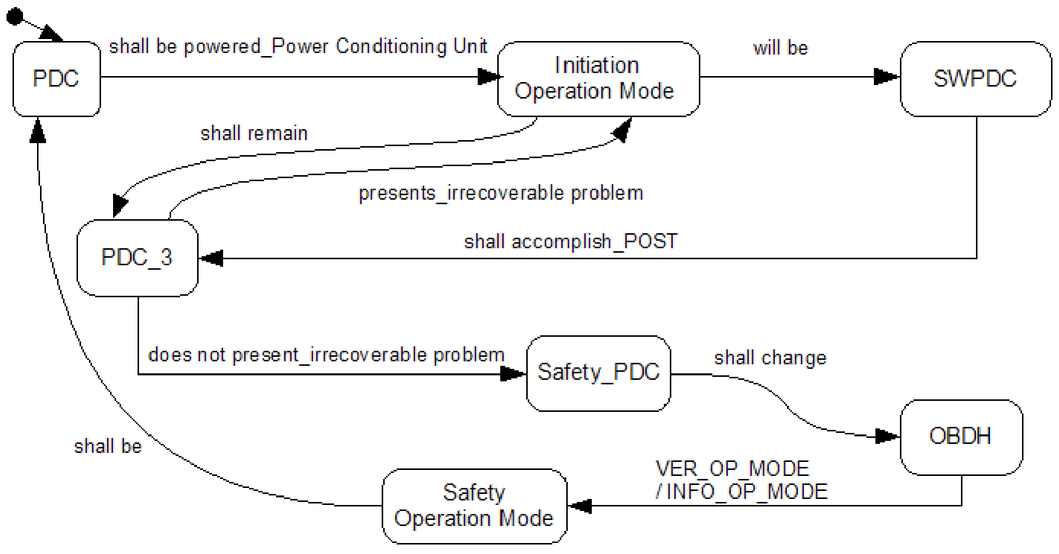

PDC
shall be powered_Power Conditioning Unit
Initiation Operation Mode
will be
SWPDC
shall remain
presents_irrecoverable problem
PDC_3
shall accomplish_POST
does not present_irrecoverable problem
Safety_PDC
shall change
OBDH
shall be
Safety Operation Mode
VER_OP_MODE
/ INFO_OP_MODE

Figure 4.7 - Statechart model after calling the *Refine_Model* method.

Object of the CSAO tuple. As a result, PDC_2 was renamed to *Initiation Operation Mode*, *computer* was replaced by *Initiation Operation Mode* too, and SWPDC_2 became Safety_PDC. These new state names are more relevant considering the modeling of this reactive system. Moreover, state *problem* and its transition in the original model were removed because neither *problem* nor OBDH (in the input event) exists in the Name set of the Dictionary. However, the OBDH state was kept because its outgoing transition input (VER_OP_MODE) is in the input of the Reactiveness set, clearly showing that information related to reactiveness shall not be discarded at all. Finally, when using GTSC to generate test cases, the resulting Flat FSM will be initially connected because of the transition *shall be* from *Safety Operation Mode* to PDC (initial state).

The user may then change the refined model (Figure 4.7) to its taste. For instance, he/she can rename the *shall be powered_Power Conditioning Unit* event to a more concise term such as *switch_PDCon*. State names may be also altered. This is the second refinement proposed in SEMAFALA.

4.2.1 Analysis of Defects

Suppose that requirement [TS012] in Figure 4.3 is changed so that it becomes [TS012a]:

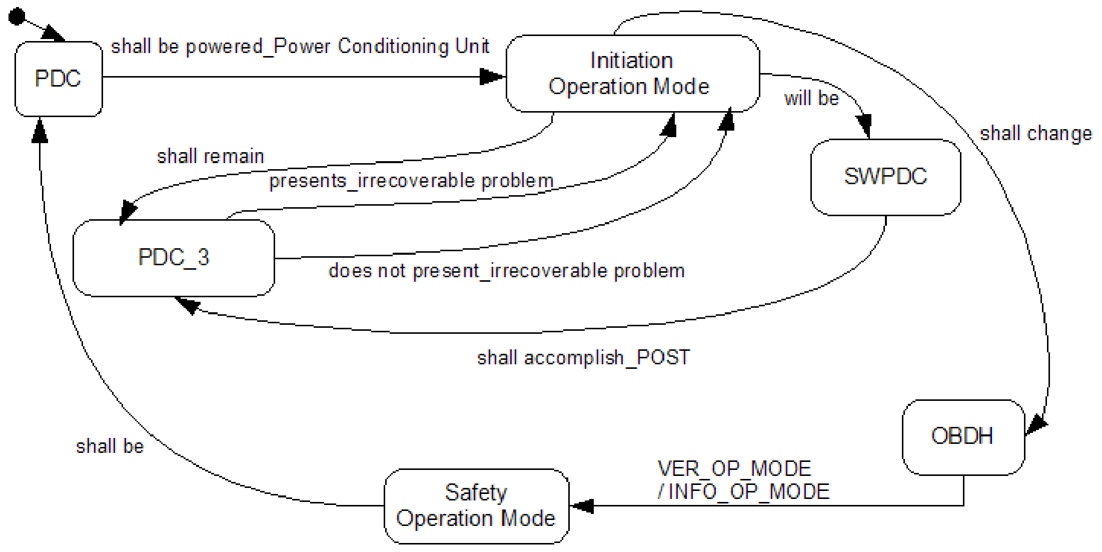
[TS012a] If PDC does not present any irrecoverable problem, after the initiation process, SWPDC shall automatically change to the Initiation Operation Mode.

