SOLIMVA: A METHODOLOGY FOR GENERATING MODEL-BASED TEST CASES FROM NATURAL LANGUAGE REQUIREMENTS AND DETECTING INCOMPLETENESS IN SOFTWARE SPECIFICATIONS

Valdivino Alexandre de Santiago Júnior

Doctorate Thesis at Post Graduation Course in Applied Computing, advised by Dr. Nandamudi Lankalapalli Vijaykumar, approved in December 12, 2011.

INPE
São José dos Campos
2011

PUBLISHED BY:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

sid.inpe.br/mtc-m19/2011/11.07.23.30-TDI

SOLIMVA: A METHODOLOGY FOR GENERATING MODEL-BASED TEST CASES FROM NATURAL LANGUAGE REQUIREMENTS AND DETECTING INCOMPLETENESS IN SOFTWARE SPECIFICATIONS

Valdivino Alexandre de Santiago Júnior

Doctorate Thesis at Post Graduation Course in Applied Computing, advised by Dr. Nandamudi Lankalapalli Vijaykumar, approved in December 12, 2011.

URL of the original document:
<http://urlib.net/8JMKD3MGP7W/3AP764B>

INPE
São José dos Campos
2011

Cataloging in Publication Data

Sa59s

Santiago Júnior, Valdivino Alexandre de.

SOLIMVA: a methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications / Valdivino Alexandre de Santiago Júnior. – São José dos Campos : INPE, 2011.

xxvi + 238 p. ; (sid.inpe.br/mtc-m19/2011/11.07.23.30-TDI)

Doctorate Thesis (Doctorate Thesis in Applied Computing) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2011.

Adviser : Dr. Nandamudi Lankalapalli Vijaykumar.

1. Model-based testing. 2. Natural language requirements. 3. Combinatorial designs. 4. Part of speech tagging. 5. Word sense disambiguation. 6. Statecharts. 7. Incompleteness. 8. Model checking. 9. Specification patterns I.Título.

CDU 004.415.5

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

Computação Aplicada

Dr. Edson Luiz França Senne

[illegible]

Presidente / UNESP/GUARA / Guaratinguetá - SP

Dr. Nandamudi Lankalapalli Vijaykumar

N. L. Vijaykumar

Orientador(a) / INPE / SJCampos - SP

Dr. Edgar Toshiro Yano

[illegible]

Convidado(a) / IEC/ITA / São José dos Campos - SP

Dra. Ana Cristina Vieira de Melo

[illegible]

Convidado(a) / IME/USP / São Paulo - SP

Dra. Miriam Celia Bergue Alves

[illegible]

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

Este trabalho foi aprovado por:

( ) maioria simples

[✓ unanimidade]

Aluno (a): Valdivino Alexandre de Santiago Júnior

São José dos Campos, 12 de dezembro de 2011

“Have faith and trust in God especially in the hardest times of your life.”

J. S. Nobre
in *“Comece o Dia Feliz (Reflexões)”*, 1994

To my wife Maria, my daughters Sofia and Lívia, my father Valdivino (in memoriam), my mother Nair, and my sister Stela

ACKNOWLEDGEMENTS

First, I would like to thank God. I could not have been successful in developing this PhD thesis if I did not have faith in God and in Jesus Christ, and if I did not believe in Their love for me.

I would like to thank and dedicate this work to my wife Maria, and my daughters Sofia and Lívia. Their patience, love, understanding during the time I devoted to this PhD thesis were crucial so I could have the strength to go on. God, the Virgin Mary, my wife Maria, and my daughters Sofia and Lívia are the reasons of my life.

I would like to thank and dedicate this work to my father Valdivino (in memoriam), my mother Nair, and my sister Stela, for all the love and support especially at the beginning of my career. I also thank my grandmother Stela (in memoriam), my grandmother Ana (in memoriam), my uncle Hemetério, my aunt Inez, my uncle Edvaldo, my aunt Maria José, my cousins Stela Maria, Regina (in memoriam), Célia, Waleska, Angelita, my sister-in-law Doralice and her family, and other relatives who somehow contributed to my growth as a human being.

I would like to thank my advisor Dr. Vijaykumar, for his friendship and advice which were important so that I could develop this PhD thesis.

I would like to thank Dr. Demísio (in memoriam), a friend who positively influenced me personally and professionally. I consider Dr. Demísio (in memoriam) as a contributor of this work.

I would like to thank the Instituto Nacional de Pesquisas Espaciais (INPE - National Institute for Space Research) for allowing me to dedicate part of my working day to develop this PhD thesis.

Finally, I would like to thank other professors of the Curso de Pós-Graduação em Computação Aplicada (CAP - Post Graduation Course in Applied Computing) which helped to improve my qualifications as a researcher.

ABSTRACT

In greater or lesser extent, Natural Language (NL) is still widely used to develop software requirements specifications or other artifacts created for documenting requirements. However, NL deliverables suffer from ambiguity, inconsistency, and incompleteness. This PhD thesis presents a methodology, **SOLIMVA**, which achieves two objectives. The primary objective is to generate model-based system and acceptance test cases considering NL requirements deliverables. For this purpose, a tool, also called SOLIMVA, was designed and implemented and such a tool makes it possible to automatically translate NL requirements into Statechart models. Once the Statecharts are created, another tool, GTSC, is used to generate Abstract Test Cases which are later translated into Executable Test Cases. Among other features, a Word Sense Disambiguation method which helps in the translation process was implemented in the SOLIMVA tool, and combinatorial designs are used to identify scenarios for model-based system and acceptance testing. The SOLIMVA methodology was applied to a main case study, a space application software product related to the Space Segment, and the methodology was compared with a previous manual approach developed by an expert under two aspects: coverage of test objectives and characteristics of Executable Test Cases. In both aspects, the SOLIMVA methodology presented benefits such as a better strategy with test objectives clearly separated according to the directives of combinatorial designs, and the generation of Executable Test Cases which predicted behaviors that did not exist in the expert's approach. In addition, guidelines to apply the SOLIMVA methodology to a second case study of the space domain related to the Ground Segment are also presented. The key advantages from applying the SOLIMVA methodology in the context of a Verification and Validation process are the ease of use and the support of a formal method, making it potentially suitable for use in complex software projects. The secondary objective is the detection of incompleteness in software specifications. The SOLIMVA methodology was then extended to achieve this secondary objective. Model Checking combined with k-permutations of n values of variables and specification patterns were used to address this goal. The SOLIMVA methodology has proved its efficiency by the detection of 21 incompleteness defects when applied to the same main case study mentioned earlier.

SOLIMVA: UMA METODOLOGIA PARA GERAÇÃO DE CASOS DE TESTES BASEADOS EM MODELOS A PARTIR DE REQUISITOS EM LINGUAGEM NATURAL E DETECÇÃO DE NÃO COMPLETUDE EM ESPECIFICAÇÕES DE SOFTWARE

RESUMO

Em maior ou menor extensão, a Linguagem Natural (LN) é ainda amplamente usada para elaborar especificações de requisitos de software ou outros artefatos criados para a documentação de requisitos. Entretanto, fornecimentos elaborados em LN apresentam ambiguidade, inconsistência e não completude. Esta tese de doutorado apresenta uma metodologia, **SOLIMVA**, a qual alcança dois objetivos. O objetivo primário é a geração de casos de teste de sistema e aceitação baseados em modelos a partir de artefatos de requisitos elaborados em LN. Para esse propósito, uma ferramenta, também denominada SOLIMVA, foi projetada e implementada, e tal ferramenta traduz automaticamente requisitos elaborados em LN em modelos Statecharts. Uma vez gerados os Statecharts, outra ferramenta, GTSC, é usada para gerar Casos de Teste Abstratos os quais depois são transformados em Casos de Teste Executáveis. Entre outras características, um método para Desambiguidade de Sentido de Palavras, o qual ajuda no processo de tradução, foi implementado na ferramenta SOLIMVA, e *designs* combinatoriais são usados para identificar cenários para testes de sistema e aceitação baseados em modelos. A metodologia SOLIMVA foi aplicada a um estudo de caso principal, um produto de software da área espacial relacionado ao Segmento Espacial, e a metodologia foi comparada com uma abordagem manual desenvolvida anteriormente por um especialista sob dois aspectos: cobertura de objetivos de teste e características dos Casos de Teste Executáveis. Em ambos os aspectos, a metodologia SOLIMVA mostrou benefícios tais como uma melhor estratégia com os objetivos de teste claramente separados de acordo com as diretivas dos *designs* combinatoriais, e a geração de Casos de Teste Executáveis que previram comportamentos que não existiam na abordagem do especialista. Além disso, diretivas para aplicar a metodologia SOLIMVA a um segundo estudo de caso do domínio espacial, relacionado ao Segmento Solo, são também apresentadas. As principais vantagens em aplicar a metodologia SOLIMVA no contexto de um processo de Verificação e Validação são a facilidade de uso e o suporte de um método formal, fazendo com que a metodologia seja potencialmente adequada para ser usada em projetos de software complexos. O objetivo secundário é a detecção de não completude em especificações de software. A SOLIMVA foi então estendida para alcançar esse objetivo secundário. *Model Checking* combinado com arranjos simples de valores de variáveis e padrões de especificação foram usados para alcançar essa meta. A metodologia SOLIMVA demonstrou a sua eficiência pelo fato de detectar 21 defeitos de não completude, ao ser aplicada ao mesmo estudo de caso principal mencionado anteriormente.

LIST OF FIGURES

Pág.

LIST OF TABLES

LIST OF ABBREVIATIONS

BSAO	–	Behavior-Subject-Action-Object
CTL	–	Computation Tree Logic
DNF	–	Disjunctive Normal Form
ECSS	–	European Cooperation for Space Standardization
EFSM	–	Extended Finite State Machine
EPP	–	Event Pre-Processor
FSM	–	Finite State Machine
GTSC	–	*Geração Automática de Casos de Teste Baseada em Statecharts*
IEEE	–	The Institute of Electrical and Electronics Engineers
INPE	–	*Instituto Nacional de Pesquisas Espaciais*
IUT	–	Implementation Under Test
LTL	–	Linear Temporal Logic
NASA	–	National Aeronautics and Space Administration
NL	–	Natural Language
NLP	–	Natural Language Processing
OBDH	–	On-Board Data Handling
PBR	–	Perspective-Based Reading
PDC	–	Payload Data Handling Computer
PhD	–	*Philosophiae Doctor* (Doctor of Philosophy)
POS	–	Part Of Speech
PUS	–	Packet Utilization Standard
QSEE	–	*Qualidade do Software Embarcado em Aplicações Espaciais*
RE	–	Software Systems Requirements Engineering
SAO	–	Subject-Action-Object
SATCS	–	Satellite Control System
SCCF	–	Statechart Coverage Criteria Family
SMV	–	Symbolic Model Verifier
SWPDC	–	Software for the Payload Data Handling Computer
UML	–	Unified Modeling Language
WSD	–	Word Sense Disambiguation

CONTENTS

1 INTRODUCTION

The various reports of serious problems in systems that have occurred due to defects in software have drawn attention, for some time already, from academicians and industry professionals. These researchers and practitioners have thus insisted on the use of standards, methodologies, techniques for developing software to try to ensure that, in the end, a high-quality software is produced. The Therac-25 medical electron accelerator (1985-1987) (LEVESON; TURNER, 1993), the Ariane-5 rocket (1996) (ESA, 1997), and the Mars Climate Orbiter (1999) (NASA JET PROPULSION LABORATORY, 2011) are classic examples in which defects in the software were the main cause of failure of such systems resulting in large financial losses and, in the case of the Therac-25 device, in losses of human lives.

Quality is therefore a desirable attribute for any kind of product including software. However, quality is very difficult to define. Specialists provide different perspectives regarding this broad 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 perspective, practitioners agree that quality is a major business factor and to acquire it requires a huge effort from the organizations.

The term Software Assurance has a very wide connotation according to the National Aeronautics and Space Administration (NASA). NASA states that Software Assurance includes several disciplines, among others: Software Quality Assurance, Software Quality Control, Software Reliability, Software **Verification and Validation**, and Software **Independent Verification and Validation** (NASA, 2009). It is clear therefore that Verification and Validation (IEEE, 1990) is one of the pillars to ensure that software products have high quality, and this discipline is particularly important if critical systems are considered.

Verification and Validation encompass a wide array of activities including formal technical reviews, inspection (requirements, design, code), and all sorts of testing. Thus, testing a software product is only a facet to get quality. However, the role of testing is undoubtedly important and it has received attention from both industry and academia.

1.1 Motivation

Model-Based Testing has drawn attention in both industrial and academic areas in which several different types of software models have been used in order to guide test case generation (EL-FAR; WHITTAKER, 2001). Although more related to test case generation, Model-Based Testing also influences other activities of the testing process such as test results evaluation (EL-FAR; WHITTAKER, 2001; SANTIAGO et al., 2008b). As with many other fields of Software Engineering, Model-Based Testing has different interpretations depending on the author or group of researchers. Definitions of Model-Based Testing and related work are presented in Chapter 2.

One problem is that test process activities require a huge effort, within the scope of the software development lifecycle, to be performed and this effort is even greater if software products developed for critical systems are considered. Thus, Software Testing automation appeared as an attempt to reduce the costs of testing, increase fault (defect) detection and shorten testing cycles. Although Software Testing automation is not a silver bullet to solve all testing problems, if properly planned and implemented, it can help to achieve cost and effectiveness of test process activities during the software development lifecycle (SANTIAGO et al., 2008a).

However, it is important to emphasize that in any initiative related to Software Testing automation, human aspect is still very relevant. Even though there are lots of commercial and open source tools and frameworks available for this purpose, in almost all cases human interventions are still necessary to analyze test cases created by a tool, to evaluate coverage of test cases, to provide expected results of the test cases, to verify whether a verdict was correctly asserted by an automated oracle, to generate models in a model-based approach, and so on (SANTIAGO JÚNIOR; VIJAYKUMAR, 2012).

In order to confirm the importance of the professional (need for manual intervention) in Software Testing automation, a recent survey was published with the purpose of characterizing Model-Based Testing techniques (approaches) usually employed in software development projects described in the technical literature (DIAS NETO et al., 2007; DIAS NETO, 2009). In the first part of this systematic review (DIAS NETO et al., 2007), 202 articles were found which dealt with Model-Based Testing techniques. Considering 78 out of the 202 articles, analysis was conducted in two levels: qualitative and quantitative. One attribute of the quantitative analysis was the level of automation of the approaches. The authors concluded that all analyzed Model-Based Testing approaches have at least one non automated step: the initial modeling of software behavior. Besides, in some approaches the initial modeling is a hard task, involving translation between models, data modeling, etc. They found other types of non automated steps such as manual modeling of intermediate model or test data definition (DIAS NETO et al., 2007).

In system and acceptance testing, test cases/sequences are derived considering the entire software product. In this case, black box testing techniques (MATHUR, 2008) are usually adopted. Besides, a scenario-based approach is also recommended for system and acceptance test case generation where distinct interactions with the system are addressed.

In order to generate model-based system and acceptance test cases, a test designer usually breaks down the entire system based on functionalities it must provide (or interactions with the system in case of scenario-based approaches), and then models are derived to address each functionality. Based on such models, test cases can be obtained. However, identification of scenarios that consequently leads to test case generation is not an easy task and it is time consuming. Test designers try to identify scenarios based on the very first deliverables (artifacts) created within the software development lifecycle[1], such as software requirements specifications. Even though a development team might produce software requirements specifications using scenario-based methods, like Unified Modeling Language (UML) use case models (OMG, 2007), a test designer must not rely only on the "perspective" of the developers because, in doing so, he/she will not have the independent point of view that is crucial in test case generation.

Dynamic analysis is the process of evaluating a system or component based on its behavior during execution (IEEE, 1990). Therefore, Software Testing can be considered a type of dynamic analysis technique. There are many other dynamic analysis techniques such as: Ernst et al. (2001) used dynamic techniques for discovering invariants from execution traces to support software evolution; Ammons et al. (2002) 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 (defects); and Lorenzoli et al. (2008) 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.

[1]In this PhD thesis, software development lifecycle models refer to the more traditional models such as the Waterfall model and V-model. Moreover, the nomenclature of the phases of these models can vary depending on the author who describes the model and also the models themselves.

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 defects[2], violations of development standards, and other problems (IEEE, 1990). Although most commonly applied to code and design (TRAVASSOS et al., 1999), Inspections have also been used for identifying defects in requirements. Software Reading Techniques attempt to increase the effectiveness of Inspections by providing procedural guidelines that can be used by individual reviewers to examine a given software artifact and identify defects. They can be performed on all documents related to the software development process and also during the very earlier phases of the software lifecycle, attempting to avoid propagation of problems to other stages of development. There are families of Software Reading Techniques that focus on defect detection in requirements such as Defect-Based Reading (DBR) (PORTER et al., 1995) and Perspective-Based Reading (PBR) (BASILI et al., 1996).

1.1.1 Requirements: importance and documentation

There are several publications in the literature that show that problems in requirements are serious factors that affect the quality of software products. Some studies revealed that more than 50% of software defects are attributed to problems in requirements and more than 80% of rework effort is spent on requirements-related defects (FERGUSON; LAMI, 2005). Lamsweerde (2000) quoted that in "a survey over 8000 projects undertaken by 350 US companies revealed that one third of the projects were never completed and one half succeeded only partially, that is, with partial functionalities, major cost overruns, and significant delays". Managers identified requirements as the main cause of failures in accordance with the following problems: the lack of user involvement (13%), requirements incompleteness (12%), changing requirements (11%), unrealistic expectations (6%), and unclear objectives (5%).

Emam and Koru (2008) questioned arguments that there is still a software crisis triggered by the Standish Group by publishing its CHAOS report in 1994. This report indicated a high cancellation rate for software projects. They conducted a replicated international web survey of Information Technology (IT) departments in 2005 and 2007. Among the goals of this survey are the estimation of IT projects' actual cancellation rates and the determination of what factors have the biggest impact

[2]Although the *IEEE Standard Glossary of Software Engineering Terminology* (IEEE, 1990) uses the term "errors", this PhD thesis will prefer the term "defects" to mean any non-correctness in the artifacts of the software product. Chapter 2 has additional comments on this topic.

on cancellation. Considering IT software projects only, they asserted that talking about a software crisis is perhaps exaggerated: their measurements showed that the IT project cancellation rate ranged from 11.5% to 15.5% and, since 1994, there has been a clear trend of decreasing cancellation rates across all studies. However, changes in requirements and scope were primary reasons for project cancellation. Again, this survey demonstrates the importance of the requirements in the success or failure of software projects.

Other study was conducted to investigate the current state of Software Systems Requirements Engineering (RE) problems and practices amongst the software development companies in Malaysia (SOLEMON et al., 2009). The authors concluded that organizational and technical factors were major obstacles to obtaining appropriate software requirements by the software companies involved in the study.

By defining software quality, Pressman (2001) emphasizes, among other things, 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 developed and, therefore, good requirements specifications are a valuable starting point towards high-quality software.

Based on these arguments, it is clear that RE also contributes to software quality. RE is the process of discovering the purpose of a software system. Nuseibeh and Easterbrook (2000) define 5 core RE activities: elicitation, modeling and analysis[3], communication, agreement, and evolution. 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. Software systems are dynamic, i.e. they evolve. Stakeholders requirements also change. Thus, evolution is a natural characteristic of requirements.

Requirements may be elicited according to several techniques. Traditional techniques include the use of questionnaires and surveys, interviews, and analysis of existing

[3]There are different activities within requirements analysis such as state-mode analysis, use-case analysis, trade-off analysis, negotiation (KO et al., 2007), and quality analysis. In this work, requirements analysis refers to the latter, which focuses on improving the quality of requirements specifications by detecting and removing defects like inconsistency, incompleteness, and ambiguity.

documentation. Model-driven techniques can also be used in the elicitation process and among these techniques are scenario-based methods, such as UML use case models (OMG, 2007), and goal-oriented methods, such as Tropos (BRESCIANI et al., 2004). Models can then be used to represent a whole range of products of the RE activities. In addition, as mentioned, many modeling techniques are used as elicitation tools. It is crucial that the requirements are communicated to different stakeholders in an appropriate way for a proper understanding of such requirements. Requirements can be documented in artifacts like software requirements specifications by using several different approaches such as formal methods, semi-formal and informal (natural) languages (NUSEIBEH; EASTERBROOK, 2000).

The academic community has been advocating the use of formal methods (languages/models) for the development of hardware and software systems for quite some time (HALL, 1990; BOWEN; HINCHEY, 1995). According to their supporters, formal methods are very helpful at finding defects early within the development, they compel professionals to think very hard about the system they propose to build, they can reduce development costs and improve product quality (HALL, 1990). Mathematical rigor enables customers, developers, validators, verifiers to analyze the system at any phase of the software development lifecycle.

Woodcock et al. (2009) undertook a survey to gather information in a consistent format from 62 industrial projects known to have employed formal methods. They collected data between November/2007 and December/2008 and the projects surveyed came (in decreasing order) from Europe, Northern America, South America, Australia, and Asia. The application domain which had the largest number of projects was transport, followed by the financial sector. In general, respondents have been positive about the successful use of formal methods in their projects. For 35% of the individuals, time was reduced due to application of formal methods while 12% said that time increased. However, several noted increased time in the specification phase, which may or may not have been compensated for by decreasing time later. Cost was reported to be decreased in 37% of the projects while only 7% of the projects had increased costs. The use of formal methods is believed by professionals to have improved quality, with 92% of all cases reporting an increase in quality compared to other techniques, and no cases reporting a decrease in quality.

The numbers shown in the survey of Woodcock et al. (2009) are very favorable for the adoption of formal methods for systems development in industry. Model Checking (CLARKE; EMERSON, 2008; QUEILLE; SIFAKIS, 2008; BAIER; KATOEN, 2008) and proof technology continue to be used by leading hardware companies. In software, the use of formal methods has provided some evidence of the potential for certain applications, including code verification. However, Woodcock et al. (2009) claimed that "In spite of their successes, verification technology and formal methods have not seen widespread adoption as a routine part of systems development practice, except, arguably, in the development of critical systems in certain domains". This statement confirms what has long been said about formal methods: despite all the benefits, formal methods have not yet been a widely adopted practice in systems development, and particularly in software. One justification for this fact is that the cost of introducing formal methods seems to be extremely high for software project managers. This view is also shared by Abrial (2006) who asseverated that formal methods are difficult to integrate with the ordinary software development process adopted in industry and, consequently, managers usually avoid the inclusion of formal methods in their established processes.

It is almost inconceivable to think about an industrial project that can proceed without tools. In accordance with several studies previously presented in the literature and also with this new survey (WOODCOCK et al., 2009), the lack of commercially supported tools is an impediment to take-up of formal methods. Woodcock et al. (2009) also remark that some comments in their survey indicate that tools are still not usable, in the words of one respondent, "by mere mortals". They identified some challenges for the development of tools to support formal methods such as support for automated deduction, and common formats for the interchange of models and analysis. Besides, they point out practical issues to be addressed such as providing multi-language support, porting to a variety of platforms, and version control.

Some authors assert that UML is currently the de facto standard for modeling (object-oriented) software, and its use is increasing in the aerospace industry (ZOUGHBI et al., 2011). Mich et al. (2004) presented an online market research conducted in 1999 in which 142 software companies were involved. The principal goal of this survey was to assess if there is a market for Natural Language Processing (NLP)-enabled Computer-Aided Software Engineering (CASE) tools. Although conducted some time ago, this survey has pointed to the considerable use of UML in the companies surveyed (77%) and this shows that UML is a standard that has been keeping its acceptance in the industrial context for some time. But specifically with regard to the documentation of requirements, they found that "The majority of the documents available for requirements analysis are in natural language and are either furnished by the customer or obtained by means of interviews". Precisely, 79% of requirements documents were developed in common Natural Language (NL), 16% in structured NL (e.g. templates, forms), and only 5% used some sort of formal language. By merging the two first classes, 95% of the requirements documents were written in NL.

The key point is that NL is straightforward and, of course, stakeholders are familiar with such language. Mich et al. (2004) also mention some advantages of using NLP tools to support the development of software systems in general and requirements analysis in particular. Among others, these NLP tools may help the analyst to concentrate on the problem rather than on the modeling, achieve traceability as from the very first documents produced, and manage more efficiently the problem of changes in requirements. Moreover, NL may be associated with requirements modeling methods, like UML use case models where a textual description exists in order to narrate the behavior by means of a sequence of actor-system interactions (SINHA et al., 2007; FANTECHI et al., 2003). So even in use case requirements specificatons NL is present to describe the UML use case models. The conclusion is that in greater or lesser extent, due to its simplicity, NL is still widely used to develop software requirements specifications or other artifacts created for documenting requirements.

Unfortunately, there are serious problems if NL is selected for the creation of software requirements specifications. Most notably, ambiguity (BERRY et al., 2003; FANTECHI et al., 2003), poor understandability, incompleteness and inconsistency (GNESI et al., 2005) make software requirements specifications unclear, and since they are created early within the software development lifecycle, their defects affect the next software artifacts, including source code, to be developed. Furthermore, poorly developed requirements specifications can give rise to risks such as acceptance, availability, performance, reliability, reproducibility, supportability, utility, cost, and schedule risks (WILSON et al., 1997).

On the other hand, Nuseibeh and Easterbrook (2000) 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". However, in order to follow these guidelines, the aforementioned issues (inconsistency, incompleteness, etc.) must be first properly detected. For instance, if an inconsistency defect 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 to adopt the tolerating inconsistency approach, the inconsistency defect itself must be first detected.

1.2 Objectives

Summarizing the various observations that have been previously emphasized and that, in fact, were the reasons that motivated the development of this PhD thesis, it can be stated:

a) In model-based system and acceptance test case generation, and particularly taking into account NL requirements documents, identification of scenarios, their respective models and test case generation are arduous and time-consuming tasks, especially for real complex applications such as software embedded in on-board computers of satellites. Moreover, it is interesting to think about a solution based on mathematical methods to define the total number of scenarios to stimulate the Implementation Under Test (IUT), i.e. the software that is the target of testing, with test cases;

b) Despite the many benefits related to formal methods, such approaches are not largely adopted for software development in general. On the other hand, NL is still widely used to develop software requirements specifications, either alone or making part of UML use case models;

c) Defects in software requirements specifications like inconsistency and incompleteness are propagated to other artifacts created during the software development lifecycle. Thus, several undesirable situations may occur such as the customer does not accept the delivered product, a significant increase in costs for developing the software product, and poor performance.

This PhD thesis has therefore two objectives as defined in the sequence.

1.2.1 Primary Objective

The primary objective of this PhD thesis is to generate model-based system and acceptance test cases considering NL requirements deliverables (artifacts).

This is really a challenging objective to be achieved due to the various problems previously mentioned in NL software requirements specifications and also due to the complexity of deriving a model that is appropriate for generating test cases from NL requirements.

1.2.2 Secondary Objective

The secondary objective of this PhD thesis is the detection of incompleteness in software specifications.

This goal therefore relates to improving the quality of the software specifications and in the end, it will almost certainly improve the quality of the software product as a whole.

1.3 A proposal to achieve the objectives

In order to reach the goals set for this PhD thesis, a methodology named **SOLIMVA**[4] was then developed. The first version of the SOLIMVA methodology, version 1.0, was developed to achieve the primary objective of this PhD thesis. For this purpose, a tool, also called SOLIMVA, was designed and implemented and such a tool makes it possible to automatically translate NL requirements into Statechart models (HAREL, 1987). Once the Statecharts are created, another tool, the *Geração Automática de Casos de Teste Baseada em Statecharts* (GTSC - Automated Test Case Generation based on Statecharts) environment (SANTIAGO et al., 2008b), is used to generate Abstract Test Cases which are later translated into Executable Test Cases. Version 1.0 of the SOLIMVA methodology relies on combinatorial designs (MATHUR, 2008) to identify scenarios for system and acceptance testing. The SOLIMVA tool uses computational linguistics techniques (Part Of Speech Tagging (TOUTANOVA et al., 2003), Word Sense Disambiguation (NAVIGLI, 2009)) in order to reason about some behavioral and semantic aspects of the model to be generated.

The SOLIMVA methodology was applied to a main case study, a space application software product (SANTIAGO et al., 2007) related to the Space Segment, and the methodology was compared with a previous manual approach developed by an expert under two aspects: coverage of test objectives and characteristics of Executable Test Cases. This case study has almost all the functions of software for data handler computers for space applications, and thus its characteristics are representative of an important class of complex software in the Space Segment. In addition, guidelines

[4]The term "SOLIMVA" is a tribute to the family of the author of this PhD thesis.

to apply the SOLIMVA methodology to a second case study of the space domain related to the Ground Segment (CARDOSO et al., 2008) are also presented.

The SOLIMVA methodology was then extended to achieve the secondary objective of this PhD thesis. Model Checking (CLARKE; EMERSON, 2008; QUEILLE; SIFAKIS, 2008; BAIER; KATOEN, 2008) combined with k-permutations of n values of variables and specification patterns (DWYER et al., 1999) were used to address this goal. Thus, this extension generated version 2.0 of the SOLIMVA methodology. The new and most important activity of version 2.0 of SOLIMVA was applied to the same case study of the Space Segment (SANTIAGO et al., 2007) in which version 1.0 of the SOLIMVA methodology was used.

With respect to Verification and Validation, version 1.0 of the SOLIMVA methodology is related to Software Testing (type of dynamic analysis technique) while the new activities of version 2.0 of the methodology are related to Software Inspection (type of static analysis technique). In the perspective of RE, the new activities of version 2.0 of the SOLIMVA methodology fall within the activity of modeling and analysis.

Some observations should be made about the meaning of the term “SOLIMVA” used in the text of this PhD thesis. As explained earlier in this section, SOLIMVA is a methodology that has two versions. The terms “SOLIMVA methodology” and “SOLIMVA” both refer to the methodology itself. Depending on the context and if they are used alone (without explicit indication of the version number), these terms can represent either version 1.0 or version 2.0 of the methodology. In addition, version 1.0 of the SOLIMVA methodology is supported by two main tools: the SOLIMVA tool and the GTSC environment. In the context of this work, the most relevant of these two tools is the SOLIMVA tool. When referring to the SOLIMVA tool in the text of this PhD thesis, the term “SOLIMVA tool” will be explicitly written to avoid confusion with the SOLIMVA methodology.

1.4 Organization of this PhD thesis

This introductory chapter presented the context, the motivations that led to the development of this PhD thesis as well as the objectives of this work and a general description of the proposal to achieve the goals. The organization of the remaining text of this PhD thesis is as follows:

- Chapter 2 presents the theoretical basis for developing this PhD thesis and related work. The theories discussed in this chapter are, among others, Model-Based Testing, combinatorial designs, Formal Verification (Model Checking), and Natural Language Processing;
- Chapter 3 presents version 1.0 of the SOLIMVA methodology which addresses the primary objective of this work. First, a brief description of the main case study is made. The methodology was applied to this main case study. This description is important because, by detailing version 1.0 of the SOLIMVA methodology, simple examples are considered and most of these examples are taken from this case study;
- Chapter 4 shows in detail the application of version 1.0 of the SOLIMVA methodology and supporting tools (SOLIMVA tool and GTSC environment) to the main case study. The comparison between the SOLIMVA methodology and the expert's (manual) approach under the aspects of coverage of test objectives and characteristics of Executable Test Cases are also detailed in this chapter. Moreover, guidelines to apply SOLIMVA to a second case study are also presented;
- Chapter 5 presents version 2.0 of the SOLIMVA methodology to address the secondary objective of this work. The results obtained by applying the new and most important activity of version 2.0 of SOLIMVA to the main case study are also shown;
- Chapter 6 shows the conclusions of this work as well as future directions to follow;
- Appendix A presents all NL requirements collected from the deliverables (documents) prepared for the main case study, and which were used by the SOLIMVA methodology and by the SOLIMVA tool to generate model-based system and acceptance test cases;
- Appendix B shows an overview of the most relevant tool that supports the SOLIMVA methodology, i.e. the SOLIMVA tool, in order to generate model-based system and acceptance test cases considering NL requirements deliverables.

2 THEORETICAL BASIS

This chapter presents the theoretical basis for developing this PhD thesis and related work. The chapter is divided into four major sections. Sections 2.1 and 2.2 and their respective sections describe the theoretical background. Section 2.3 and its sections present some of the research literature related to this PhD thesis. Section 2.4 has concluding remarks on this chapter.

2.1 Verification and Validation

As mentioned in Chapter 1, Verification and Validation is a discipline related to Software Assurance. In this PhD thesis, these terms are defined in accordance with the *IEEE Standard Glossary of Software Engineering Terminology* (IEEE, 1990) as follows:

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.

There is some confusion about the meaning of Verification and Validation in the Software Engineering community. However, by the definitions given above, Validation relates to checking that the software product will achieve what customers actually need, while Verification refers to whether the software is well-engineered and that it has the fewest number of defects. Verification will help to determine whether a high-quality software has been produced, but it will not ensure that the software is indeed useful (EASTERBROOK, 2010). Nevertheless, there is another definition for Verification which will be discussed in Section 2.1.2.

Three important concepts are related to Verification and Validation: *fault*, *error*, and *failure*. Dependability researchers consider them the impairments to dependability (LAPRIE; KANOUN, 1996). In this work, fault and error are defined according to the *IEEE Standard Glossary of Software Engineering Terminology* (IEEE, 1990), and failure in accordance with Laprie and Kanoun (1996) as follows:

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.

Defect is another term used as a synonym of 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 went through 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.

Despite the fact that the above definition given for defect (fault) seems to be more related to the artifact source code, in this work the term defect is used to represent any **non-correctness** in several other artifacts created during the software development. For example, a defect may be a wrong instruction in the source code but it can also be incompleteness in requirements documents.

2.1.1 Software Testing and test case generation

The simplest definition of Software Testing was given by Myers (2004): "Testing is the process of executing a program with the intent of finding errors".[1] Thus, a good test strategy is one that finds defects in the IUT. Myers (2004) 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 defects exist. Moreover, in general, no one wants to find defects in one's own product. It is a destructive feeling.

Software Testing is a process/method related to Verification and Validation. A Software Testing process is composed of a set of activities and organizations can adopt different solutions. Typical activities include *Plan Test* (IEEE, 1998), *Gener-*

[1]However, in accordance with the definition given earlier and also with this perspective of Myers (2004), it is more adequate to use the term "defect" because the main intention of Software Testing would be to find non-correctness in the source code.

ate/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). In the sequence, only the *Generate/Select Test Cases* activity will be discussed because it is the one most associated with this PhD thesis.

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 defects (faults). The *Generate/Select Test Cases* activity aims to deal with this issue. Some important definitions are provided below.

Definition 2.1. A *test case* consists of a set of *test input data* and the corresponding *expected result*. The expected result is the outcome that is expected to occur when the IUT is stimulated by a certain test input data.

Definition 2.2. A set of test cases is a *test suite*.

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

Table 2.1 adapts with some slight modifications Mathur's proposal for the classifier *source of test generation*. Table 2.2 adapts the classifier *lifecycle phase in which testing takes place*.

Table 2.1 shows that test cases can be generated from informally or formally specified requirements and without the aid of the source code. This approach is known as *black box testing*. Category 1 in this table shows examples of black box testing techniques based on informal requirements, among others, ad hoc testing and some heuristics like equivalence partitioning, boundary-value analysis, and the Category-Partition method (OSTRAND; BALCER, 1988).

Categories 2 and 3 in Table 2.1 will be discussed in Section 2.1.1.1. Test cases derived from the source code are known as *white box testing* as shown in category 4 in this table. According to Mathur (2008), code can be used directly or indirectly for test case generation. In the former case, a tool, or a test designer, examines the code

Table 2.1 - Adaptation of Mathur's definition for the classifier *source of test generation*. Caption: Cat = Category

Cat	**Source**	**Approach**	**Example**
1	Requirements (Informal)	Black Box	- Ad hoc testing - Boundary-value analysis - Equivalence partitioning - Category-Partition method - Classification trees - Random testing
2	Requirements (Formal)	Model-based Specification	- Statechart testing - Finite State Machine testing - B testing - Z testing - Pairwise testing
3	UML	UML-based Document	- UML-based testing (state machine, class, sequence, etc. diagrams)
4	Source Code	White Box	- Control Flow testing - Data Flow testing - Mutation testing

Table 2.2 - Adaptation of Mathur's definition for the classifier *lifecycle phase in which testing takes place*

Phase	**Testing Level**
Coding	Unit testing
Integration	Integration testing
System Integration	System testing
Prerelease	Acceptance testing
Maintenance	Regression testing

and focuses on a given path to be traversed. A test case is generated to cover this path. In the indirect case, test cases generated using a black box testing technique are assessed against white box test coverage criterion. Additional test cases can then be derived to cover the uncovered portions of the source code. Control flow, data flow and mutation testing can be used for both direct and indirect white box test case generation. White box and black box testing techniques have both benefits and disadvantages and they can be seen as complementary approaches within a Software Testing process.