A logical contradiction exists between requirements [TS003] and [TS012a]. The reason is that no matter what PDC presents (external event e) or does not present (external event $\neg e$) an irrecoverable problem, the PDC Operation Mode will be Initiation. By applying the 4 algorithms presented earlier in this chapter, the resulting Statechart model is shown in Figure 4.8. The SEMAFALA tool can then inspect the model and detect the problem.

Now suppose another version of requirement [TS012], named [TS012b]:

[TS012b] If PDC presents any irrecoverable problem, after the initiation process, SWPDC shall automatically change to Safety the PDC Operation Mode.

At this time, a non-determinism inconsistency is detected as shown in Figure 4.9. The input event *presents_irrecoverable problem* goes from state PDC_3 to two different states: *Initiation Operation Mode* and Safety_PDC.



PDC
shall be powered_Power Conditioning Unit
Initiation Operation Mode
will be
SWPDC
shall change
shall remain
presents_irrecoverable problem
PDC_3
does not present_irrecoverable problem
shall accomplish_POST
OBDH
shall be
VER_OP_MODE
/ INFO_OP_MODE
Safety Operation Mode

Figure 4.8 - A logical contradiction.

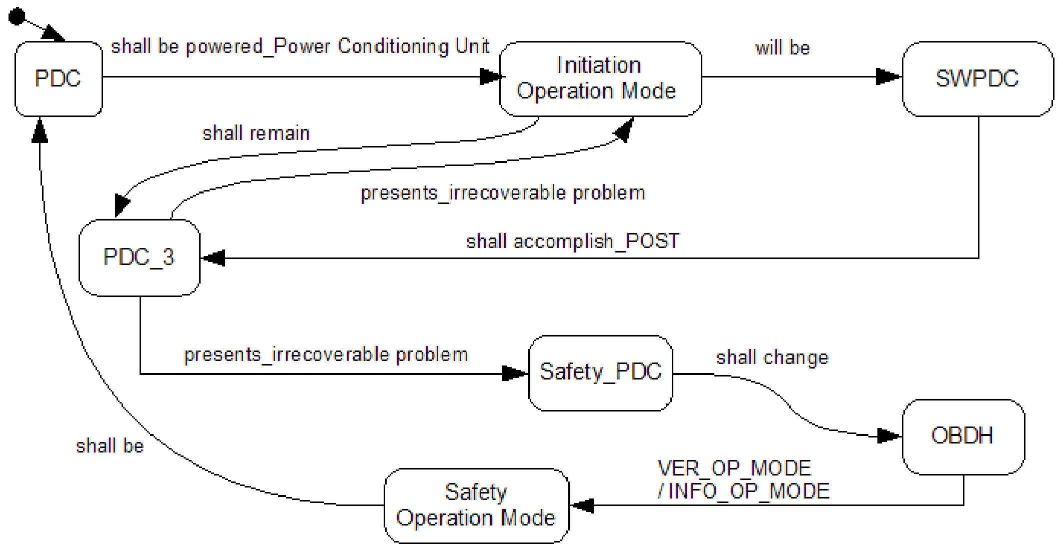


Figure 4.9 - A non-determinism inconsistency.

The ideas behind the SEMAFALA methodology/tool to detect inconsistency tend to be very suitable for analyzing larger NL requirements specifications. SEMAFALA does not rely on very complex reasoning mechanisms but in a plain analysis of the models generated. Naturally, more complex models and forms of non-determinism, e.g. when hierarchy and parallelism are present, need a more careful investigation to detect such conflicts.

The attention now is turned to the problem of incompleteness. As mentioned in section 4.1, combinatorial designs help to address this issue in SEMAFALA. Table 4.2 shows another example of factors and levels for the SWPDC case study.

Table 4.2 - Factors and levels for the incompleteness analysis of the SWPDC case study.

Factors	**Levels**					
OpMode	Nom	Init	Safe	Diag		
Services	Sci	Hk	Dmp	Load	Dg	Tst
Cmd	TxSci	PrpHk	TxHk	VOpM	LdDat	ExeP
Storage	MemMg	No				
HkTime	Std	Min	Max			

In Table 4.2 factors OpMode, Cmd, and Storage are the same as in Table 4.1. Levels of OpMode and Storage are also the same, but some commands are different (6 out of 37 possible commands defined in the communication protocol between PDC and OBDH computers were considered). Services refer to the capability of SWPDC to obtain, generate, format and transmit scientific (Sci), houkeseeping (Hk), dump (Dmp), diagnostic (Dg), and test (Tst) data, as well as to load and execute new programs (Load) during satellite's operation. In particular, the standard (Std), minimum (Min), and maximum (Max) levels of the houseekeeping time (HkTime) factor are formulae as described below:

Std: $time \wedge housekeeeping \wedge standard \wedge (millisecond \vee milliseconds \vee second \vee seconds \vee minute \vee minutes)$;

Min: $time \wedge housekeeeping \wedge minimum \wedge (millisecond \vee milliseconds \vee second \vee seconds \vee minute \vee minutes)$;

Max: $time \wedge housekeeeping \wedge maximum \wedge (millisecond \vee milliseconds \vee second \vee$

seconds $\vee$ *minute* $\vee$ *minutes*).

Using the IPO implemented in TConfig (UNIVERSITY OF OTTAWA, 2008), the generated *Multi-Level Covering Array* is $MCA(36,\ 2^1\ 3^1\ 4^1\ 6^2,\ 2)$. In other words, 36 *runs* were generated. Table 4.3 shows the resulting MCA.