Table 2.2 takes into account the fact that testing activities occur throughout the software development lifecycle. When developers are coding, unit testing can be applied. As a system is usually decomposed into several units, integration testing takes place when such units starts to be integrated. System testing occurs considering that the entire system has been developed. A customer can generate his/her own test suites in order to accept the product developed by a supplier. Actually, an independent organization may be in charge to create such tests and to 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 important to mention that black box testing techniques can be used for all phases of the software development 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 the source code when the entire system is considered.

2.1.1.1 Model-Based Testing

As highlighted in Chapter 1, Model-Based Testing has different interpretations depending on the researchers. According to Mathur (2008), "Model-based or specification-based testing occurs when the requirements are formally specified, as for example, using one or more mathematical or graphical notations such as Z, statecharts, and an event sequence graph, and tests are generated using the formal specification." In this sense, Model-Based Testing is also a form of black box testing. Therefore, the category 2 of Table 2.1 is the one that fits the definition of Mathur (2008). Among the formal methods used for system and acceptance model-based test case generation are Statecharts (HAREL, 1987; HAREL et al., 1987; SANTIAGO et al., 2008b), Finite State Machines (FSMs) (SIDHU; LEUNG, 1989; LEE; YANNAKAKIS, 1996), and Z (HIERONS, 1997; CRISTIÁ et al., 2010).

On the other hand, El-Far and Whittaker (2001) state that Model-Based Testing is an approach that bases common activities of the Software Testing process such as test case generation (*Generate/Select Test Cases*) and test results evaluation (*Evaluate Test Results*) on a software model of the IUT. Such a definition considers test cases being generated based on formal specifications and other non-formal notations, like UML models (OMG, 2007). Therefore, categories 2 and 3 of Table 2.1 are encompassed in this definition.

According to Utting and Legeard (2007), Model-Based Testing consists of a test strategy in which test cases are derived entirely or partly from a model that describes some aspect (e.g. functionality, safety, performance, etc.) of a software. The application of Model-Based Testing requires that the behavior or structure of the software (or part thereof) has been formalized by means of models with well-defined rules (such as formal methods, UML diagrams, etc.). They also say that a Model-Based Testing technique can be applied to any type of testing (functional, structural, etc.). Since a Model-Based Testing technique can be applied to structural (white box) testing then Model-Based Testing can be regarded as a form of white box testing too. Thus, categories 2, 3 and 4 of Table 2.1 are in accordance with their definition.

In the survey presented by Dias Neto and his partners (DIAS NETO et al., 2007; DIAS NETO, 2009), they divided the obtained articles into five categories. Two categories were defined taking into account articles which describe Model-Based Testing techniques where models were extracted from the internal structure of software which includes, for instance, the source code. So, test cases derived from Control Flow Graphs and Def/Use Graphs which represent the source code of the IUT were also considered. Therefore, their approach follows the definition given by Utting and Legeard (2007).

It makes sense to consider Model-Based Testing a form of white box testing. After all, a graph (Control Flow Graph, Def/Use Graph) is nonetheless a model in which, in this case, models the source code. However, this perspective to consider Model-Based Testing a form of white box testing is not accepted by the entire testing community, as seen in the definition proposed by Mathur (2008).

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 2.3. An FSM M 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 $\delta : D_A \rightarrow S$, and
- λ is an output function $\lambda : D_A \rightarrow Y$.

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 an FSM as an acceptor too. Hopcroft and Ullman (1979) define a Determininistic Mealy machine, like the one described above, as a DFA with output. 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.

Complex software products usually present features like parallel activities and hierarchy. These features are very difficult to represent using FSMs, so this leads to considering higher-level techniques as Statecharts. The formal definition of Statecharts is more elaborate than that of FSMs. The detailed definition of the formal syntax and semantics of the Statecharts language can be seen in Harel et al. (1987). For information purposes, the syntax of Statecharts is defined over the following basic sets of elements: states, primitive events, primitive conditions, and variables. Using these basic sets of elements, extended sets are defined: expressions, conditions, events, actions, labels, and transitions. Table 2.4 (see later in this chapter) gives a good idea of the formal syntax of the Statecharts language.

2.1.1.2 The Statechart Coverage Criteria Family

This section presents a brief description of the *Statechart Coverage Criteria Family* (SCCF) (SOUZA, 2000). SCCF is a family of testing coverage criteria for Statechart models. Test requirements established by the SCCF criteria are obtained from the Statechart reachability tree (MASIERO et al., 1994). A reachability tree is only a behavioral representation of Statecharts showing the possible configurations and paths that the system can reach. SCCF requires a reachability tree in order to generate, at the end of the process, test cases. SCCF will be discussed only to the extent that it relates to this PhD thesis. Some concepts that are essential for understanding the test criteria of SCCF are defined as follows.

Definition 2.4. A configuration C_i is a set of states that are active in a step of computation, and C_0 is the initial configuration. At each step is supposed that the events which are associated with the current configuration are valid and they fire the related transitions, so that it is possible to model the space of possible configurations of the system.

Definition 2.5. A path is a finite sequence of configurations $(C_0, C_i, C_j, \ldots, C_m, C_k)$, $k \geq 1$, so that the first configuration is the initial configuration of the reachability tree (C_0), and there is a transition from C_i to C_j, $\forall C_i, C_j \mid 0 \leq i < k$ and $j = i + 1$.

Definition 2.6. A simple path is a path P such that all the configurations that compose this path, except possibly the first and the last, are different.

Definition 2.7. A resettable path is a path P in which the first and last configuration of the path correspond to the initial configuration C_0, i.e. they are the paths that make the model returns to its initial configuration.

Figure 2.1 shows the inclusion (hierarchical) relation among the SCCF test criteria considering resettable models. The essential point of the inclusion relation is to define which test criteria include other test criteria. Criterion c_1 **includes** criterion c_2 if for any set of paths P that satisfies c_1 then P also satisfies c_2 for any model/specification. Criterion c_1 **strictly includes** criterion c_2, denoted $c_1 \longrightarrow c_2$, provided that c_1 includes c_2 but c_2 does not include c_1. Note that this is a transitive relation. Criteria c_1 and c_2 are **incomparable** if neither $c_1 \longrightarrow c_2$ nor $c_2 \longrightarrow c_1$ (RAPPS; WEYUKER, 1985). Thus, the all-paths test criterion strictly includes all-paths-k-configurations, all-one-loop-paths, all-simple-paths, etc. as well as the all-paths-k-C0-configuration test criterion. The all-simple-paths test crietrion stricly includes both all-loop-free-paths and all-configurations. However, all-simple-paths and all-transitions test criteria are incomparable.

Definitions of some test criteria of SCCF are provided below.

Definition 2.8. all-transitions: this test criterion requires that all transitions are traversed at least once by a test suite T.

Definition 2.9. all-simple-paths: this test criterion requires that all simple paths are traversed at least once by a test suite T.

Definition 2.10. all-paths-k-C0-configuration: this test criterion requires that all paths containing k repetitions of the initial configuration C_0 are traversed at least once by a test suite T. For $k = 2$, each path restarts the model once and for $k > 2$

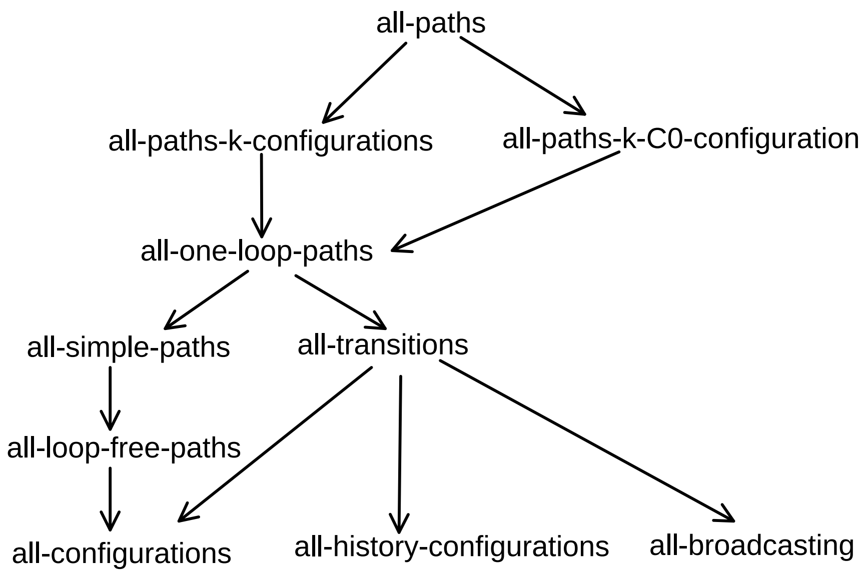


Figure 2.1 - Inclusion relation among the SCCF test criteria considering resettable models
SOURCE: Adapted from Souza (2000)

each path restarts the model $k - 1$ times.

Definition 2.11. all-paths-k-configurations: this test criterion requires that all paths containing at most k repetitions of each configuration are traversed at least once by a test suite T.

2.1.1.3 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 (defects) 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.

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*, $MCA(N, {l_1}^{k_1}{l_2}^{k_2}...{l_p}^{k_p}, t)$, 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.

2.1.2 Formal Verification

As mentioned earlier, there is another definition for Verification given in the *IEEE Standard Glossary of Software Engineering Terminology* (IEEE, 1990): formal proof of program correctness. This definition, although it seems to be more associated with software (program), is what is called **Formal Verification**.

Formal Verification refers to mathematical analysis of proving or disproving the correctness of a hardware or software system with respect to a certain specification or property (GANAI; GUPTA, 2007). By the extensive use of mathematical logic, Formal Verification has strong connections with theoretical computer science. The methods for analysis are known as *Formal Verification Methods* and they can be broadly classified into: **Theorem Proving** and **Model Checking**. Theorem Provers use mathematical reasoning and logical inference to prove the correctness of systems, and often require a specialist with substantial understanding of the system under verification. Model Checking will be briefly described in the next section.

2.1.2.1 Model Checking

Model Checking (CLARKE; EMERSON, 2008; QUEILLE; SIFAKIS, 2008; CLARKE; LERDA, 2007; BAIER; KATOEN, 2008) is probably the most popular Formal Verification Method in the academic community. According to Baier and Katoen (2008), "Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model". In the traditional approach, properties are generated based on requirements documents and they are formalized using some sort of temporal logic such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL). Thus, a property specifies the desired behavior of the system under consideration. If the finite-state model (transition system) does not satisfy a property, a counterexample is generated showing a trace that indicates the violation. On the other hand, the finite-state model describes the behavior of the system.

Dwyer et al. (1999) proposed a system of property specification patterns for finite-state verification. They proposed 8 patterns and 5 pattern scopes. Hence, based on a requirement, one identifies a pattern and the scope within the pattern that mostly characterize such requirement. Having decided which is the pattern and scope, they proposed a template to generate the properties in LTL, CTL, and Quantified Regular Expressions. Some descriptions of pattern/pattern scope are presented below with the corresponding CTL state formulae (in order to understand CTL formulae, Table 2.3 shows the path quantifiers and temporal modalities (operators) of CTL):

- **Absence Pattern and Globally Scope**: a given state/event p does not occur within the entire program/model execution. CTL formula: $\forall\Box\neg p$;

- **Response Pattern and Globally Scope**: a state/event p must always be followed by a state/event q within the entire program/model execution. CTL formula: $\forall\Box(p \rightarrow \forall\Diamond q)$;
- **Precedence Pattern and Globally Scope**: a state/event p must always be preceded by a state/event q within the entire program/model execution. CTL formula: $\neg\exists[\neg q \cup (p \wedge \neg q)]$.

The Absence Pattern and Globally Scope is indeed a **safety property** which is often characterized as “nothing bad should happen”. In the above descriptions, for brevity, the phrase “a given state/event occurs” means “a state in which the given state formula is true, or an event from the given disjunction of events occurs” (DWYER et al., 1999).

Table 2.3 - CTL’s path quantifiers and temporal modalities

Path quantifier		**Temporal modality**	
Notation	**Meaning**	**Notation**	**Meaning**
$\forall$	for all paths	$\Box$	always (globally)
$\exists$	for some path	$\Diamond$	eventually
		$\bigcirc$	next
		$\cup$	until

State explosion is one limitation of Model Checking. 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. execution time and/or available memory. Techniques have been proposed to tackle the state explosion problem such as *Symbolic Model Checking*, a technique that uses Binary Decision Diagrams (BDDs) to represent sets of states and transitions (BRYANT, 1986). However, BDD-based Symbolic Model Checking still presents some problems when trying to solve the state explosion problem (CLARKE; LERDA, 2007). Model Checking was originally conceived for verifying finite-state systems such as sequential circuit designs and communication protocols and, despite its limitations, it has been used by semiconductor manufacturers for hardware design and also to formally verify software products. There are plenty of available tools (Model Checkers) for applying Model Checking such as SPIN (HOLZMANN, 2003; BEN-ARI, 2008), NuSMV (FONDAZIONE BRUNO KESSLER / CARNEGIE MELLON UNIVERSITY / UNIVERSITY OF GENOVA / UNIVERSITY OF

TRENTO, 2011), UPPAAL (BEHRMANN et al., 2004), and *Java Pathfinder* (NASA AMES RESEARCH CENTER, 2011).

2.2 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 (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 (2000) distinguish into six categories the knowledge of language needed to engage in complex language behavior: 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, Word Sense Disambiguation (WSD), lexicalized and probabilistic parsing, Chomsky hierarchy, pumping lemma, first order predicate calculus, and many others. In the sequence, topics related to this work are briefly discussed.

Lexical category or POS is a linguistic category of words (or more precisely lexical items), which is generally defined by the syntactic or morphological behavior of the lexical item in question. Common linguistic categories include noun and verb, among others. POS can be used to inform how a word is pronounced, in stemming for information retrieval, and they are very often used for "partial parsing" texts, for example for quickly finding names or other phrases for information extraction applications (JURAFSKY; MARTIN, 2000). POS tagging or word category disambiguation is then the process of assigning POS to the words in a text based on both its definition as well as its context. The point is that a word can have different lexical categories, e.g. "bank" can be a verb or a noun. Thus, it is the task of POS tagging to determine the correct one.

2.2.1 Word Sense Disambiguation

According to Navigli (2009), "Word Sense Disambiguation (WSD) is the ability to identify the meaning of words in context in a computational manner". In the field of NLP, WSD has been studied for a long time. Even in the same lexical category a word can have different meanings (senses). For instance, the noun "bank" can be a depository financial institution, the building of the financial institution, or a long pile, just to name a few. Thus, WSD is used to identify the correct sense in a certain context.

WordNet is an electronic lexical database created and maintained at Princeton University (MILLER, 1998). The basic building block of *WordNet* is a *synset* consisting of all the words that express a given concept. Alternatively, one may say that *WordNet* encodes concepts in terms of sets of synonyms (the *synsets*) (NAVIGLI, 2009). Therefore, the design of *WordNet* resembles a thesaurus but in some ways it also resembles a traditional dictionary, providing definitions and sample sentences for their *synsets*. Consider the word "test". In version 2.1 of *WordNet*, "test" can be a verb or a noun. Examples of two *synsets* of the noun "test" are shown below:

- test#2, trial#2, run#2 – (the act of testing something; "in the experimental trials the amount of carbon was measured separately"; "he called each flip of the coin a new trial");
- trial#5, trial run#1, test#4, tryout#1 – (trying something to find out about it; "a sample for ten days free trial"; "a trial of progesterone failed to relieve the pain").

The sense number of each word in each *synset* is shown immediately after the word, and is preceded by #. Thus, the first *synset* says that sense number 2 of the words "test", "trial", and "run" have the same meaning, i.e. "the act of testing something". Likewise, the second *synset* says that sense numbers 5 of "trial", 1 of "trial run", 4 of "test", and 1 of "tryout" have the same meanings. Examples of two *synsets* of the verb "test" are shown below. The same remarks related to the meaning of the *synsets* made for the noun apply to the verb:

- test#1, prove#5, try#2, try out#1, examine#5, essay#2 – (put to the test, as for its quality, or give experimental use to; "This approach has been tried with good results"; "Test this recipe");
- quiz#1, test#3 – (examine someone's knowledge of something; "The teacher tests us every week"; "We got quizzed on French irregular verbs").

In *WordNet*, there are lexical relations, such as Antonymy and Pertainymy, which connect word senses included in the respective *synsets*, and semantic relations, such as Hypernymy (*is-a* relation), Hyponymy (inverse relation of Hypernymy), and Meronymy (*part-of* relation), which connect *synsets*. Thus, WordNet is also a semantic network.

Among the numerous approaches to address WSD, one in particular is highly relevant to this PhD thesis and a brief description of such a proposal is given below. A graph-based algorithm for WSD was proposed by Sinha and Mihalcea (2007). In their approach, they constructed a sense (label) dependency graph based on measures of word semantic similarity like the *Leacock and Chodorow* (LEACOCK; CHODOROW, 1998), *Jiang and Conrath* (JIANG; CONRATH, 1997), and *Lesk* (LESK, 1986) measures. These measures work well in the *WordNet* hierarchy. For example, consider the *Jiang and Conrath* (JIANG; CONRATH, 1997) measure which is defined as

$$jcn(c_1, c_2) = \frac{1}{IC(c_1) + IC(c_2) - 2 * res(c_1, c_2)}$$

where:

- $c_1, c_2 =$ concepts (*synsets*);
- $IC(c) = -logP(c)$ (Information Content);
- $res(c_1, c_2) = IC(LCS(c_1, c_2))$ (*Resnick* measure);
- $LCS =$ Least Common Subsumer.