SEMAFALA would classify 20 *runs* as incomplete. However, 6 of these are not. This is the case of *run* 2 where the problem is matching No memory management (Storage factor) in the Nominal (Nom) Operation Mode. In the SWPDC requirements specification there is a requirement asserting that memory management shall be made to store scientific, housekeeping, and dump data. However, no requirement states that memory management shall not be done for diagnosis and test data. However, the latter two are not handled in the Nominal Operation Mode and, hence, *run* 2 shall not be regarded as an incompleteness.

Runs number 4, 14, 19, 27, and 28 are other examples of mistakenly detected incompleteness. At this time, the issue is the command to verify the current operation mode of PDC (VOpM in the Cmd factor). Cmd factor represents a subset of commands defined in the protocol specified for PDC⇔OBDH communication. It is not mandatory for all such commands to appear in the software requirements specification and specifically to be related to a particular PDC operation mode. In the specification there is a requirement that references the communication protocol document, and many commands defined are general-purpose requests. Hence, the fact of VOpM was not found in the Nominal (Nom), Safety (Safe), and Diagnosis (Diag) modes does not imply an incompleteness.

However, the remaining 14 *runs* are real incompletenesses and they can be broadly divided into two categories. The first category is related to the Initiation (Init) Operation Mode. *Runs* from 7 to 12, 26 and 32 are the ones concerned with this problem. The specification is really poorly elaborated regarding this mode, and even a subsection does not exist for it. It is mentioned that a processing called POST shall be accomplished once the PDC is in the mode. Besides, it is said that the PDC is available to communicate with OBDH only after 1 minute has elapsed since the initiation process. On reading that, a developer might infer that during this time the SWPDC shall reject all commands (Cmd factor) received from the OBDH. However, it would be more complete if the requirements explicitly provide these details. The other issue refers to the Storage factor. It is clear in the specification that none

Table 4.3 - MCA considering factors and levels of Table 4.2.

Run	**OpMode**	**Services**	**Cmd**	**Storage**	**HkTime**
1	Nom	Sci	TxSci	MemMg	Std
2	Nom	Hk	PrpHk	No	Min
3	Nom	Dmp	TxHk	MemMg	Max
4	Nom	Load	VOpM	MemMg	Min
5	Nom	Dg	LdDat	MemMg	Std
6	Nom	Tst	ExeP	MemMg	Std
7	Init	Sci	PrpHk	MemMg	Max
8	Init	Hk	TxSci	No	Std
9	Init	Dmp	VOpM	No	Std
10	Init	Load	TxHk	No	Std
11	Init	Dg	ExeP	No	Min
12	Init	Tst	LdDat	No	Max
13	Safe	Sci	TxHk	No	Min
14	Safe	Hk	VOpM	MemMg	Max
15	Safe	Dmp	TxSci	MemMg	Min
16	Safe	Load	PrpHk	No	Std
17	Safe	Dg	TxSci	MemMg	Max
18	Safe	Tst	TxSci	No	Min
19	Diag	Sci	VOpM	MemMg	Std
20	Diag	Hk	TxHk	No	Min
21	Diag	Dmp	PrpHk	MemMg	Max
22	Diag	Load	TxSci	No	Max
23	Diag	Dg	PrpHk	MemMg	Std
24	Diag	Tst	PrpHk	No	Min
25	Nom	Dg	TxHk	MemMg	Max
26	Init	Tst	TxHk	No	Std
27	Safe	Dg	VOpM	MemMg	Min
28	Diag	Tst	VOpM	No	Max
29	Safe	Sci	LdDat	MemMg	Min
30	Diag	Hk	LdDat	No	Std
31	Nom	Dmp	LdDat	MemMg	Min
32	Init	Load	LdDat	No	Max
33	Safe	Sci	ExeP	MemMg	Max
34	Diag	Hk	ExeP	No	Std
35	Safe	Dmp	ExeP	MemMg	Min
36	Diag	Load	ExeP	No	Max

of the 6 services (Sci, Hk, Dmp, Load, Dg, Tst) are enabled in this mode. Thus, one might conclude that there is no need for memory management in the Initiation

Mode and indeed this is true. Once again, it would be better if this is informed. Two requirements as shown below would solve these problems:

[TS013] The PDC shall be available to communicate with OBDH only after 1 minute has elapsed since PDC's initiation process. During this period, none command eventually sent by the OBDH shall be accepted by PDC.

[TS014] In the Initiation Operation Mode, the PDC shall not accomplish any data memory management.

Runs 29, 30, 33, 34, 35, and 36 are related to the second category of incompleteness. The issue now involves the Safety (Safe) and Diagnosis (Diag) Operation Modes, and the commands (Cmd factor) to load program data into memory (LdDat), and to execute a program that has just been uploaded into the computer memory (ExeP). These commands refer to the on-the-fly mechanism to load and execute a new program into the experiment's computer memory. The specification clearly asserts that in the Safety and Diagnosis Modes this service (Load) is not available. But, there is no requirement which states what to do if these two commands were sent by the OBDH in the aforementioned modes. What shall SWPDC do? Should it respond with a kind of acknowledge response defined in the communication protocol? Or should it stay quiet so that a timeout would occur in the OBDH side of the communication? Once again, a developer would have to make a decision about this. A solution to this problem is:

[TS015] In the Safety(Diagnosis) Operation Mode, the SWPDC shall not allow new programs to be uploaded into the computer memory. On receiving LOAD_DATA or EXECUTE_PROGRAM commands in this mode, the SWPDC shall not reply any of these requests.

The detected incompletenesses show the potential of SEMAFALA when addressing this issue. Naturally, the user may derive many others sets of factors and levels in order to obtain different *runs*, and thus SEMAFALA can analyze these new situations. Note that the user is important not only to define the sets of factors and levels but also to analyze *false positive* incompletenesses. However, SEMAFALA will automate significantly the incompleteness detection process, and it tends to be very scalable in the context of handling large NL requirements specifications.

5 CONCLUSIONS

This PhD proposal presented a methodology, SEMAFALA, aiming to tackle two problems. The first problem concerns with the automated translation of NL requirements into behavioral models addressing system and acceptance test case generation. The second issue is how to addresss automatically incompleteness and inconsistency in NL software requirements specifications. A tool will be developed to support the methodology. NL specifications were chosen because they are still the state-of-practice in most application domains.

Four algorithms related to the *CSAO Generator* and the *CSAO-to-Statecharts Translator* components of the SEMAFALA tool were presented. These algorithms make it possible the automated translation from NL requirements into behavioral models. On the automated analysis of defects, two types of inconsistencies are envisaged to be addressed: logical contradiction and non-determinism inconsistency. Such analysis will be accomplished within the Statecharts models automatically generated. On the other hand, the approach of incompleteness involves the use of combinatorial designs and a posterior evaluation by the user.

The SEMAFALA methodology has a basic and an alternate workflows. In the basic workflow, the sequence of actions performed is: models generation, inconsistency, and incompleteness analysis. In the alternate, the sequence is: incompleteness analysis, models generation, and inconsistency analysis. The alternate option may seem interesting for non expert professionals in a certain application domain because it provides directions, by means of *runs* of MCAs, to define the scenarios for test case generation.

The preliminary results by the application of SEMAFALA are promising even though the tool has not yet been developed. The Statecharts model generated based on a few NL requirements from the SWPDC case study resembles the model that would be created by an expert. Moreover, it was demonstrated how both types of the proposed inconsistencies can de detected. Concerning incompleteness, 14 detections were detected divided into two main categories.

Despite the proposal for automation, the user/Requirements Engineer/test designer has important roles within the SEMAFALA methodology. He/she needs to refine NL requirements, to define the domain Dictionary, to eventually change the models generated by the trio *Link Grammar Parser*, *CSAO Generator* and *CSAO-to-Statecharts Translator*, to evaluate the detected inconsistencies and incompleteness reported by the tool. However, all these tasks do not demand from the user "complex" mathematical knowledge. Hence, two characteristics are behind the philosophy of the SEMAFALA methodology/tool:

a) Easiness of use but supported by formal method. Users do not need to learn or know any formal method (language, model, logic) to use the proposed tool. However, the NL requirements are translated into a formal language (Statecharts) to support MBT and reasoning of inconsistency. Approaches like that may encourage a widespread implicit adoption of formal methods in organizations;

b) Scalability. Provided that the mechanisms for detecting inconsistency and incompleteness are not extremely complex if compared to other approaches in the literature, but they seem to be effective, the belief is that this tool may be able to handle complex NL requirements specifications.

5.1 Future Work and Schedule

The future directions of this PhD work can be divided into 3 main categories.

Possible improvements

Some possible improvements are:

a) it is not mentioned how parallel and hierarchical Statecharts models can be derived from NL requirements. One possibility for parallelism is to create automatically new parallel states whenever a time unit (millisecond, second, ...) is found in an NL requirement. For hierarchy, an analysis of the model automatically created may be the option;

b) the control features of the CSAO 4-tuples shall be better elaborated. More situations shall be addressed, for instance, how to translate requirements characterized by logical implication. Besides, how to identify self-loop transitions within the NL requirements;

c) it is likely that the four algorithms proposed may need to be changed when their implementation;

d) it is interesting to investigate the possibility of using model checking to detect inconsistency and incompleteness in NL requirements as an alternative or in order to supplement the ideas proposed in this work.

Development of the SEMAFALA tool

The development of the tool will demand a significant computational effort. At first, the idea is to develop it according to the Object-Oriented Programming paradigm using the Java language. Some tasks regarding this point:

a) development of all GUIs;

b) adaptation of the *Link Grammar Parser* to compose the tool. *Link Grammar* is developed in C language;

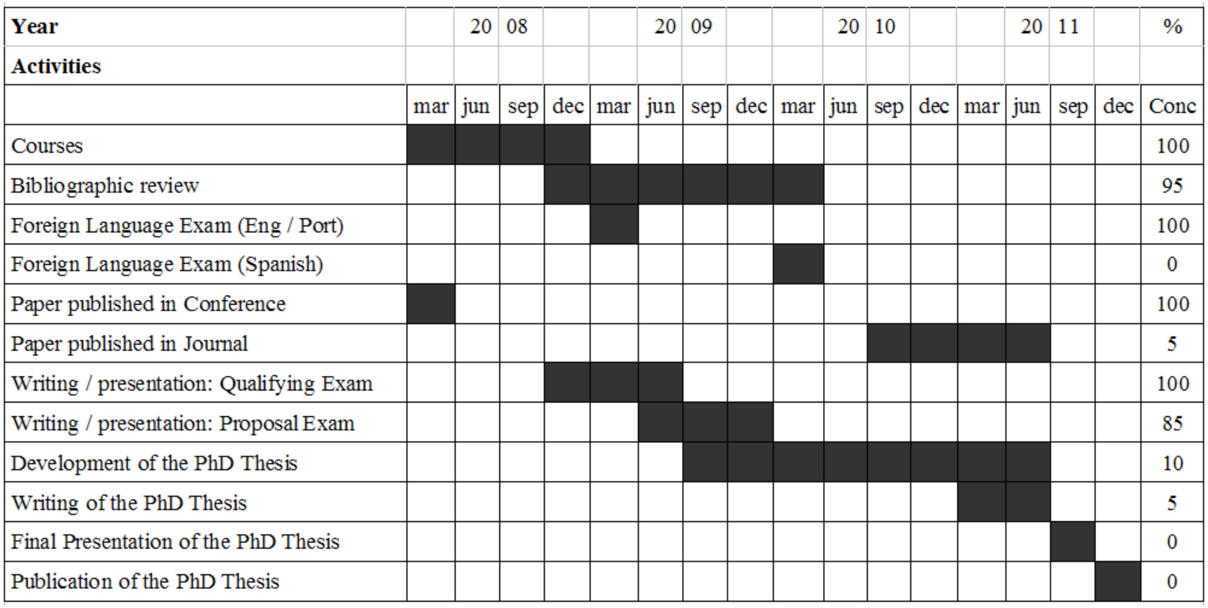
c) implementation of the components *CSAO Generator*, *CSAO-to-Statecharts Translator*, and *Defects Analyzer*. In the case of the incompleteness part of the *Defects Analyzer*, the IPO proposed by Mathur (MATHUR, 2008) may be implemented, or the TConfig tool (UNIVERSITY OF OTTAWA, 2008) may be adapted to compose the environment;