The *Jiang and Conrath* measure is based on Information Content (IC) which is a measure of specificity for a concept (*synset* in *WordNet*). Higher values of IC are associated with more specific concepts, while lower values of IC are related to more general ones. IC is calculated based on frequency counts of concepts as found in a corpus of text. The frequency associated with a concept is incremented in *WordNet* each time that concept is observed, as are the counts of the ancestor concepts in the *WordNet* hierarchy (for nouns and verbs). Also note the importance of the Least Common Subsumer (LCS), the most specific concept that is a shared ancestor of the two concepts.

A weighted, undirected, not fully connected sense dependencies graph is derived by adding a vertex for each admissible sense of the words in a text, and an edge for each pair of senses of distinct words for which a dependency is identified. A window, *wn*, is defined so that no edges will be drawn between senses corresponding to words that are more than *wn* words apart, counting all running words, i.e.

nouns, verbs, adjectives, and adverbs. After the graph construction, the scores of senses are determined using some graph-based centrality algorithms like *indegree* and an adaptation of the *PageRank* algorithm. For instance, consider the *indegree* algorithm. For an undirected weighted graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, the indegree is defined as

$$indegree(V_a) = \sum_{V_b \in V} wg_{ab}$$

where wg_{ab} is the weight on the edge between V_a and V_b. In other words, the *indegree* of a vertex V_a is obtained taking into account the weights on the edges, and adding them together into a score. Finally, the approach proposed by Sinha and Mihalcea (2007) determines the most likely set of senses by identifying the sense with the highest score for each word.

2.3 Related work

The following sections present some of the research literature related to this PhD thesis.

2.3.1 Model-Based Testing

This section presents some important approaches related to Model-Based Testing.

Briand and Labiche (2002) presented the *Testing Object-orienTed systEms with the unified Modeling language* (TOTEM) approach based on UML diagrams addressing functional system testing. 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.

FSMs (SIDHU; LEUNG, 1989; LEE; YANNAKAKIS, 1996) have been commonly used for testing. Simplicity is one of the key advantages in using FSM and this technique has been adopted 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[2] 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 presented in Sidhu and 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*, by accounting for their associated semantics. However, 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 transitions as proposed in their approach. It is not very clear and probably it is not feasible to generate test cases with a model with these characteristics.

A Model-based approach to generate a set of conformance test cases for interactive systems, i.e. those which react to operations invoked by external environment, was proposed by 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 algorithm that generates a partition of the input domain from a Z specification has been introduced by 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

[2]Some authors prefer the term "method" rather than "criterion". This work will adopt the latter.

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.

Conformance and Fault Injection (CoFI) is another model-based test case generation methodology (AMBRÓSIO et al., 2007). In CoFI, the system behavior is partially represented in state-transition diagrams representing FSMs, and test cases are generated based on such FSMs. In order to use the methodology, the test designer must first identify a set of scenarios (services) to stimulate the IUT, and then he/she must precisely define the IUT's interfaces. CoFI is basically a use case-based testing approach (FROHLICH; LINK, 2000; BERTOLINO; GNESI, 2003) with some emphasis in hardware fault tolerance. One limitation of CoFI is not to provide mathematical-based guidelines for the test designer to identify the usage scenarios (services).

Several approaches have been proposed to generate test cases based on Statecharts. Hong et al. (2000) provides a way to derive EFSMs from Statecharts to devise test criteria based on control and data flow analysis. Binder (1999) adapted the W test criterion to a UML context and named it round-trip path testing, in which flattening a Statechart model is a prerequisite before using the criterion itself. 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 among three test criteria proposed by Offutt and Abdurazik (1999) and the very same round-trip path.

A system testing approach to cover *elementary transition paths* was proposed by Sarma and Mall (2009). The technique relies on the derivation of a System State Graph (SSG) based on UML 2.0 use case models, sequence diagrams, and Statechart models. Their technique aims to satisfy the test criterion *transition path coverage*, which states that each elementary transition path p of the SSG must be covered at least once by a test suite T. Their work presents some limitations but the most severe of all is related to 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, the behavior of a loop is like a control structure *if*. 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 may restrict their approach and the problem of testing applications in the presence of huge number of paths is not properly addressed.

Frohlich and Link (2000) presented a system testing method based on textual descriptions of UML use cases. They translated a use case description into a UML Statechart (UML state machine) and, after that, they applied Artificial Intelligence 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.

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 case 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. One limitation of their approach is that it does not support orthogonality to model each component, a feature quite common in current software products.

Santiago et al. (2006) proposed a methodology to transform hierarchical and orthogonal Statecharts into FSMs in order to generate test cases, with the support of the *PerformCharts* tool (VIJAYKUMAR et al., 2006). *PerformCharts* is a tool originally designed for performance evaluation. The tool allows that a reactive system modeled in Statecharts can be converted into a Markov Chain (BAIER; KATOEN, 2008). Then, based on analytical solutions, steady-state probabilities are determined for the corresponding Markov Chain and these are the basis for calculating performance measures. However, an FSM can be used as a representation of a Markov Chain. This was the motivation to use *PerformCharts* in combination with an implementation of the FSM test criterion switch cover (1-switch) (PIMONT; RAULT, 1976) to generate test cases.

PerformCharts is currently incorporated into the GTSC environment (SANTIAGO et al., 2008b). GTSC is an important part of this PhD thesis and it will be briefly discussed in Section 3.2. GTSC has been developed and used in the context of research projects in the space domain and the environment is beginning to be applied to a real application, a stratospheric balloon project to lift up a high energy astrophysics experiment under development at Instituto Nacional de Pesquisas Espaciais (INPE - National Institute for Space Research). Recently, a proposal for combining

Statechart-based and Z-based testing was presented (CRISTIÁ et al., 2010). GTSC has been used in conjunction with the *Fastest* tool (CRISTIÁ; MONETTI, 2009) in order to meet this goal.

PerformCharts and consequently GTSC support almost all features of the Statecharts language. Table 2.4 shows the features suported by *PerformCharts* with respect to the formal syntax of the Statecharts language as described in Harel et al. (1987). The user then has enough flexibility to model a system using the main features of Statecharts available in GTSC (*PerformCharts*). The following sets are considered in Table 2.4:

- S is the set of states,
- V_p is the set of variables,
- V is the set of expressions,
- C_p is the set of primitive conditions,
- C is the set of conditions,
- E_p is the set of primitive events,
- E is the set of events, and
- A is the set of actions.

Table 2.4 - Subset of the Statecharts language supported by *PerformCharts*/GTSC. Features marked with X are supported

Syntax feature	**Support**
States	X
Shallow and Deep History	X
Hierarchy Function	X
Type Function	X
Default Function	X
Expressions: If k is a number then $k \in V$	X
Expressions: If $v \in V_p$ then $v \in V$	X
Expressions: If $v \in V$ then $current(v) \in V$	
Expressions: If $v_1, v_2 \in V$ and op is an algebraic operation then $op(v_1, v_2) \in V$	X
Conditions: $T, F \in C$, T, F stand for $true$, $false$, respectively	X
Conditions: If $c \in C_p$ then $c \in C$	X

(Continues)

Table 2.4 - Conclusion

Syntax feature	**Support**
Conditions: If $s \in S$ then $in(s) \in C$	X
Conditions: If $e \in E$ then $not\_yet(e) \in C$	
Conditions: If $u, v \in V$, $R \in \{=, >, <, \neq, \leq, \geq\}$ then $u\ R\ v \in C$	X
Conditions: If $c \in C$ then $current(c) \in C$	
Conditions: If $c_1, c_2 \in C$ then $c_1 \vee c_2$, $c_1 \wedge c_2$, $\neg c_1 \in C$	X
Events: $\lambda \in E$, λ is the *null* event	X
Events: If $e \in E_p$ then $e \in E$	X
Events: If $c \in C$ then $true(c) \in E$	X
Events: If $c \in C$ then $false(c) \in E$	
Events: If $v \in V$ then $changed(v) \in E$	
Events: If $s \in S$ then $exit(s)$, $entered(s) \in E$	X
Events: If $e_1, e_2 \in E$ then $e_1 \vee e_2$, $e_1 \wedge e_2 \in E$	X
Events: If $e \in E$, $c \in C$ then $e[c] \in E$	X
Actions: $\mu \in A$, μ is the *null* action	X
Actions: If $c \in C_p$, $d \in C$ then $c := d \in A$	
Actions: If $v \in V_p$, $u \in V$ then $v := u \in A$	X
Actions: If $a_i \in A$, $i = 0, \ldots, n$ then $a_0; \ldots; a_n \in A$	
Labels	X
Transitions	X

Hierons et al. (2009) published a survey which described formal methods, Software Testing, and a number of ways in which a formal specification can be used in order to assist testing. They divided formal specification languages into several categories, among others *Model-Based Languages* (e.g. Z(SPIVEY, 1998)), *Finite State-Based Languages* (e.g. FSM[3](LEE; YANNAKAKIS, 1996), Statecharts (HAREL, 1987)), and *Process Algebra State-Based Languages* (e.g. Communicating Sequential Processes (CSP) (HOARE, 1985)). They concluded that "software testing benefits from the presence of a formal specification in a number of important ways. In particular, the presence of a formal specification aids test automation and allows the tester to reason about test effectiveness".

The literature related to Model-Based Testing is very extensive. For example, after the completion of the second part of the survey on characterization of Model-Based

[3]Traditionally, an FSM is seen as a computational model and not as a language (HOPCROFT; ULLMAN, 1979).

Testing techniques (DIAS NETO, 2009), 271 articles describing a total of 219 different techniques of Model-Based Testing were found. Some additional observations about the final version of this systematic review should be mentioned. These observations take into account all the 219 techniques found. Considering the *lifecycle phase in which testing takes place* classifier (Table 2.2), 71% of the surveyed techniques were used for system testing, 16% were applied to unit testing, 15% were used for integration testing, and 3% for regression testing. Model-Based Testing was originally intended for system testing and thus this explains the majority of approaches related to this testing level. The percentage of Model-Based Testing techniques that have tools to support test case generation was 69.86% while in 30.14% of the techniques there were no evidence that tools were developed for this purpose. Regarding the model/language, 34.70% of the Model-Based Testing techniques used UML models so that the most adopted were state machine (variant of Harel's Statecharts), class diagram, and sequence diagram. On the other hand, 65.30% of the techniques adopted non-UML methods and among the most used are FSM, Z language, graphs, and Markov Chains.

2.3.2 Software Testing based on Natural Language requirements

Only one publication was found in the literature that proposed to generate test cases starting from NL requirements: *Text Analyzer* (SNEED, 2007). It is a tool that supports black box testing and it is intended to be used for system and acceptance testing. *Text Analyzer* needs heavy intervention from the user 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 IUT. Such objects are in turn the elements 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 seems to require considerable time and that may make the approach less attractive, especially considering complex NL requirements documents.

2.3.3 Analysis of defects in Natural Language requirements

Literature has been addressing the analysis of software requirements to detect defects such as ambiguity, incompleteness, and inconsistency. As said in Chapter 1, NL is still a fairly common option to build software requirements specifications or other artifacts created for documenting requirements. This section will present some approaches that deal with the analysis of defects in NL requirements documents such as ambiguity, incompleteness, and inconsistency.

Quality Analyzer for Requirements Specification (QuARS) is a tool that enables the user to automatically analyze NL requirements (LAMI; TRENTANNI, 2004; GNESI et al., 2005; FANTECHI et al., 2003; FERGUSON; LAMI, 2005; BUCCHIARONE et al., 2008). Three categories of quality properties should be accounted for when analyzing NL specifications: expressiveness (ambiguity mitigation and understandability improvement), 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.

The analysis performed by QuARS is limited to syntax-related issues of NL requirements regarding expressiveness (ambiguity, understandability). Despite the remarkable work, it is not evident whether the mechanisms employed by QuARS are sufficient to detect all sorts of ambiguity. QuARS is able to partially support inconsistency and incompleteness analysis by clustering the requirements, also known as View Derivation, according to specific topics. The quality model for expressiveness has sub-characteristics associated with inconsistency (under reference) and incompleteness (underspecification). However, QuARS does not address completely incompleteness and inconsistency because these are complex issues requiring analysis of the semantics of the NL sentences (FANTECHI et al., 2003). Moreover, incompleteness and inconsistency analysis is mostly done manually by examining the Views.

Automated Requirements Measurement (ARM) is a tool developed to provide metrics so that NASA project managers can assess the quality of NL requirements specifications and identify risks that poorly developed requirements would introduce into their projects (WILSON et al., 1997). ARM has an associated quality model similar to that developed for QuARS. This quality model was defined by compiling first a list of quality attributes (e.g. complete, consistent, correct, unambiguous, etc.) that requirements specifications are expected to exhibit. After that, a list of aspects of a requirements specification that can be objectively and quantitavely measured was developed. The two lists were analyzed to identify relationships between what can be measured and the desired quality attributes. This analysis resulted in the identification of *categories* (equivalent to sub-characteristics in the model of QuARS) of sentences and individual items (i.e. words and phrases) that are *primitive indicators* of the specification's quality and that can be detected and counted by using the document text file. The set of primitive indicators were then refined by using a database of words and phrases derived from the analysis of 46 requirements specifications, acquired from a broad cross section of NASA projects.

It seems that QuARS employs a more sophisticated analysis than ARM because QuARS has an embedded syntax parser which helps in the mitigation of ambiguity. Moreover, ARM suffers from the same issues of QuARS because the analyis which is performed by ARM is not enough to adequately address defects like incompleteness and inconsistency in requirements specifications: ARM basically scans the requirements documents searching for the primitive indicators, totals and reports the occurrence of such indicators individually and by category.

CIRCE is an environment that supports modeling and analysis of requirements described in NL (AMBRIOLA; GERVASI, 2006; AMBRIOLA; GERVASI, 1997). The tool parses and transforms NL requirements into a forest of parse trees. To do that, CIRCE uses a domain-based parser called CICO. By defining requirements in accordance with the formal model embedded in CIRCE, 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 greatest issue related to CIRCE is the difficulty to express the domain. 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 need to write *Model, Action, Substitution* (MAS) rules, which are formal rules that drive the CICO's parsing algorithm, can be a significant obstacle for practitioners who are not specialized in the formal rules and wish to use the tool.

Both forms of developing NL requirements, unrestricted and controlled approaches, have supporters. Mich (1996) presented the *Natural Language - Object Oriented Production System* (NL-OOPS), a tool that 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 NLP 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 requirements but there is no evidence that it supports automated detection of incompleteness and inconsistency.

However, Ambriola and Gervasi (2006) mentioned that the lack of domain knowledge is a limiting factor for using systems based on unrestricted NL requirements taking into account real and complex projects. 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.

Park et al. (2000) proposed a tool to support analysis of NL requirements by measuring the similarity between requirement sentences written in Korean. The similarity measurement method combines a sliding window model and a syntactic parser, and uses *z-scores* and *Salton's cosine coefficients*. Their approach to detect ambiguity is similar to that used in QuARS (GNESI et al., 2005), by identifying indicators (words) that might make a requirement ambiguous, and putting them into a repository. Their tool supports traceability between NL documents by measuring the similarity between a sentence in a high-level document and a sentence in a low-level document. Although the support for traceability between NL documents may eventually improve the detection of incompleteness defects, the detection itself is indeed done manually by the user when looking at the results in the interface of the tool and, for example, realizing that there are no sentences in the low-level document associated with a given sentence in the high-level document. But this is more a problem of traceability than incompleteness which in this case, the main issues concern the omission of requirements and the lack of software responses to all possible input data (IEEE, 1984).

Still on the work of Park et al. (2000), their tool supports the detection of inconsistency defects between NL requirements and also duplication of requirements in a single document. However, identifications of inconsistency and duplication are indeed performed by the user when looking at NL sentences and candidate sentences in the interface of the tool. The authors claimed that their system allows an analyst to improve completeness of a single NL document but again this is not the case.

Gervasi and Zowghi (2005) proposed a formal framework for identifying, analyzing, and managing inconsistency in NL requirements derived from multiple stakeholders.

A prototype tool, CARL, was developed incorporating all the techniques described in the reference. They focused on a particular kind of inconsistency, *logical contradiction*, and the authors claim that the framework supports the detection of both explicit and hidden inconsistencies.[4] 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"[5] are used aiming at detecting inconsistencies. Despite these remarkable 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 considering that the example shown in the reference is very simple.

Hunter and Nuseibeh (1998) proposed a formal approach to reason, analyze, and accomplish actions in inconsistent specifications. It seems that such a work influenced in some ways the development of CARL (GERVASI; ZOWGHI, 2005). They adopted the tolerating inconsistency approach, 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 (it is the same case study used by Gervasi and Zowghi (2005)).

Attempto Controlled English (ACE) is a controlled NL specifically constructed to write specifications (FUCHS et al., 1999; FUCHS et al., 2000). ACE is a subset of standard English, and its specifications are computer-processable and can be translated into first order logic. In order to constrain structural ambiguity of NL, ACE avoids some ambiguous constructs, deals with others by means of interpretation rules, and the user is required to rephrase the input when necessary. There are tools based on ACE like the *ACE Parser* (APE), the *ACE Reasoner* (RACE), and the *AceRules* (UNIVERSITY OF ZURICH, 2009). Despite the significant effort in handling ambiguity, inconsistency and incompleteness issues are not in the scope of ACE. Although it is

[4]In hidden inconsistencies, the inconsistency occurs due to the consequences of some requirements rather than the requirements themselves.

[5]Actually, CARL does not really apply Model Checking according to its most common definition (CLARKE; LERDA, 2007; BAIER; KATOEN, 2008).

claimed that ACE combines NL with formal methods and it is easier to learn and use than formal languages, the language is very restricted. Very restricted versions of NL are often comparable to formal languages with NL-like keywords (AMBRIOLA; GERVASI, 2006).

Kim and Sheldon (2004) presented a method that models and evaluates NL software requirements specifications using the Z formal language and Statecharts. Their method transforms an NL specification into a Z specification which in turn derives the Statechart 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 detection of defects.

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 explores 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 applied to a simple case study and, even so, a significant number of SAO triads were incorrectly extracted (21%) as a large number of extractions were not detected as well (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.