d) adaptation of the *Graph Visualization Software* (Graphviz) (GRAPHVIZ.ORG, 2009) in order to show the Statecharts models generated.

Case studies

The NL requirements specifications of the SWPDC case study will be evaluated in their entirety. Besides, 3 more requirements specifications of real software products under development or already developed at the *Divisão de Desenvolvimento de Sistemas de Solo* (DSS - Ground Systems Development Division) at INPE will be considered for the evaluation of the methodology/tool.

The schedule related to this PhD Thesis is shown in Figure 5.1. Note that each column corresponds to 3 months. For instance, *mar/2008* implies March, April, and May of 2008. The column *Conc* indicates the estimated percentage already developed of each activity.



Year		20	08			20	09			20	10			20	11		%
Activities																	
	mar	jun	sep	dec	mar	jun	sep	dec	mar	jun	sep	dec	mar	jun	sep	dec	Conc
Courses	■	■	■	■													100
Bibliographic review				■	■	■	■	■	■								95
Foreign Language Exam (Eng / Port)					■												100
Foreign Language Exam (Spanish)									■								0
Paper published in Conference	■																100
Paper published in Journal											■	■	■	■			5
Writing / presentation: Qualifying Exam				■	■	■											100
Writing / presentation: Proposal Exam						■	■	■									85
Development of the PhD Thesis							■	■	■	■	■	■	■	■			10
Writing of the PhD Thesis													■	■			5
Final Presentation of the PhD Thesis															■		0
Publication of the PhD Thesis																■	0

Figure 5.1 - Schedule of the main acitivities regarding this PhD Thesis.

REFERENCES

ABDURAZIK, A.; OFFUTT, J. Using UML collaboration diagrams for static checking and test generation. In: **UML 2000 - the Unified Modeling Language**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2000. v. 1939, p. 383–395. Lecture notes in computer science. 30

AMBRIOLA, V.; GERVASI, V. Processing natural language requirements. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), 12., 1997, Incline Village, NV, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1997. p. 36–45. 18, 49

_____. On the systematic analysis of natural language requirements with CIRCE. **Automated Software Engineering**, v. 13, n. 1, p. 107–167, 2006. 18, 20, 49

AMMONS, G.; BODIK, R.; LARUS, J. R. Mining specifications. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 29., 2002, Portland, OR, USA. **Proceedings...** New York, NY, USA: ACM, 2002. p. 4–16. 14

ANTONIOL, G.; BRIAND, L. C.; DI PENTA, M.; LABICHE, Y. A case study using the round-trip strategy for state-based class testing. In: IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), 13., 2002, Annapolis, MD, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2002. p. 269–279. 33

BALCER, M.; HASLING, W.; OSTRAND, T. Automatic generation of test scripts from formal test specifications. **ACM SIGSOFT Software Engeneering Notes**, v. 14, n. 8, p. 210–218, 1989. 33, 34

BALZER, R. Tolerating inconsistency. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 13., 1991, Austin, TX, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1991. p. 158–165. 15, 20, 53

BERRY, D. M.; KAMSTIES, E.; KRIEGER, M. M. **From contract drafting to software specification: linguistic sources of ambiguity**. Waterloo, Ontario, Canada: University of Waterloo, 2003. 80 p. Available from:

<http://se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf>. Access in: Apr 15, 2009. 18

BINDER, R. V. **Testing object-oriented systems**: models, patterns, and tools. USA: Addison-Wesley Professional, 1999. 1248 p. 28, 33

BRESCIANI, P.; PERINI, A.; GIORGINI, P.; GIUNCHIGLIA, F.; MYLOPOULOS, J. Tropos: an agent-oriented software development methodology. **Autonomous Agents and Multi-Agent Systems**, v. 8, n. 3, p. 203–236, 2004. 15

BRIAND, L. C.; LABICHE, Y. A UML-based approach to system testing. **Journal of Software and Systems Modeling**, v. 1, n. 1, p. 10–42, 2002. 31

BRIAND, L. C.; LABICHE, Y.; WANG, Y. Using simulation to empirically investigate test coverage criteria based on Statechart. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 26., 2004, Edinburgh, Scotland, UK. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2004. p. 86–95. 34

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, v. 35, n. 8, p. 667–691, 1986. 26

BUCCHIARONE, A.; GNESI, S.; LAMI, G.; TRENTANNI, G.; FANTECHI, A. QuARS Express - a tool demonstration. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), 23., 2008, L'Aquila, Italy. **Proceedings...** Los Alamitos, CA, USA: IEEE, 2008. p. 473–474. 18