Lu et al. (2008) presented the *Model-driven Object-oriented Requirement Editor* (MOR Editor), a tool that supports requirement document modeling and model-driven document editing. According to the authors, the user can reuse requirements patterns in order to help avoiding incompleteness in requirements. It seems that the editor can help handling inconsistency between requirements documents and artifacts of other phases of the software development lifecycle, but there is no evidence that inconsistency in the requirements themselves is treated. It is possible to transform the informal (NL) requirements into *Model-based OO Requirement Mod-*

els (MOORMs), templates from which software requirements can be instantiated. Although the authors stated that, it was not very clear how the tool can effectively aid in the detection of incompleteness defects in requirements, the approach lacks mathematical formalism, and the translation from NL requirements into MOORMs is not straightforward.

Konrad and Cheng (2006) presented a process that supports the specification and analysis of UML models with respect to behavioral properties specified in NL. 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 formalize properties in LTL (BAIER; KATOEN, 2008). Therefore, the aim is to check UML models against the NL properties (requirements) and is not to detect problems in the NL requirements themselves.

NATT OCH DAG et al. (2002) presented empirical evaluations of the benefits of automated similarity analysis of NL requirements, where information retrieval techniques were used to statistically measure requirements similarity. The similarity analysis aimed at detecting duplicate requirements and was focused on market-driven development organizations, where software is developed for a large market, there is a high pressure on short time-to-market, and requirements arrive continuously and may, at any time, affect previous, current, and coming releases of the software product. The identification of duplicates is important to avoid doing the same job twice, assigning the same requirement to different developers, or achieving two solutions to the same problem. They concluded that such automated similarity analysis is promising not only for duplicate requirements identification but also for requirements interdependencies. And, according to Fantechi et al. (2003), "the automatic determination of clusters of requirements dealing with the same arguments may support the human analysis, aimed at detecting inconsistencies and discrepancies, by focusing on smaller sets of requirements".

2.3.4 Translation of requirements

This section presents some studies related to the translation of requirements from one notation into a different notation. In particular, approaches that translate NL requirements into formal methods can be quite convenient because they relieve professionals of the cost of learning a formal method and, at the same time, provide the requirements converted so that the benefits of formalization can be explored during the development of software. In Section 2.3.3, CIRCE (AMBRIOLA; GERVASI, 2006; AMBRIOLA; GERVASI, 1997), CARL (GERVASI; ZOWGHI, 2005), ACE (FUCHS et al., 1999; FUCHS et al., 2000), the *SPIDER* tool suite (KONRAD; CHENG, 2006), and Kim and Sheldon method (KIM; SHELDON, 2004) are examples of such approaches.

Liang and Palmer (1994) discussed the correspondence between NL requirements sentence structure patterns and events/transitions concepts in state-transition diagrams representing EFSMs. The goal was how to extract events and transitions from conditional sentences. In order to support such extraction, a pattern matching and clustering-based approach was proposed. The clustering algorithm used a similarity measure and was applied to all event clauses. Hence, each resulting cluster contains similar events. The same algorithm can be applied to extract conditions and actions (outputs). Once events are identified, the original requirements are clustered based on events to identify related-transition information. There was no tool to automate this approach. The identification of events/conditions/actions per se is manually accomplished by the user by examining clusters.

Fraser et al. (1991) proposed to bridge the gap between informal and formal requirements specification languages. They used Structured Analysis (SA), by means of Data Flow Diagrams (DFDs), and the Vienna Development Model (VDM) as surrogates for informal and formal languages, respectively. Two approaches were presented for integrating SA and VDM. The first approach used the SA model of a system to guide the Requirements Engineer's understanding of the system and the development of VDM specifications. The second one proposed a rule-based method for generating VDM specifications from a set of corresponding SA specifications. This proposal did not start the translation directly from NL requirements.

There are also approaches that aim to automatically translate NL sentences for the purpose of Formal Verification. In some of these studies, the goal is to automatically translate NL requirements into properties formalized in some sort of logic (HOLT et al., 1999; NELKEN; FRANCEZ, 1996; FANTECHI et al., 1994) while others aim to automatically transform NL sentences into a model (KOH; SEONG, 2009). Some of these approaches are discussed in the next section.

2.3.5 Formal methods and requirements specifications

Some researchers have made efforts in applying formal methods to requirements specifications with diverse purposes such as to perform analysis of these deliverables themselves and to generate test cases. This section details some of these efforts.

NL2ACTL is a tool for the formalization of requirements (FANTECHI et al., 1994). NL2ACTL supports the automatic translation of NL sentences into formulae of Action Computation Tree Logic (ACTL). The tool was incorporated into a verification environment, JACK, which was able to cover several aspects of the formal software development process. The Model Checker included in JACK allowed the satisfiability of ACTL formulae on the model representing the system to be verified. Nelken and Francez (1996) presented another work that also aims to translate NL requirements into ACTL formulae. Their translation method was implemented in a tool, *SpecTran 1.0*. Holt et al. (1999) developed a system that allowed Formal Verification of digital circuits using specifications developed in English. Their system supports the translation of NL sentences into CTL (BAIER; KATOEN, 2008) formulae. However, their implementation dealt with only a restricted subset of English.

Koh and Seong (2009) addressed safety analysis of NL software requirements specifications by using Model Checking. The authors argued that the conventional method for applying Fault Tree Analysis for safety evaluation has problems in terms of correctness and efficiency, since the fault tree generated from NL specifications may contain defects while the manual work of safety verification is time-consuming. Therefore, they proposed an approach where a fault tree is generated from the Symbolic Model Verifier (SMV) model but not from NL specifications, and the safety properties, formalized in CTL, are automatically verified by using the Model Checker SMV (McMILLAN, 1993).

Software Cost Reduction (SCR) is a formal method based on a tabular notation. There is a set of software tools to analyze requirements specifications in the SCR notation such as a *specification editor*, a *consistency checker*, a *simulator*, and the capability to work with traditional Model Checkers (HEITMEYER et al., 1998; BHARADWAJ; HEITMEYER, 1999; GARGANTINI; HEITMEYER, 1999).

Bharadwaj and Heitmeyer (1999) described how a complete requirements specification created in SCR can be model checked. Their approach used Model Checking to analyze properties of complete SCR specifications with variables ranging over many data types. They described two methods for producing abstraction from requirements specifications, and showed how SCR specifications can be translated into the language of either SMV (McMILLAN, 1993) or SPIN (HOLZMANN, 2003) Model Checkers. According to their experiments, explicit state Model Checkers, such as SPIN, were computationally less expensive than symbolic Model Checkers, such as SMV, for detection of defects. On the other hand, when a property was violated, SMV was guaranteed to produce the shortest possible counterexample which was an advantage over SPIN.

The use of formal methods within partial specifications in the context of Independent Verification and Validation of the requirements for Fault Detection, Isolation and Recovery on the International Space Station (ISS) was presented by Easterbrook and Callahan (1998). They used a combination of AND/OR tables (HEIMDAHL; LEVESON, 1996), the SCR method (BHARADWAJ; HEITMEYER, 1999), and the SPIN (HOLZMANN, 2003) Model Checker. By manually restating the requirements in AND/OR tables, they removed ambiguity and their approach found inconsistencies by using the SCR *consistency checker* and SPIN. The problems they encountered in applying formal methods to safety-critical complex specifications were: the process of translating informal (NL) specifications into a formal notation was error-prone, it was hard to guarantee fidelity between informal (NL) and formal specifications, and it was hard to manage consistency between partial specifications developed in different notations.

Heimdahl and Leveson (1996) described an automated approach to analyzing a Requirements State Machine Language (RSML) specification for some aspects of completeness (a set of criteria related to robustness) and consistency (conflicting requirements and non-determinism). RSML is a formal state-based requirements specification language which includes several features of Statecharts (HAREL, 1987). One difference is that Statecharts use predicate calculus to describe the guarding conditions on the transitions and RSML uses AND/OR Tables, a tabular representation of DNF. Their method was applied to the specification of an avionics system, the *Traffic alert and Collision Avoidance System II* (TCAS II), and they claimed that state-space explosion problems were eliminated by applying the analysis at a high level of abstraction: instead of generating a reachability graph for analysis, the analysis was performed directly on the model.

Despite the interesting results, Heimdahl and Leveson's approach has issues due to the presence of spurious (false positive) reports of defects. Basically, two features of the predicates in RSML contributed to this problem: the use of simple arithmetic and the use of mathematical functions. The number of spurious reports of defects increased dramatically when the number of predicates including these features increased. In addition, since they did not explore the reachable state space, the states that exhibit inconsistency or incompleteness may not be reachable, and their decision to consider only transitions with the same trigger may limit the accuracy of their method (CHAN et al., 1998).

Chan et al. (1998) translated a portion of a preliminary version of the TCAS II requirements specification from RSML into the language of the SMV (McMILLAN, 1993) Model Checker. Their goal was more related to realize about the effectiveness of Model Checking on software systems, so that applying the technology was more important than the individual results. They defined rules for declaring and initializing SMV variables from RSML models, for translating deterministic and a class of non-deterministic RSML transitions. Their approach was able to detect some inconsistency defects but there were no evident results that assure it can detect incompleteness defects in the specifications (although the authors claimed it). Besides, their translation mechanism involved significant manual effort, such as modifications of the SMV Model Checker and the use of special-purpose macro processors (BHARADWAJ; HEITMEYER, 1999).

Pontes et al. (2009b) presented two approaches to refine software requirements specifications: Model Checking using timed automata and the UPPAAL Model Checker (BEHRMANN et al., 2004), and the CoFI testing methodology (AMBRÓSIO et al., 2007). They used a very simple case study, an automatic coffee machine, to show the usefulness of their proposal and they claimed that Model Checking contributed to detect defects like ambiguity, inconsistency, and incompleteness in informal (NL) requirements. Moreover, CoFI could identify a larger number of incompleteness defects.

The work of Pontes et al. (2009b) is interesting, however, it has some issues. First, the authors stated that there are no guidelines to translate informal requirements into CTL formulae. This is not quite right because Dwyer et al. (1999) proposed directives for the translation from informal (NL) requirements into CTL formulae just as Konrad and Cheng (2005) proposed the translation into Timed Computation Tree Logic (TCTL) (BAIER; KATOEN, 2008).

Second, they stated that Model Checking contributed to detect ambiguities when CTL formulae associated with requirements were defined. This assertion that Model Checking contributes to the detection of ambiguity in informal (NL) requirements is also shared by other authors (BAIER; KATOEN, 2008). But, it was not really Model Checking that detected ambiguities. In order to state that it is because of Model Checking that ambiguities are detected, it is necessary that properties are formalized, a model is generated and the checking of the model against the properties derives results that somehow indicate that there are ambiguities in the informal requirements. Without checking the model against the properties it is not possible to assert that Model Checking is really applied, and that it actually helps to detect ambiguities. When attempting to generate properties from informal requirements, the professional (human being) is who actually contributes and detects ambiguities. Whatever the type of ambiguity (lexical, syntactic) present in NL requirements specifications, its automated detection is a complex process and requires computational linguistics techniques. For example, a parser can help to detect syntactic ambiguity by deriving two parse trees from the same NL requirement. Other approaches, such as Software Reading Techniques (BASILI et al., 1996), can be used but in such proposals the use of Model Checking does not seem appropriate.

Still on the work of Pontes et al. (2009b), the incompleteness problem in informal (NL) requirements was addressed by deriving models and manually/visually inspecting them. As a result, for the detection of incompleteness, Model Checking was not really applied because the verification of the model against properties was not made. In some sense, the use of Model Checking and the CoFI methodology resembles techniques already proposed in the literature: Software Reading Techniques such as PBR (BASILI et al., 1996). It seems that the only situation where Model Checking was coherently applied was limited to inconsistency detection but even so it was not very clear which incosistencies were detected by the application of Model Checking and which ones were detected by visual inspection of the models. In another work from the same authors (PONTES et al., 2009a), the problems quoted above are also present with respect to the application of Model Checking.

More recently, Pontes and his colleagues presented another work aiming at using Model Checking to verify software embedded in on-board data handling computers of satellites (PONTES et al., 2010). The case study was based on six subtypes of two services of the European Cooperation for Space Standardization (ECSS) - Packet Utilization Standard (PUS) (ECSS, 2003). Although the case study is more interesting than that reported in the previous studies (PONTES et al., 2009b; PONTES et al., 2009a), the authors continue stating that there are no directives to transform informal requirements into CTL properties. Thus, they do not provide guidelines to formalize the properties.

Yu et al. (2008) proposed an approach to perform completeness and consistency analysis on requirements, and another approach to correcting the inconsistencies identified. A formal scenario model based on first order logic was used to represent requirements so that scenario elements of condition guards, events and actions could be automatically extracted (a prototype tool was developed). Condition guards associated with the same event were constructed into a tree in order to perform completeness analysis and supplement missing requirements. Consistency analysis focused on three types of inconsistencies and it was accomplished according to the intra-relations among condition guards and inter-relations with actions. However, scalability seems to be a problem because their approach was applied to a very simple case study.

Gargantini and Heitmeyer (1999) proposed a method for generating test cases from an operational SCR requirements specification containing mixed variable types, i.e. integers, booleans, and enumerated types. Model Checking was used to generate test cases. Their method is based on two ideas. First, the Model Checker is used as an oracle to compute the expected results. Second, the Model Checker's ability to generate counterexamples is used to construct the test cases. A tool was developed to automatically translate the SCR specification into the language of either SMV (McMILLAN, 1993) or SPIN (HOLZMANN, 2003) Model Checkers[6], to execute the selected Model Checker, and to derive the test cases. There are also other studies that use Model Checking for test case generation (AMMANN et al., 1998; ENGELS et al., 1997).

2.4 Final remarks about this chapter

This chapter presented the theory and research related to this PhD thesis. The areas of knowledge associated with this work include, among others, Model-Based Testing, combinatorial designs, Formal Verification (Model Checking), NLP, and formal methods in general. Each of these research areas have much information, so that this chapter has tried to present only what is necessary for the proper understanding of this work. Also, emphasis was given to studies which will then be compared in the chapters concerning the development of the PhD thesis itself. Next chapter presents version 1.0 of the SOLIMVA methodology.

[6]The translation method was the same as described in Bharadwaj and Heitmeyer (1999).

3 THE SOLIMVA METHODOLOGY

This chapter presents version 1.0 of the SOLIMVA methodology which addresses the primary goal of this PhD thesis: generation of model-based system and acceptance test cases considering NL requirements deliverables. In this chapter when used without any explicit indication of the version number, the terms "SOLIMVA methodology" and "SOLIMVA" refer to version 1.0 of the SOLIMVA methodology. In SANTIAGO JÚNIOR and VIJAYKUMAR (2012) there is an abridged version of this chapter. As mentioned in Chapter 1, SOLIMVA is supported by a main tool, also called SOLIMVA, that makes it possible to automatically translate NL requirements into Statechart models. Once the Statecharts are created, the GTSC environment is used to generate Abstract Test Cases which are later translated into Executable Test Cases. However, in order to improve understandability, the main case study in which the SOLIMVA methodology and supporting tools (SOLIMVA tool and GTSC environment) were applied will be firstly described.

3.1 Main case study: software embedded in satellite payload

This case study is a space application software product, Software for the Payload Data Handling Computer (SWPDC), developed in the context of the *Qualidade do Software Embarcado em Aplicações Espaciais* (QSEE - Quality of Space Application Embedded Software) research project (SANTIAGO et al., 2007). QSEE was an experience in outsourcing the development of software embedded in satellite payload. INPE was the customer and there were two SWPDC's suppliers: INPE itself and a Brazilian software company. The QSEE research project used ECSS standards (ECSS, 2008) in order to guide the relationship between customer and supplier. The main goals of QSEE were: (i) to transfer to Brazilian software industry INPE's knowledge in software development for space applications, particularly Verification and Validation methods and techniques applied to software embedded in scientific instruments of satellites and in balloon applications; (ii) to update the software development methodology for scientific satellites and balloon applications under development at INPE; iii) to create a methodology so that INPE can accept software developed by private software companies.

Figure 3.1 shows the functional architecture defined for QSEE project. Note the following computing units in the architecture: On-Board Data Handling (OBDH) Computer, Payload Data Handling Computer (PDC), Event Pre-Processors (EPPs; EPP H1 and EPP H2), and Ionospheric Plasma Experiments (IONEX) Computer. OBDH is the satellite platform computer in charge of processing platform and

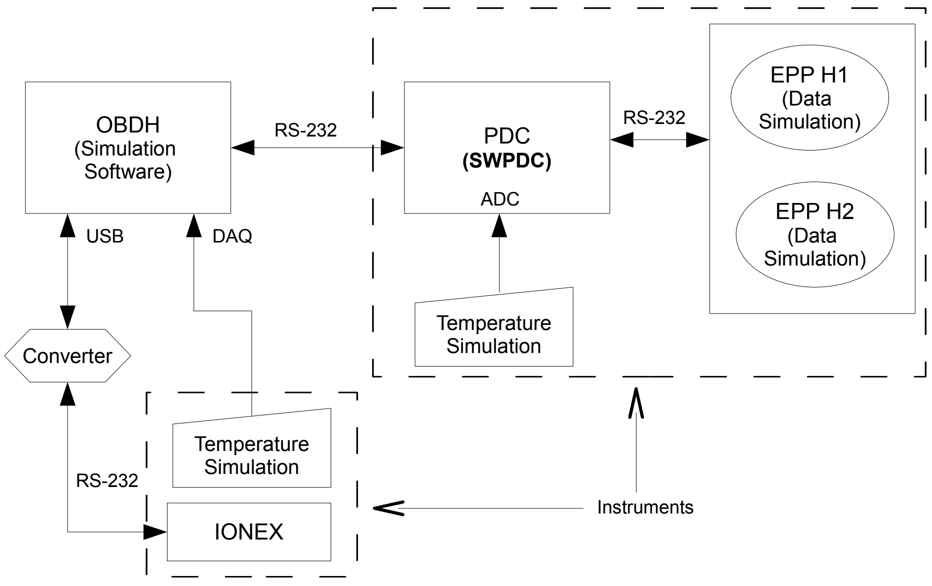


Figure 3.1 - Functional architecture defined for QSEE project. Caption: ADC = Analog-to-Digital Converter; DAQ = Data Acquisition Board; RS-232 = Recommended Standard 232; USB = Universal Serial Bus
SOURCE: Adapted from Santiago et al. (2007)

payload information and formatting/generating data to be transmitted to Ground Stations. The payload is composed of two scientific instruments (note the dashed rectangles). However, for the purpose of this case study, the main instrument is the one in which PDC exists, because SWPDC is embedded into PDC. The main goal of PDC is to obtain scientific data from EPPs and to transmit them to the OBDH. EPPs are front-end processors in charge of fast data processing of X-ray camera signals.