CALLAGHAN, P. **An evaluation of LOLITA and related natural language processing systems**. 212 p. Thesis (PhD in Computer Science) — University of Durham, Durham, UK, 1998. 20

CHAKI, S.; CLARKE, E. M.; GROCE, A.; JHA, S.; VEITH, H. Modular verification of software components in C. **IEEE Transactions on Software Engineering**, v. 30, n. 6, p. 388–402, 2004. 26

CHOW, T. S. Testing software design modeled by finite-state machines. **IEEE Transactions on Software Engineering**, SE-4, n. 3, p. 178–187, 1978. 32, 34

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In: **25 years of model checking**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2008. v. 5000, p. 196–215. Lecture notes in computer science. 26

CLARKE, E. M.; LERDA, F. Model checking: software and beyond. **Journal of Universal Computer Science**, v. 13, n. 5, p. 639–649, 2007. 20, 26

DICK, J.; FAIVRE, A. Automating the generation and sequencing of test cases from model-based specifications. In: **FME'93: industrial-strength formal methods**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1993. v. 670, p. 268–284. Lecture notes in computer science. 33

DOLDI, L. **Validation of communications systems with SDL**: the art of SDL simulation and reachability analysis. Chichester, West Sussex, UK: John Wiley & Sons, 2003. 310 p. 31

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Patterns in property specifications for finite-state verification. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 21., 1999, Los Angeles, CA, USA. **Proceedings...** New York, NY, USA: ACM, 1999. p. 411–420. 24

EL-FAR, I. K.; WHITTAKER, J. A. Model-based software testing. In: MARCINIAK, J. J. (Ed.). **Encyclopedia of software engineering**. USA: Wiley, 2001. 30

ERNST, M. D.; COCKRELL, J.; GRISWOLD, W. G.; NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. **IEEE Transactions on Software Engineering**, v. 27, n. 2, p. 99–123, 2001. 14

FABBRINI, F.; FUSANI, M.; GNESI, S.; LAMI, G. The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool. In: ANNUAL NASA GODDARD SOFTWARE ENGINEERING WORKSHOP, 26., 2001, Greenbelt, MD, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE, 2001. p. 97–105. 18

FANTECHI, A.; SPINICCI, E. A content analysis technique for inconsistency detection in software requirements documents. In: WORKSHOP EM ENGENHARIA DE REQUISITOS (WER), 8., 2005, Porto, Portugal. **Anais...** [S.l.], 2005. p. 245–256. 21, 39, 41

FERGUSON, B.; LAMI, G. **Automated natural language analysis of requirements**. Carnegie Mellon University, 2005. 39 slides. Available from: <http://www.incose.org/delvalley/data/INCOSE-preview-QuARS_21June05.ppt>. Access in: Apr 20, 2009. 15, 18

FROHLICH, P.; LINK, J. Automated test case generation from dynamic models. In: **ECOOP 2000: object-oriented programming**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2000. v. 1850, p. 472–491. Lecture notes in computer science. 34

GERVASI, V.; ZOWGHI, D. Reasoning about inconsistencies in natural language requirements. **ACM Transactions on Software Engineering and Methodology**, v. 14, n. 3, p. 277–330, 2005. 16, 20, 21, 48, 49

GNESI, S.; LAMI, G.; TRENTANNI, G. An automatic tool for the analysis of natural language requirements. **International Journal of Computer Systems Science and Engineering**, v. 20, n. 1, p. 1–13, 2005. 18

GODBOLE, N. S. **Software quality assurance**: principles and practice. Oxford, UK: Alpha Science International, 2006. 419 p. 13

GRAPHVIZ.ORG. 2009. Available from: <http://www.graphviz.org/>. Access in: Oct 20, 2009. 65

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, v. 8, p. 231–274, 1987. 31

HAREL, D.; PNUELI, A.; SCHMIDT, J. P.; SHERMAN, R. On the formal semantics of Statecharts (extended abstract). In: IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE (LICS), 2., 1987, Ithaca, NY, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1987. p. 54–64. 31

HARTMANN, J.; IMOBERDORF, C.; MEISINGER, M. UML-based integration testing. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS (ISSTA), 2000, Portland, OR, USA. **Proceedings...** New York, NY, USA: ACM, 2000. p. 60–70. 34

HARTMANN, J.; VIEIRA, M.; FOSTER, H.; RUDER, A. A UML-based approach to system testing. **Journal of Innovations in System Software Engineering**, v. 1, p. 12–24, 2005. 31

HIERONS, R. M. Testing from a Z specification. **The Journal of Software Testing, Verification and Reliability**, v. 7, n. 1, p. 19–33, 1997. 33

HOLZMANN, G. J. **The SPIN model checker**: primer and reference manual. USA: Addison-Wesley Professional, 2003. 608 p. 24, 26

HONG, H. S.; KIM, Y. G.; CHA, S. D.; BAE, D. H.; URAL, H. A test sequence selection method for Statecharts. **Software Testing, Verification and Reliability**, v. 10, n. 4, p. 203–227, 2000. 33

HOPCROFT, J. E.; ULLMAN, J. D. **Introduction to automata theory, languages, and computation**. Reading, MA, USA: Addison Wesley, 1979. 418 p. 32

HOWDEN, W. E. Reliability of the path analysis testing strategy. **IEEE Transactions on Software Engineering**, SE-2, n. 3, p. 208–215, 1976. 34

HUNTER, A.; NUSEIBEH, B. Managing inconsistent specifications: reasoning, analysis, and action. **ACM Transactions on Software Engineering and Methodology**, v. 7, n. 4, p. 335–367, 1998. 21

JURAFSKY, D.; MARTIN, J. H. **Speech and language processing**: an introduction to natural language processing, computational linguistics and speech recognition. Englewood Cliffs, NJ, USA: Prentice-Hall, 2000. 950 p. 17

KIM, H. Y.; SHELDON, F. T. Testing software requirements with Z and Statecharts applied to an embedded control system. **Software Quality Journal**, v. 12, n. 3, p. 231–264, 2004. 21, 22, 49

KONRAD, S.; CHENG, B. H. C. Real-time specification patterns. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 27., 2005, St. Louis, MO, USA. **Proceedings...** New York, NY, USA: ACM, 2005. p. 372–381. 24

______. Automated analysis of natural language properties for UML models. In: **Satellite events at the MoDELS 2005 conference**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2006. v. 3844, p. 48–57. Lecture notes in computer science. 24

LAMI, G.; TRENTANNI, G. An automatic tool for improving the quality of software requirements. **ERCIM News**, n. 58, p. 18–19, 2004. 18

LAPRIE, J.-C.; KANOUN, K. Software reliability and system reliability. In: LYU, M. R. (Ed.). **Handbook of software reliability engineering**. New York, NY, USA: McGraw-Hill, 1996. chapter 2, p. 27–69. 27

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines: a survey. **Proceedings of the IEEE**, v. 84, n. 8, p. 1090–1123, 1996. 31

LORENZOLI, D.; MARIANI, L.; PEZZÈ, M. Automatic generation of software behavioral models. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 30., 2008, Leipzig, Germany. **Proceedings...** New York, NY, USA: ACM, 2008. p. 501–510. 14

MASIERO, P. C.; MALDONADO, J. C.; BOAVENTURA, I. G. A reachability tree for Statecharts and analysis of some properties. **Information and Software Technology**, v. 36, n. 10, p. 615–624, 1994. 33, 36

MATHUR, A. P. **Foundations of software testing**. Delhi, India: Dorling Kindersley (India), Pearson Education in South Asia, 2008. 689 p. 28, 29, 30, 37, 49, 65

McMILLAN, K. L. **Symbolic model checking**. New York, NY, USA: Springer-Verlag, 1993. 216 p. 26

MICH, L. NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. **Natural Language Engineering**, v. 2, n. 2, p. 161–187, 1996. 19

MICH, L.; FRANCH, M.; INVERARDI, P. Market research for requirements analysis using linguistic tools. **Requirements Engineering Journal**, v. 9, n. 1, p. 40–56, 2004. 15, 18, 19

MICH, L.; MYLOPOULOS, J.; ZENI, N. **Improving the quality of conceptual models with NLP tools: an experiment**. Trento, Italy: University of Trento, 2002. 12 p. (DIT-02-0047). 20

MORGAN, R.; GARIGLIANO, R.; CALLAGHAN, P.; PORIA, S.; SMITH, M.; URBANOWICZ, A.; COLLINGHAM, R.; COSTANTINO, M.; COOPER, C.; LOLITA Group. University of Durham: description of the LOLITA system as used in MUC-6. In: MESSAGE UNDERSTANDING CONFERENCE (MUC-6), 6., 1995, Columbia, MD, USA. **Proceedings...** [S.l.], 1995. p. 71–85. 19, 20

MYERS, G. J. **The art of software testing**. 2. ed. Hoboken, NJ, USA: John Wiley & Sons, 2004. 234 p. 27

NUSEIBEH, B.; EASTERBROOK, S. Requirements engineering: a roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING, 2000, Limerick, Ireland. **Proceedings...** New York, NY, USA: ACM, 2000. p. 35–46. 14, 15, 16

OFFUTT, J.; ABDURAZIK, A. Generating tests from UML specifications. In: **UML'99: the Unified Modeling Language**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1999. v. 1723, p. 416–429. Lecture notes in computer science. 34

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating functional tests. **Communications of the ACM**, v. 31, n. 6, p. 676–686, 1988. 29, 31, 33, 34

PARADKAR, A. Towards model-based generation of self-priming and self-checking conformance tests for interactive systems. In: ACM SYMPOSIUM ON APPLIED COMPUTING (SAC), 18., 2003, Melbourne, FL, USA. **Proceedings...** New York, NY, USA: ACM, 2003. p. 1110–1117. 33

PETRENKO, A.; YEVTUSHENKO, N. Testing from partial deterministic FSM specifications. **IEEE Transactions on Computers**, v. 54, n. 9, p. 1154–1165, 2005. 31, 32

PIMONT, S.; RAULT, J. C. A software reliability assessment based on a structural and behavioral analysis of programs. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2., 1976, San Francisco, CA, USA. **Proceedings...** New York, NY, USA: ACM, 1976. p. 486–491. 32

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. 5. ed. New York, NY, USA: McGraw-Hill, 2001. 860 p. 13, 25

QUEILLE, J. P.; SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In: **International symposium on programming**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1982. v. 137, p. 337–351. Lecture notes in computer science. 26

RUSSELL, S. J.; NORVIG, P. **Artificial intelligence**: a modern approach. Englewood Cliffs, NJ, USA: Prentice-Hall, 1995. 932 p. 17

SANTIAGO, V.; AMARAL, A. S. M.; VIJAYKUMAR, N. L.; MATTIELLO-FRANCISCO, M. F.; MARTINS, E.; LOPES, O. C. A practical approach for automated test case generation using Statecharts. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE & APPLICATIONS CONFERENCE (COMPSAC) - INTERNATIONAL WORKSHOP ON TESTING AND QUALITY ASSURANCE FOR COMPONENT-BASED SYSTEMS (TQACBS), 30., 2006, Chicago, IL, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2006. p. 183–188. 33, 35