Essentially, this system employs a two-level primary/secondary communication model. In the first level, OBDH is the primary unit, PDC and IONEX are the secondary units. In the second level, PDC is the primary unit and EPPs (EPP H1 and EPP H2) are the secondary units. Communication protocols were specified to make the interface among the several computing units within the architecture.

The main functions of the SWPDC software product are: (i) interaction with EPPs in order to collect Scientific, Diagnosis and Test Data; (ii) data formatting; (iii) memory management to store data temporarily before transmission to the OBDH; (iv) implementation of flow control mechanisms; (v) Housekeeping Data generation; (vi) implementation of complex fault tolerance mechanisms; and (vii) loading of new programs on the fly (SANTIAGO et al., 2007). This case study has, therefore, almost all the functions of data handler computers for space applications and thus, the characteristics of SWPDC are representative of an important class of software in space domain.

Figure 3.2 shows QSEE's software development lifecycle processes (rectangles), formal technical reviews (circles), and main deliverables (artifacts). It is worth mentioning that the Independent Verification and Validation process was conducted by an independent team at INPE and started since the beginning of the software development lifecycle. Formal technical reviews were the main interaction points between customer and supplier. Each formal technical review had associated a set of deliverables which were assessed by reviewers and discussed during formal technical reviews meetings in order to improve their overall quality.

The most important deliverables evaluated within each formal technical review are below each review's circle in Figure 3.2. For example, within the Preliminary Design Review (PDR), PDC-OBDH Communication Protocol Specification (POCP), PDC-EPPs Communication Protocol Specification (PECP), Software Development Plan (SwDevPlan), Software Requirements Specification (SRS), and Independent Verification and Validation Plan (IVVPlan) were the main input and output deliverables. Suppliers provided the deliverables marked with an asterisk (*), e.g. SRS, and the customer was in charge of the others with no asterisk, e.g. POCP. Furthermore, deliverables in boldface and underlined mean that their output version within the review is considered their final version. Hence, Requirements Baseline (RB) was developed by the customer, it was input and output of the System Requirements Review (SRR), and RB's output version within SRR was frozen. The IVVPlan was also developed by the customer, and it was assessed within SRR, PDR, and Critical Design Review (CDR). IVVPlan's output version within CDR was frozen. SRSs were responsibility of the suppliers and they were evaluated within PDR and Detailed Design Review (DDR). SRSs' output versions within DDR were frozen.

In the next two sections version 1.0 of the SOLIMVA methodology, the main tool that supports the methodology, i.e. the SOLIMVA tool, and, very briefly, the GTSC environment will be described and, whenever necessary, the SWPDC case study will be used to demonstrate the execution of the activities. In order to apply version 1.0 of the SOLIMVA methodology, four deliverables were consulted: RB, SRS, POCP, and PECP.

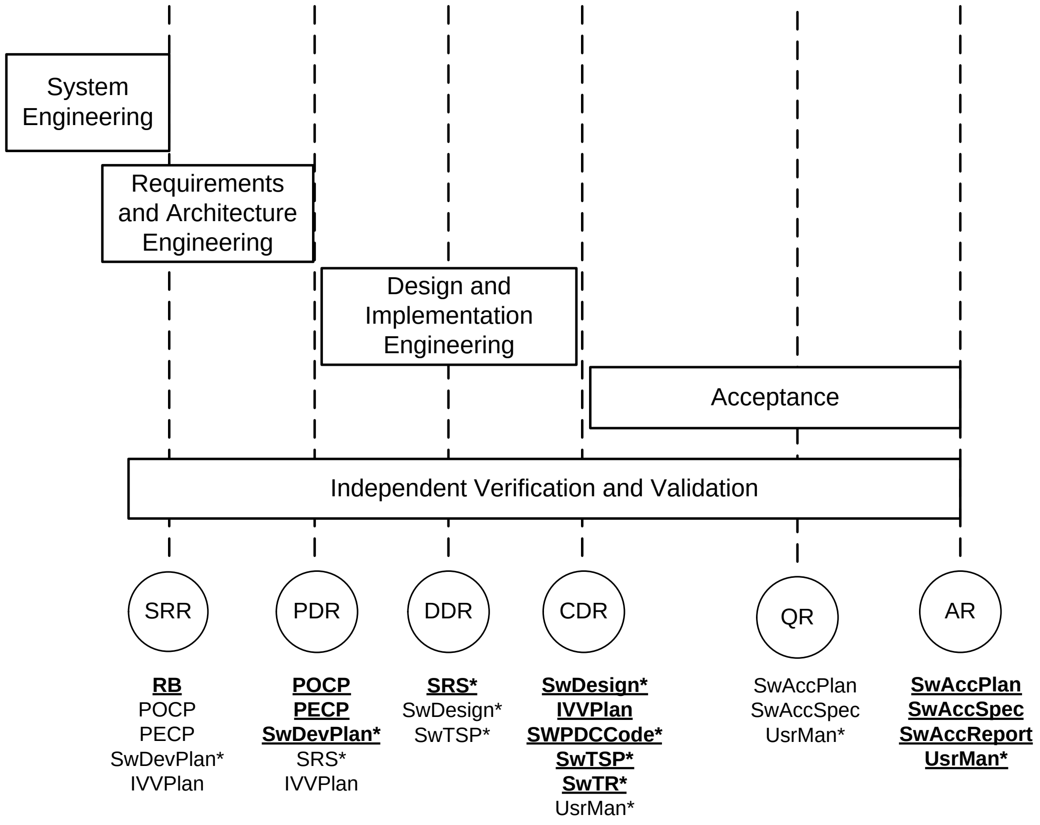


Figure 3.2 - QSEE project: software development lifecycle processes, formal technical reviews, and main deliverables. Caption: SRR = System Requirements Review; PDR = Preliminary Design Review; DDR = Detailed Design Review; CDR = Critical Design Review; QR = Qualification Review; AR = Acceptance Review; RB = Requirements Baseline; POCP = PDC-OBDH Communication Protocol Specification; PECP = PDC-EPPs Communication Protocol Specification; SwDevPlan = Software Development Plan; IVVPlan = Independent Verification and Validation Plan; SRS = Software Requirements Specification; SwDesign = Software Design Document; SwTSP = Software Test Specification and Plan; SwTR = Software Test Report; SWPDCCode = SWPDC's Source Code; UsrMan = User Manual; SwAccPlan = Software Acceptance Test Plan; SwAccSpec = Software Acceptance Test Specification; SwAccReport = Software Acceptance Test Report
SOURCE: Adapted from Santiago et al. (2007)

3.2 Description of version 1.0 of the SOLIMVA methodology

Version 1.0 of the SOLIMVA methodology is illustrated in the activity diagram of Figure 3.3. The first activity is the definition of a Dictionary by the user/test designer. The Dictionary defines the application domain and it is considered as a quintuple $\langle N, R, S_{TM}.C, S_{TM}.F, S_{TM}.Y \rangle$, where:

- N is a set of Names defining mainly the names of states of the model;
- $R : R.I_E \rightarrow R.O_E$. R is a function[1] from $R.I_E$ (input event set) to $R.O_E$ (output event set) that represents the Reactiveness of the system;
- S_{TM} is the *Semantic Translation Model*. S_{TM} is composed of two sets and a function. One set characterizes specific control behaviors, $S_{TM}.C$. The other set characterizes the occurrence of self transitions within the model, $S_{TM}.F$; and $S_{TM}.Y : Y.I_P \rightarrow Y.O_P$. $S_{TM}.Y$ is a function from $Y.I_P$ (input pattern set) to $Y.O_P$ (output pattern set) that is related to hierarchy (depth) in the Statechart model.

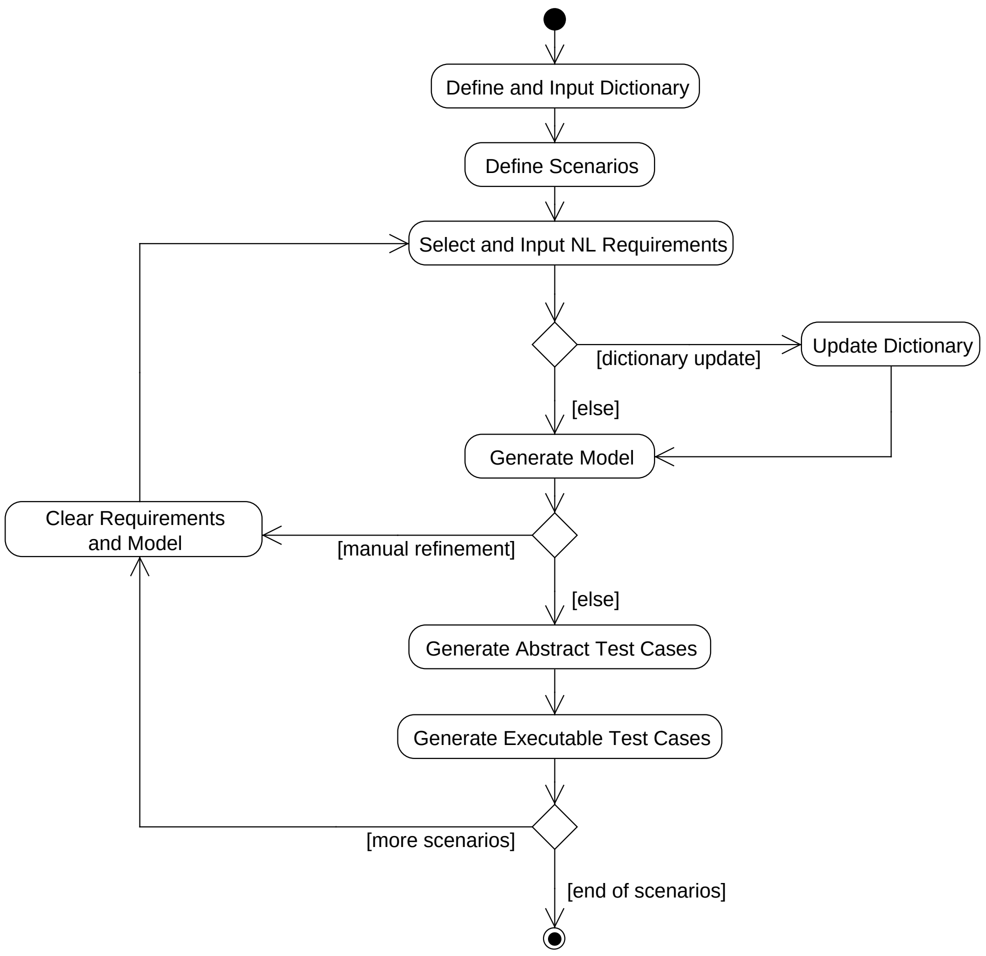

Define and Input Dictionary
Define Scenarios
Select and Input NL Requirements
[dictionary update]
Update Dictionary
[else]
Generate Model
Clear Requirements and Model
[manual refinement]
[else]
Generate Abstract Test Cases
Generate Executable Test Cases
[more scenarios]
[end of scenarios]

Figure 3.3 - Version 1.0 of the SOLIMVA methodology

[1]In this work, a function F is considered as a set of ordered pairs (x, y), where x is an element of the domain of F, and y is an element of the codomain of F.

The set N, the functions R and $S_{TM}.Y$ are defined by the user. The sets $S_{TM}.C$ and $S_{TM}.F$ are already defined within the SOLIMVA tool which supports the methodology, although the user can change if needed. Users enter data via Graphical User Interface and using NL and, moreover, they are not required to have any knowledge in formal methods and their notations to define the application domain. It is worth mentioning that the Reactiveness feature of the Dictionary comes into picture because reactive systems are the main targets of SOLIMVA.

Consider the SWPDC case study described in Section 3.1. The Name (N) set of the Dictionary will be composed mainly by relevant words or set of words that map to important entities of the application domain. These include the first-level primary computing unit (OBDH), the computer in which SWPDC will be embedded (PDC) and the operation modes of such computer, SWPDC itself, and so on. Hence, N can be composed of:

> N={PDC, SWPDC, OBDH, Initiation Operation Mode, Safety Operation Mode, ...}.

The Reactiveness (R) function is basically a mapping between the commands (the domain of R, i.e. $R.I_E$) and responses (the codomain of R, i.e. $R.O_E$) defined in the PDC-OBDH Communication Protocol Specification and possibly in the PDC-EPPs Communication Protocol Specification as well. For instance, VERIFY PDC's OPERATION MODE (VER-OP-MODE) is a command (an element of $R.I_E$) that the OBDH sends to PDC in order to know which is its current operation mode. The response that PDC sends back to the OBDH is the INFORMATION REGARDING THE PDC's OPERATION MODE (INFO-OP-MODE), an element of $R.O_E$. The OBDH may CHANGE PDC's OPERATION MODE TO NOMINAL (CH-OP-MODE-NOMINAL) or CHANGE PDC's OPERATION MODE TO SAFETY (CH-OP-MODE-SAFETY). Assuming there is no problem during the transmission of the command to the PDC, in both cases the PDC responds with a positive acknowledgement, i.e. COMMAND CORRECTLY RECEIVED (CMD-REC). Hence, $R.I_E$, $R.O_E$, and R can be composed of:

- $R.I_E$ = {VER-OP-MODE, CH-OP-MODE-NOMINAL, CH-OP-MODE-SAFETY, ...},
- $R.O_E$ = {INFO-OP-MODE, CMD-REC, ...},

- R = {(VER-OP-MODE, INFO-OP-MODE), (CH-OP-MODE-NOMINAL, CMD-REC), (CH-OP-MODE-SAFETY, CMD-REC), ...}.

The *Semantic Translation Model* of the Dictionary will be detailed in subsequent sections. After the definition of the Dictionary, scenarios are identified. A scenario is defined as follows.

Definition 3.1. A scenario is defined as an interaction between a user and the IUT. Associated with each scenario there is a set of requirements which characterize such an interaction.

Figure 3.4 shows the *Define Scenarios* activity of the SOLIMVA methodology in form of an activity diagram while Figure 3.5 shows the *Define Scenarios* activity in form of a procedure. In the sequence, the explanations about the *Define Scenarios* activity are given taking into account the form of procedure that describes the activity (Figure 3.5). The first task, which is optional, serves to obtain the basic elements that enable interaction with the IUT. In terms of embedded reactive systems, these basic elements can be protocol data units, commands, etc. that characterize the interface between two or more computing systems. Hence, a set of very **simple scenarios** are determined aiming at observing the correct implementation of these elements by the IUT.

This first task is optional because the test designer might simply rely on test cases applied on previous phases of the software development lifecycle, e.g. unit testing, and consider that these basic elements are correctly implemented. In the SWPDC case study, a simple scenario is to switch the PDC on and send the VER-OP-MODE command to evaluate whether SWPDC has correctly implemented the reception and processing of this command.

In SOLIMVA, combinatorial designs (MATHUR, 2008) are used to help to identify scenarios. The basic idea is to define factors (input variables) and levels (values assignable to a factor) and to use a combinatorial designs algorithm to determine the set of levels, one for each factor, known as a factor combination or run (Section 2.1.1.3). Among the combinatorial designs techniques available, the SOLIMVA methodology adopted the Mixed-Level Covering Array. The algorithm used is the *In-Parameter-Order* (IPO) (LEI; TAI, 1998), a procedure that can generate Mixed-Level Covering Arrays. At present, the SOLIMVA methodology uses version 2.1 of *TConfig* (UNIVERSITY OF OTTAWA, 2008), an open source tool which has implemented the

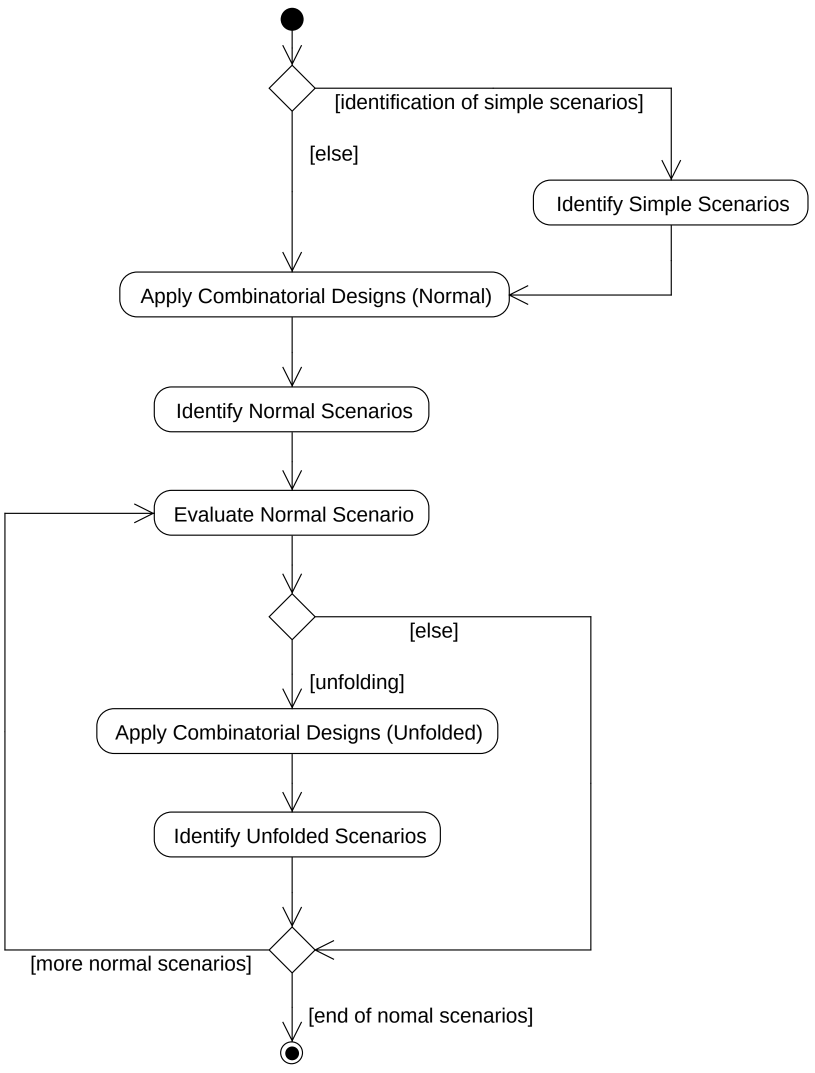

[identification of simple scenarios]
[else]
Identify Simple Scenarios
Apply Combinatorial Designs (Normal)
Identify Normal Scenarios
Evaluate Normal Scenario
[else]
[unfolding]
Apply Combinatorial Designs (Unfolded)
Identify Unfolded Scenarios
[more normal scenarios]
[end of nomal scenarios]