SANTIAGO, V.; MATTIELLO-FRANCISCO, F.; COSTA, R.; SILVA, W. P.; AMBROSIO, A. M. QSEE project: an experience in outsourcing software development for space applications. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING & KNOWLEDGE ENGINEERING (SEKE), 19., 2007, Boston, MA, USA. **Proceedings...** Skokie, IL, USA: Knowledge Systems Institute Graduate School, 2007. p. 51–56. 28, 39, 53

SANTIAGO, V.; SILVA, W. P.; VIJAYKUMAR, N. L. Shortening test case execution time for embedded software. In: IEEE INTERNATIONAL CONFERENCE ON SECURE SYSTEM INTEGRATION AND RELIABILITY IMPROVEMENT (SSIRI), 2., 2008, Yokohama, Japan. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p. 81–88. 1 CD-ROM. 13, 28

SANTIAGO, V.; VIJAYKUMAR, N. L.; GUIMARAES, D.; AMARAL, A. S.; FERREIRA, E. An environment for automated test case generation from Statechart-based and Finite State Machine-based behavioral models. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST) - WORKSHOP ON ADVANCES IN MODEL BASED TESTING (A-MOST), 1., 2008, Lillehammer, Norway. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p. 63–72. 1 CD-ROM. 31, 35, 36, 39

SARMA, M.; MALL, R. Automatic generation of test specifications for coverage of system state transitions. **Information and Software Technology**, v. 51, n. 2, p. 418–432, 2009. 34

SIDHU, D. P.; LEUNG, T. K. Formal methods for protocol testing: a detailed study. **IEEE Transactions on Software Engineering**, v. 15, n. 4, p. 413–426, 1989. 32, 34

SILVA, W. P. **QSEE-TAS/SPAC: uma ferramenta para execução automatizada de testes de software e processamento de dados científicos para aplicações espaciais**. 99 p. Dissertation (Master in Applied Computing) — Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil, 2008. 28, 39

SILVA, W. P.; SANTIAGO, V.; MATTIELLO-FRANCISCO, M. F.; PASSOS, D. QSEE-TAS: uma ferramenta para execução e relato automatizados de testes de software para aplicações espaciais. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE (SBES) - SESSÃO DE FERRAMENTAS, 20., 2006, Florianópolis, SC, Brazil. **Anais...** Porto Alegre, RS, Brazil: Sociedade Brasileira de Computação, 2006. p. 43–48. 28

SILVA, W. P.; SANTIAGO, V.; VIJAYKUMAR, N. L.; MATTIELLO-FRANCISCO, F. SPAC: ferramenta para processamento e análise de dados científicos no processo de validação de software em aplicações espaciais. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE (SBES) - SESSÃO DE FERRAMENTAS, 21., 2007, João Pessoa, PB, Brazil. **Anais...** Porto Alegre, RS, Brazil: Sociedade Brasileira de Computação, 2007. p. 70–76. 28

SINGH, H.; CONRAD, M.; SADEGHIPOUR, S. Test case design based on Z and the classification-tree method. In: INTERNATIONAL CONFERENCE ON FORMAL ENGINEERING METHODS (ICFEM), 1., 1997, Hiroshima, Japan. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1997. p. 81–90. 33

SINHA, A.; PARADKAR, A.; WILLIAMS, C. On generating EFSM models from use cases. In: INTERNATIONAL WORKSHOP ON SCENARIOS AND STATE MACHINES (SCESM), 6., 2007, Minneapolis, MN, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p. 1–8. 15, 32

SLEATOR, D. D.; TEMPERLEY, D. Parsing English with a link grammar. In: INTERNATIONAL WORKSHOP ON PARSING TECHNOLOGIES, 3., 1993, Tilburg, The Netherlands. **Proceedings...** [S.l.], 1993. p. 277–292. 21, 39

SNEED, H. M. Testing against natural language requirements. In: INTERNATIONAL CONFERENCE ON QUALITY SOFTWARE (QSIC), 7., 2007, Portland, OR, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p. 380–387. 22

SOUZA, S. R. S. **Validação de especificações de sistemas reativos: definição e análise de critérios de teste**. 264 p. Thesis (PhD in Applied Physics) — Universidade de São Paulo, São Carlos, SP, Brazil, 2000. 33

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE). **IEEE Std 610.12-1990**: IEEE standard glossary of software engineering terminology. New York, NY, USA, 1990. 83 p. 14, 25, 27

_____. **IEEE Std 829-1998**: IEEE standard for software test documentation. New York, NY, USA, 1998. 52 p. 28

THE OBJECT MANAGEMENT GROUP (OMG). **OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2**. Needham, MA, USA, 2007. 722 p. 15, 17, 30

UNIVERSITY OF OTTAWA. **Alan Williams' Page**. 2008. Available from: <`http://www.site.uottawa.ca/~awilliam/`>. Access in: Oct 15, 2009. 49, 52, 60, 65

VIJAYKUMAR, N. L.; CARVALHO, S. V.; FRANCÊS, C. R. L.; ABDURAHIMAN, V.; AMARAL, A. S. M. Performance evaluation from Statecharts representation of complex systems: Markov approach. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (CSBC) - WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO, 26., 2006, Campo Grande, MS, Brazil. **Proceedings...** Porto Alegre, RS, Brazil: Sociedade Brasileira de Computação, 2006. p. 183–202. 33, 35

WEYUKER, E. J. On testing non-testable programs. **The Computer Journal**, v. 25, n. 4, p. 465–470, 1982. 28