Figure 3.4 - The *Define Scenarios* activity of the SOLIMVA methodology: activity diagram

IPO procedure.[2]

Let f_i, $1 \leq i \leq m$, 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. Hence, a factor combination number X (*fc X*) of the generated Mixed-Level Covering Array could be *fc X*= $\{l_{11}, l_{23}, \ldots, l_{m1}\}$ which means the first level of factor 1, the third level of factor 2, and so on until the first level of factor m. Thus, a factor combination derived by the

[2]The IPO procedure was originally conceived for 2-way (pairwise) testing. Version 2.1 of the *TConfig* tool can generate from 2-way to 6-way Mixed-Level Covering Arrays. However, it is not clear if the tool has implemented the *In-Parameter-Order-General* (IPOG) (LEI et al., 2007) algorithm which deals with general t-way testing or else if another approach was adopted.

```
input  : NL requirements deliverables
output: scenarios for model-based test case generation
1  if identification of simple scenarios is needed then
2      Identify simple scenarios based on the core elements that enable interaction with the IUT;
3  end
4  Identify $factors$ and $levels$;
5  Define $strength = \#factors - 1$;
6  Run the combinatorial designs algorithm, using the $strength$ defined in the previous step;
7  Identify normal scenarios. Consider the interpretation of at least $(factors - 1)$ out of
   $factors$ $levels$ of each factor combination when identifying normal scenarios. Each factor
   combination will drive the identification of a scenario;
8  foreach normal scenario do
9      if unfolding is needed then
10         Identify new $factors$ and $levels$;
11         Define a priority $factor$;
12         Define $strength = \#factors - 1$;
13         Run the combinatorial designs algorithm, using the $strength$ defined in the previous
           step;
14         Identify scenarios obtained by unfolding the normal scenario. At least $(factors - 1)$
           out of $factors$ $levels$ of each factor combination shall be accounted for when
           identifying unfolded scenarios. The number of unfolded scenarios will be the number
           of $levels$ of the priority $factor$;
15     end
16 end
```

Figure 3.5 - The *Define Scenarios* activity of the SOLIMVA methodology: procedure

combinatorial designs algorithm is interpreted by the test designer in order to define a scenario (the interaction between the user and the IUT). In this work the expression “scenario X”, where X is given by the tool that has implemented the combinatorial designs algorithm, means that this is the scenario defined by the test designer due to the interpretation of *fc X*. This perspective applies only to **normal scenarios** and **unfolded scenarios** (further explanation of these scenarios in sequence) which are those generated with the aid of the combinatorial designs algorithm. In other words, there is no *fc X* to interpret to define simple scenarios because these are not derived with the help of the tool that has implemented the combinatorial designs algorithm.

As seen from lines 4 to 7 in Figure 3.5, scenarios identified by the interpretation of the factor combinations due to the first use of combinatorial designs are called **normal scenarios**. Besides, note that $strength = \#factors - 1$ with the objective of achieving the maximum number of factor combinations without using an exhaustive (all to all) approach (#factors = total number of factors).

A scenario might have more that one test objective associated. However, these test objectives should not be too disparate. This recommendation might lead the user to neglect one level on interpreting the factor combination in order to identify a scenario. The explanation for this fact is that the characteristics of the factors can be considerably different so that if the test designer defines a scenario based on all levels of a factor combination, it is possible that such a scenario has several unrelated test objectives which is not a good approach, resulting in a bad strategy in terms of test objectives. For instance, in a web application, interpretation of a factor combination may generate a scenario in which it is necessary to verify whether a web service is correctly implemented, and some security requirements are satisfied, and information retrieval from a data base due to a specific type of request is consistent. There are many different and unrelated test objectives in this case. Thus, it is more interesting to disregard the interpretation of a level to decrease the amount of test objectives. In such situations, "-" is used to mean "do not consider any level of this factor for this particular scenario".

A simplified choice of factors and levels for the SWPDC case study is shown in Table 3.1. The explanation for the factors and levels follows:

a) Cmd. This factor relates to the commands defined in the PDC-OBDH Communication Protocol Specification. These commands were grouped into levels considering processing activities to acquire and transmit data (DtAcqTx), and handling of hardware and software parameters (HwSwHnd);

b) OpMode. This factor relates to PDC's operation modes. In this simplified example, only the Nominal Operation Mode (Nom) was considered;

c) Services. This factor relates to the services supported by SWPDC. In this example, only the services related to acquisition, formatting, and transmission of Scientific Data (Sci), and generation, formatting, and transmission of Housekeeping Data (Hk) were taken into account.

Note that each factor has a level *Inv*, which stands for invalid value. The SOLIMVA methodology strongly recommends that in each factor there is such a level to address Robustness testing which is particularly useful for embedded critical software where, for instance, it is possible to observe the behavior of the IUT under non specified test input data. Running the combinatorial designs algorithm with $strength =$

Table 3.1 - Simplified choice of factors and levels for the SWPDC case study

Factors	**Levels**		
Cmd	DtAcqTx	HwSwHnd	Inv
OpMode	Nom	Inv	
Services	Sci	Hk	Inv

$\#factors - 1 = 2$, nine factor combinations are generated. Table 3.2 shows the normal scenarios based on the interpretation of each factor combination.

The first remark about the generated scenarios is that when a level is not present in a factor combination, this does not necessarily imply that it will not be somehow related to the scenario derived from the interpretation of such factor combination. It depends on the kind of factor. For instance, in Table 3.2, normal scenario 1 has level DtAcqTx due to the command factor (Cmd). It does not mean that the selection of NL requirements that characterize normal scenario 1 will be such that no requirement shall be related to commands to handle hardware and software parameters (HwSwHnd). Indeed, it is very likely that HwSwHnd commands should be sent to PDC in order to drive the SWPDC to the appropriate state so that the test objective of normal scenario 1 can be achieved. For example, in order to acquire, format, and transmit Scientific Data in the Nominal Operation Mode, first EPP H1 and EPP H2 must both be turned on. But, the commands to switch them on are HwSwHnd commands. Hence, they need to be sent to PDC prior to data acquisition. However, for normal scenario 1, the main contribution of the command factor is related to data acquisition and transmission (DtAcqTx).

In normal scenarios 2, 3 and 4, one level was not accounted for ("-") when interpreting the factor combinations. This level was precisely *Inv* which was addressed in other scenarios (5, 6, 7, 9) whose main goals were related to Robustness testing. This stresses the important recommendation that the methodology provides to incorporate Robustness testing covering several different situations. Scenarios where robustness is the main test objective can be mapped to "unhappy cases" in use case modeling.

The *Inv* level may be translated into many different test input data. This is a characteristic of combinatorial designs testing where each factor combination may drive one or more test input data. For instance, in normal scenario 5, the test designer may select several different invalid commands by looking at the PDC-OBDH Communication Protocol Specification and choosing "commands" that are

Table 3.2 - Normal scenarios due to the factors and levels of Table 3.1. Caption: #Scn = Scenario number

Factor Combination	#Scn	Scenario (Interpretation)
{DtAcqTx, Nom, Sci}	1	Acquire, format, and transmit Scientific Data in the Nominal Operation Mode
{DtAcqTx, -, Hk}	2	Generate, format, and transmit Housekeeping Data in the Nominal Operation Mode
{HwSwHnd, Nom, -}	3	Verify the correct implementation of commands related to hardware parameters manipulation in the Nominal Operation Mode
{HwSwHnd, -, Sci}	4	Verify the correct implementation of commands related to hardware parameters manipulation during acquisition, formatting, and transmition of Scientific Data
{Inv, Nom, Hk}	5	Verify the behavior of SWPDC when receiving commands with inconsistent values during transmission of Housekeeping Data in the Nominal Operation Mode (Robustness testing)
{Inv, Inv, Inv}	6	Verify the behavior of SWPDC when receiving commands with inconsistent values, when trying to change PDC's operation mode to an unspecified operation mode, and when asking SWPDC to provide services not defined in the Software Requirements Specification (Robustness testing)
{Inv, Nom, Sci}	7	Verify the behavior of SWPDC when receiving commands with inconsistent values during acquisition, formatting, and transmission of Scientific Data in the Nominal Operation Mode (Robustness testing)
{HwSwHnd, Nom, Hk}	8	Verify the correct implementation of commands related to software parameters manipulation, and generate, format, and transmit Housekeeping Data in the Nominal Operation Mode
{DtAcqTx, Nom, Inv}	9	Verify the behavior of SWPDC when asking SWPDC to provide services not defined in the Software Requirements Specification during data acquisition, generation, and transmission (Scientific or Housekeeping Data) in the Nominal Operation Mode (Robustness testing)

not specified to be sent to PDC. To do that, other traditional black box testing techniques like boundary-value analysis might be applied to choose such invalid values. Hence, normal scenario 5 may have as many as needed invalid commands the test designer wishes, addressing the robustness of SWPDC. Another approach if the contribution of the Cmd factor is *Inv* within a factor combination is to address situations where a command is not entirely received by PDC due to problems in the physical transmission medium.

The fact that the user can neglect one level in interpreting the factor combination to derive a scenario does not mean that the entire test suite, considering all test cases derived according to all Statechart models, will be incomplete. Looking at Table 3.2, the set of derived scenarios cover all aspects of factors/levels of Table 3.1. In other words, within the scenarios defined in Table 3.2, it is possible to acquire, format, and transmit Scientific Data in the Nominal Operation Mode, to generate, format, and transmit Housekeeping Data in the Nominal Operation Mode, to verify the correct implementation of HwSwHnd commands, related to hardware and software parameters, and of DtAcqTx commands too. What matters is the expertise of the test designer in the application domain in order to interpret the factor combinations and to define scenarios that will generate, at the end, a test suite that is sufficiently complete.

However, it is possible that some normal scenarios have to be unfolded so that more factor combinations should be generated. The explanation for the need of such unfolding process lies in the fact that some levels may implicitly have specific values of variables (e.g. initial and final values of memory addresses) so that it is necessary to deal with situations which address the combination of such values. Not all normal scenarios need to have this demand. For those that need, new factors and levels are defined, as well as a priority factor. The total number of **unfolded scenarios** due to such process must be equal to the number of levels of the priority factor.

In case a normal scenario is unfolded, it is not necessary to accomplish that more than once because multiple unfoldings will make the methodology complex without substantial benefit in practical terms. Normal scenario 8 in Table 3.2 is the only one that needs to be unfolded into other scenarios. This is because the software parameters that are updated via commands have a default value, but also minimum and maximum values. Hence, it is interesting to replace normal scenario 8 with more specific scenarios addressing several situations regarding such values. Table 3.3 shows a simplified set of factors and levels for unfolding normal scenario 8.

Table 3.3 - Unfolding normal scenario 8: simplified choice of factors and levels

Factors	**Levels**			
HkTime	Min	Def	Max	Inv
IniPtr	Min	InRng	Max	Inv
SmpTime	Min	Def	Max	Inv

In Table 3.3, the priority factor is the parameter that defines the period (time interval) in which Housekeeping Data are generated (HkTime). In the PDC-OBDH Communication Protocol Specification, not only a default value (Def) is specified but also minimum (Min) and maximun (Max) values of this parameter. The same levels of HkTime apply to the sampling time of the analog input channels (SmpTime). The initial pointer (memory address) in which it is possible to load new executable code on the fly (IniPtr) has also minimum (Min) and maximum (Max) values, but it also allows initial addresses in range (InRng), i.e. between the minimum and maximum values.

Since the priority factor has four levels then four unfolded scenarios will be defined and will replace normal scenario 8. For instance, unfolded scenario 8.2, i.e. the second scenario unfolded from normal scenario 8, suggests the test designer to add requirements related to commands so that the following situations are covered:

- HkTime = Def, IniPtr = Min, SmpTime = Def;
- HkTime = Def, IniPtr = InRng, SmpTime = Min;
- HkTime = Def, IniPtr = Max, SmpTime = Inv;
- HkTime = Def, IniPtr = Inv, SmpTime = Max.

Note that the time to generate Housekeeping Data remains fixed in the default value within unfolded scenario 8.2. The other parameters must be udpated via commands with different values. Besides, Robustness testing is still in order due to the invalid values. In these cases, SWPDC can receive but it must not process any of the invalid commands.

After the previous steps, the user must select and input a set of NL requirements which together characterize a single scenario (simple, normal, unfolded). Then, the user must search these requirements in documents such as software requirements specifications. For normal and unfolded scenarios, the test designer must search for these NL requirements so that each level of a factor combination is addressed and thus a scenario can be characterized. For example, in the SWPDC case study, each interaction with the IUT requires that the PDC is energized and, after that, the activities of initializing the computing system are performed by SWPDC. Thus the following requirements, defined in the SWPDC's Software Requirements Specification, relate to the beginning of each scenario (SRS001, SRS002, and SRS003 are the identification of requirements. Appendix A has more details on this matter):

- SRS001 - The PDC shall be powered on by the Power Conditioning Unit.
- SRS002 - The PDC shall be in the Initiation Operation Mode after being powered on. The 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.
- SRS003 - If PDC does not present any irrecoverable problem, after the initiation process, the PDC shall automatically enter into the Safety Operation Mode.

Chapter 4 addresses the SWPDC case study in more detail, showing sets of chosen NL requirements that characterize scenarios.

The Dictionary does not necessarily have to be defined completely at once. The user can start defining and inputting part of the Dictionary at the beggining and, after choosing the NL requirements that characterize each scenario, the Dictionary can be updated according to new important words. This is the reason behind the optional activity *Update Dictionary*. Hence, the creation of the Dictionary is incremental and dependent on the selected set of NL requirements. This approach prevents the user to completely define the Dictionary at an early stage when applying SOLIMVA.

After that, the generation of the Statechart model follows. This activity will be discussed in detail in Section 3.3. After generating the model, the test designer may decide to manually refine it. Hence, he/she can accomplish this refinement in which the requirements of the scenario and the respective created model must be cleared (*Clear Requirements and Model* activity). The user can make such a refinement because he/she realized that the generated model can be improved and/or there is some kind of incoherence in the model probably due to mistakes when inserting the NL requirements (e.g. wrong sequence of requirements). The relevance of the manual refinement will be demonstrated with requirements SRS001, SRS002, and SRS003 presented above. If the test designer mistakenly switch the order of NL requirements SRS002 and SRS003, the resulting Statechart model due to this wrong input sequence of requirements will be naturally incorrect. Figure 3.6 shows a piece of the correct Statechart model while Figure 3.7 shows a piece of the incorrect Statechart model due to the wrong order of NL requirements.

After these steps, **Abstract Test Cases** are generated by using the GTSC environment (SANTIAGO et al., 2008b). GTSC allows test designers to model software

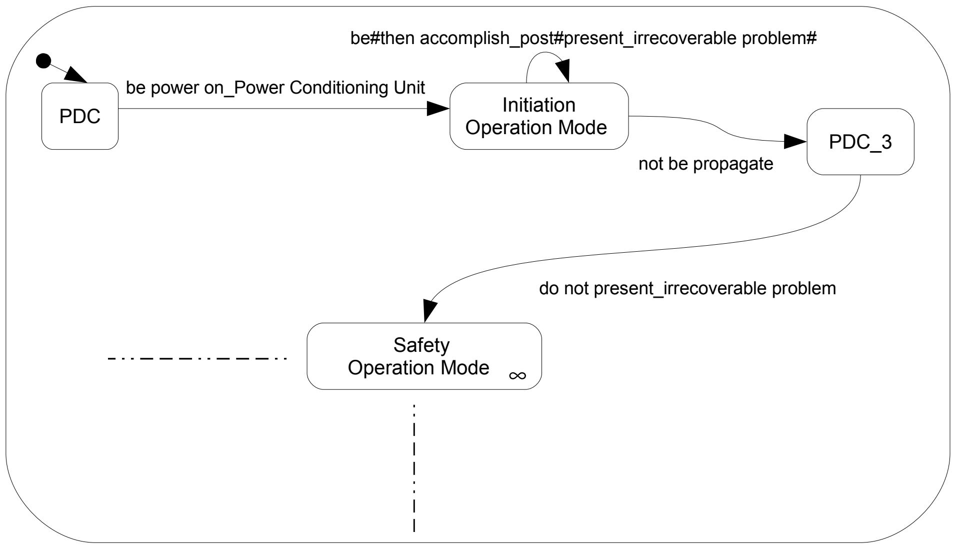

be#then accomplish_post#present_irrecoverable problem#
PDC
be power on_Power Conditioning Unit
Initiation Operation Mode
PDC_3
not be propagate
do not present_irrecoverable problem
Safety Operation Mode

Figure 3.6 - Piece of the entire Statechart model related to a scenario: correct model

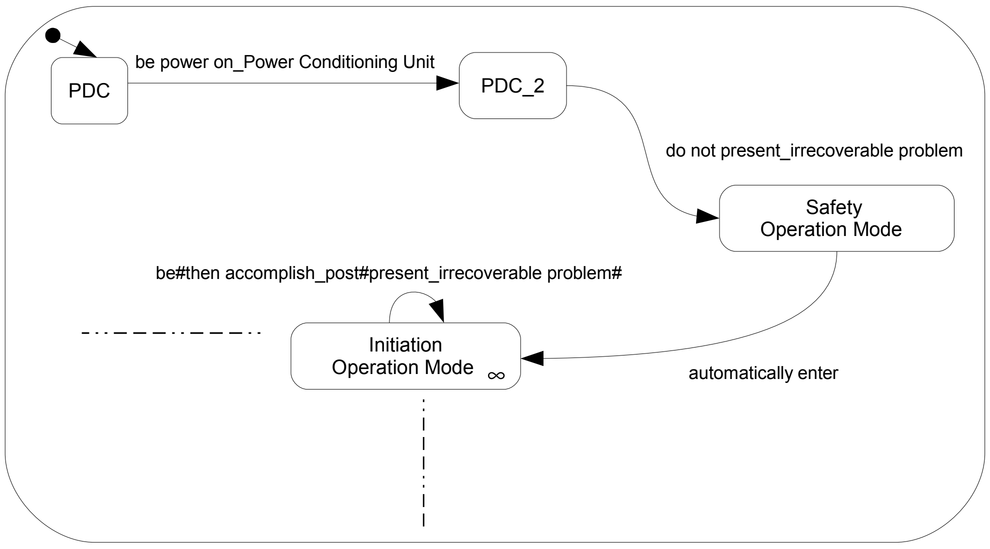

PDC
be power on_Power Conditioning Unit
PDC_2
do not present_irrecoverable problem
Safety Operation Mode
be#then accomplish_post#present_irrecoverable problem#
Initiation Operation Mode
automatically enter

Figure 3.7 - Piece of the entire Statechart model related to a scenario: incorrect model

behavior using Statecharts and/or FSMs in order to automatically generate test cases based on some test criteria for FSM and some for Statecharts. At present, GTSC has implemented DS, UIO (SIDHU; LEUNG, 1989) and H-switch cover (SOUZA, 2010) test criteria for FSM models, and four test criteria from SCCF (SOUZA, 2000), all-transitions, all-simple-paths, all-paths-k-C0-configuration, and all-paths-k-configurations, targeting Statechart models. In GTSC, the implementation of the all-paths-k-C0-configuration and all-paths-k-configurations test criteria considered $k = 2$. Thus, test criteria define the rules that drive test case generation in GTSC.

Figure 3.8 shows the architecture of GTSC where, in addition to the architectural elements themselves which are encapsulated by the larger rectangle, there are elements prepared/generated (Reachability Tree, Test Cases, ...) when using the GTSC. These external elements are not part of the architecture and they were placed in the figure only for better understanding.

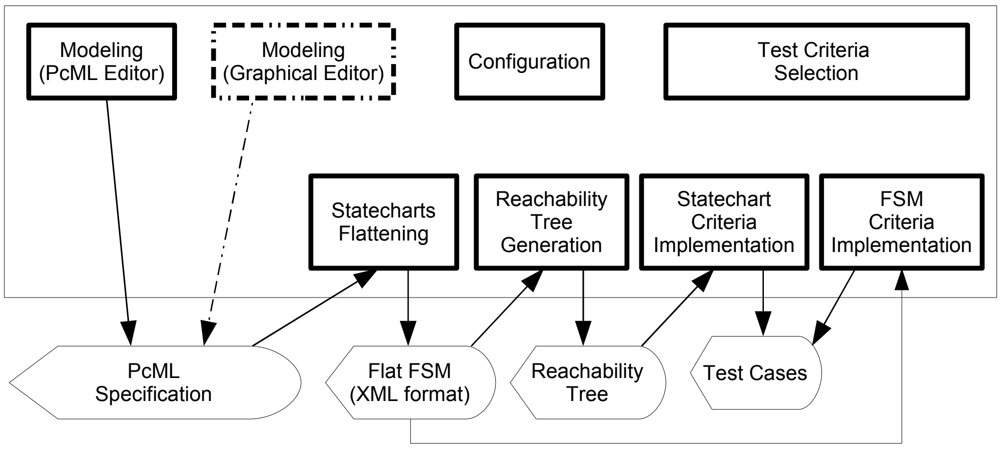


Figure 3.8 - The GTSC architecture. Caption: PcML = PerformCharts Markup Language, XML = Extensible Markup Language
SOURCE: Adapted from SANTIAGO JÚNIOR et al. (2010)

In order to use GTSC, the test designer translates the Statechart behavioral model into a *PerformCharts Markup Language* (PcML) specification (SANTIAGO et al., 2006). Based on a PcML specification, GTSC obtains a flat FSM. A flat FSM is a model where all hierarchical and orthogonal features of a Statechart model were removed. *PerformCharts* tool (VIJAYKUMAR et al., 2006), one of the components of the GTSC environment, is responsible for such a transformation. Each state of the resulting flat FSM is actually a configuration C_i (Section 2.1.1.2) of the Statechart model at a certain step of computation. This flat FSM is indeed the basis for test case generation.

A test designer may then follow two approaches to generate test cases. If an SCCF test criterion is selected to derive test cases, GTSC adapts the flat FSM to resemble a reachability tree (MASIERO et al., 1994). Thus, based on the selected test criterion of SCCF and on this tree, test cases are created. On the other hand, if an FSM test criterion is the option then GTSC simply generates test cases based on the flat FSM and on the selected test criterion.

The generated Statechart model is an abstract representation of the behavior of the IUT according to a specific scenario. Hence, the test cases derived from this model are a kind of functional tests on the same level of abstraction as the model. As will be shown in Chapter 4, the generated "test input data" and "expected results" of the test cases based on the Statechart model are usually pieces of NL sentences (particularly in the SWPDC case study, some commands/responses of Communication Protocol Specifications may also be present). Hence, the test cases generated by GTSC are Abstract Test Cases and thus they cannot be directly executed against the IUT due to the fact they are on the incorrect level of abstraction. Therefore, the test designer shall accomplish the translation from Abstract Test Cases into **Executable Test Cases** to enable the effective execution of test cases.

Having created the test cases (Executable Test Cases) for a single scenario, the test designer starts again selecting and inserting the NL requirements for the next scenario. But before doing this, he/she must clear the requirements and related model of the current scenario. This process must be repeated until there is no more scenario.

3.3 Model generation

The *Generate Model* activity in Figure 3.3 is composed of two sub-activities. When the user selects in the Graphical User Interface of the SOLIMVA tool the options related to these sub-activities, algorithms are executed to meet the goals. These sub-activities are described below.

3.3.1 Generation of tuples

The first task refers to the generation of *Behavior-Subject-Action-Object* (BSAO) 4-tuples. The BSAO 4-tuples are an extension of the concept of SAO triads used in the J-RAn tool (FANTECHI; SPINICCI, 2005).

In the SOLIMVA tool, the first extension is the inclusion of Behavior features in the SAO triad transforming into a BSAO 4-tuple. The reason behind this lies in the fact that words like "if" determine a particular behavior in the created model. For instance, finding an if-then-else situation in one or in several NL requirements (e.g. in a requirement: "If an echo is received ..."; in the same or in the next requirement:

"If an echo is not received ...") 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. Thus, if such a situation is detected within the NL requirements that characterize a scenario, an approach is followed which is similar to when a Control Flow Graph must be built from the source code and when a control structure *if* is found.

The second modification is related to Object identification. J-RAn presented a large number of extractions that were not detected (FANTECHI; SPINICCI, 2005), and one explanation for this fact is because J-RAn used a single link type ("O") of the *Link Grammar Parser* (SLEATOR; TEMPERLEY, 1993) to identify Objects. However, it is possible that there is no explicit Object generated by *Link Grammar* depending on the NL requirement. Consider the following requirement:

Users' data shall be updated on the server every 12 hours.

When using *Link Grammar*, there is no Object because none of the link types with respect to Object ("O", "OT", ...) appears in the parser output. An attempt to circumvent this problem was embedded in the SOLIMVA tool where an algorithm was developed and implemented to automatically identify the BSAO tuples. The algorithm makes use of version 3.0 of the *Stanford POS Tagger* (TOUTANOVA et al., 2003) in order to identify the lexical categories (POS) of each sentence of the NL requirements. Actually, after the user has selected and inserted the NL requirements that characterize a scenario, the SOLIMVA tool combines all such requirements into a file. This file is input to the *Stanford POS Tagger* which assigns the POS tag of each word.[3]

Figure 3.9 shows, in form of an activity diagram, the algorithm to automatically generate the BSAO tuples based on the NL requirements while Figure 3.10 shows the same algorithm in form of a procedure. In the sequence, the explanations about the algorithm are given taking into account the form of procedure that describes the algorithm (Figure 3.10). Besides the Dictionary (*dic*), the algorithm takes as input all the words (*allw* set) in the file that contains the set of NL requirements. In case of verbs and nouns, the algorithm obtains the lemma[4] and adds it in the *allw* set

[3]The *Stanford POS Tagger* adopts the *Penn Treebank POS tagset* (MARCUS et al., 1993).

[4]In linguistics, one definition of lemma is: the canonical form, dictionary form, or citation form of a set of words (headword). For instance, in English, "run", "runs", and "ran" are forms of the same lexeme, with "run" as the lemma.

instead of adding the word itself. For simplification, from now onwards when writing "word", this may refer to the word itself or to its lemma.

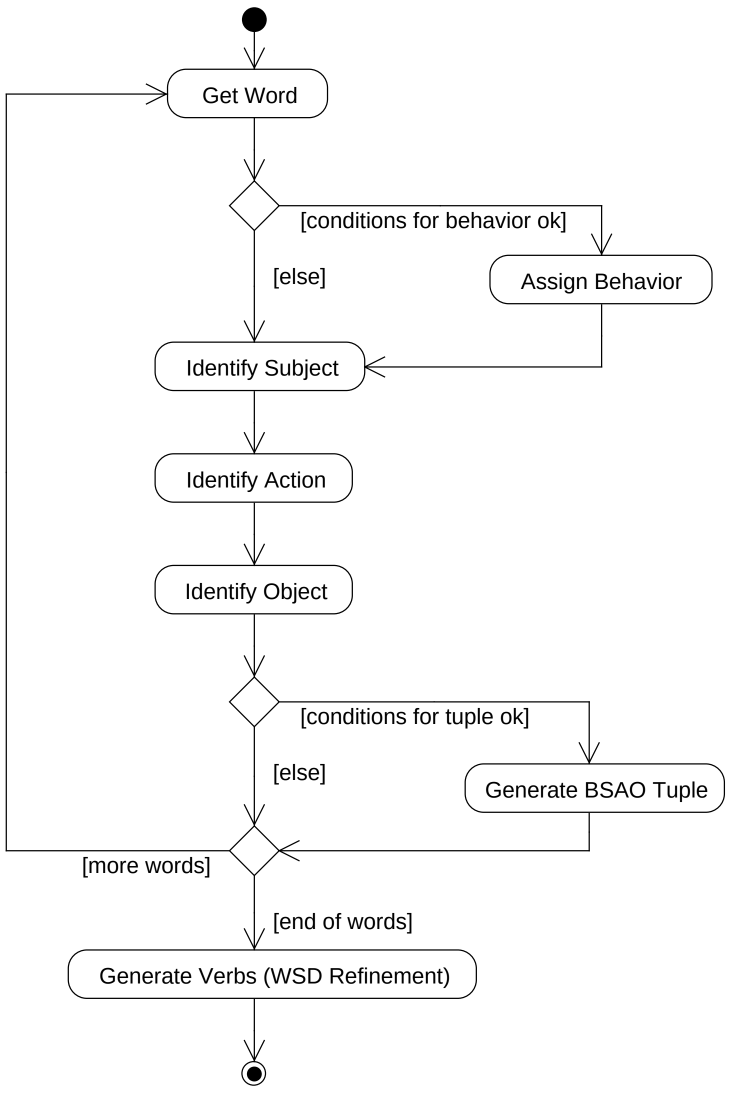


Figure 3.9 - Main algorithm for generating BSAO tuples: activity diagram

With respect to the notation in Figure 3.10, the *allw* set is in fact a set of ordered triples where each ordered triple is composed of: a word (or lemma) (upper index *lw*), a POS tag (upper index *tg*) obtained from the POS tagging algorithm implemented in the SOLIMVA tool, and a counter for the exact identification of the word (upper index *id*). Hence, *allw* is a set with the following ordered triples:

$$allw = \{(w_1^{lw}, w_1^{tg}, w_1^{id}), (w_2^{lw}, w_2^{tg}, w_2^{id}), \ldots, (w_h^{lw}, w_h^{tg}, w_h^{id})\}.$$

Thus, w_1^{lw} means the word of the first ordered triple, w_4^{tg} means the POS tag of the fourth ordered triple, and w_7^{id} represents the identification of the seventh ordered triple of the set. In the tuples (tup) set, b, s, a, o represent Behavior, Subject, Action, and Object, respectively. Similarly, tup is a set of ordered quadruples such as:

$$tup = \{(t_1^b, t_1^s, t_1^a, t_1^o), (t_2^b, t_2^s, t_2^a, t_2^o), \ldots, (t_k^b, t_k^s, t_k^a, t_k^o)\}.$$

The verbs for WSD set ($vrbwsd$) will be discussed in Section 3.3.2.2. Some additional sets were predefined in order to help the process to derive the BSAO tuples. They are:

a) $usefulWords$. This set contains POS tags that aid in the process of creating a key for the words within the requirements. The key is defined as "counter of Useful Words ($cntUW$) - word". This key is useful for accurate identification of words which are part of the Action of the tuple. This set also defines the lexical categories of the running/useful words for the WSD algorithm (Section 3.3.2.2). The chosen lexical categories were: common nouns (singular and plural), verbs, adjectives, and adverbs;

b) $candSubObj$. This set contains POS tags that define the candidate words to be Subjects and Objects. The selected lexical categories were: common nouns (singular and plural), proper nouns (singular and plural), adjectives, cardinal numbers, and coordinating conjunctions;

c) $confirmSub$. This set contains POS tags that confirm that a previous word identified as a Subject is indeed a Subject. The selected lexical categories were: modal verbs and verbs. This set is important because after a Subject usually there is a verb. If no verb is found, it is likely that a word previously identified as a Subject is not in fact a Subject;

d) $confirmAct$. Set that contains POS tags that characterize an Action. The selected lexical categories were: verbs, adverbs, and coordinating conjunctions;

e) $endTuple$. Set that contains POS tags which aid in the decision whether a BSAO tuple must be created or not. The chosen lexical categories were: common nouns (singular and plural), proper nouns (singular and plural), verbs, adjectives, adverbs, modal verbs, and cardinal numbers.

```
input  : dictionary $dic = \{N, R, S_{TM}.C, S_{TM}.F, S_{TM}.Y\}$
input  : allWords $allw = \{w_f^g \mid g = lw, tg, id\}, f = 1..h$
output: tuples $tup = \{t_i^j \mid j = b, s, a, o\}, i = 1..k$
output: verbsWSD $vrbwsd = \{v_p^q \mid q = ivr, inf\}, p = 1..r$
initializeAuxiliaryVariables();
for $f \leftarrow 1$ to $h$ do
    while $w_f^{tg} \neq$ "." do
        if $w_f^{tg} \in usefulWords$ then
            $cntUW \leftarrow cntUW + 1$;
        end
        if $w_f^{tg} =$ "$IN$" $\wedge$ $w_f^{lw} \in S_{TM}.C$ then
            $behavior \leftarrow behavior + w_f^{lw}$;
        end
        if $w_f^{tg} \in candSubObj \wedge \neg subCreated$ then
            if $iLastSub = 1 \vee iLastSub = f - 1$ then
                $subject \leftarrow subject + w_f^{lw}$;
                $iLastSub \leftarrow f$;
            else
                if $iLastSub \neq f - 1$ then
                    $subject \leftarrow empty$;
                    $subject \leftarrow w_f^{lw}$;
                    $iLastSub \leftarrow f$;
                end
            end
        else
            if $w_f^{tg} \in confirmSub \wedge subject \neq empty$ then
                $subCreated \leftarrow true$;
                if $w_f^{tg} \in confirmAct$ then
                    if $iLastAct = 1 \vee iLastAct = f - 1$ then
                        $action \leftarrow action +$ "$cntUW$" $+$ " $-$ " $+ w_f^{lw}$;
                        $iLastAct \leftarrow f$;
                        $actCreated \leftarrow true$;
                    end
                end
            else
                if $w_f^{tg} \in confirmAct \wedge subCreated$ then
                    if $iLastAct = 1 \vee iLastAct = f - 1$ then
                        $action \leftarrow action +$ "$cntUW$" $+$ " $-$ " $+ w_f^{lw}$;
                        $iLastAct \leftarrow f$;
                        $actCreated \leftarrow true$;
                    end
                else
                    if $w_f^{tg} \in candSubObj \wedge actCreated$ then
                        if $iLastObj = 1 \vee iLastObj = f - 1$ then
                            $object \leftarrow object + w_f^{lw}$;
                            $iLastObj \leftarrow f$;
                            $objCreated \leftarrow true$;
                        end
                    end
                end
            end
        end
        $f \leftarrow f + 1$;
        if $subCreated \wedge actCreated \wedge objCreated \wedge w_f^{tg} =$ "." $\vee$ $w_f^{tg} =$ "," $\vee$ $w_f^{tg} \notin endTuple$ then
            $(t_i^b,\ t_i^s,\ t_i^a,\ t_i^o) \leftarrow (behavior,\ subject,\ action,\ object)$;
        end
    end
end
$vrbwsd \leftarrow$ generateVerbsWSD($tup, dic$);
```

Figure 3.10 - Main algorithm for generating BSAO tuples: procedure

The auxiliary variables such as *behavior*, *subject*, *action*, and *object* are also initialized (task accomplished by the procedure *initializeAuxiliaryVariables()* in line 1 of Figure 3.10) elsewhere in the algorithm: after the line 51 and also just before the line 54. One of the first tasks accomplished within the algorithm is the probable identification of a Behavior (b) feature. Thus, the algorithm first verifies if a word is a *preposition or subordinating conjunction* (POS tag "IN") and also if such a word is in $S_{TM}.C$ (lines 7 to 9). If these conditions are matched, then the b element of the tuple is assigned to such a word. If not, b is empty. Note that the $S_{TM}.C$ set has words like "if", and, at first, the user does not need to alter it. This set is independent of the application domain and it was derived aiming to add in the resulting Statechart model behaviors due to the semantics associated with some requirements.

After the determination of b, the Subject (s), Action (a), and Object (o) elements of the tuple are identified (lines 10 to 48). Essentially, the algorithm verifies whether the POS tags of words are equal to predefined POS tags that characterize a Subject (according to the sets *candSubObj* and $confirmSub$), an Action (according to the set $confirmAct$), or an Object (based on set *candSubObj*). If they match, the corresponding s, a, and o elements of the tuple are fulfilled.

However, a BSAO tuple is created **if and only if** a Subject and an Action and an Object were identified, and some other conditions were satisfied (lines 50 to 52). If these conditions are not satisfied, no BSAO tuple is created. This is to avoid situations which might occur in NL sentences, and might produce ill-formed tuples. For instance, a piece of a sentence might derive an s, an a but not an o because the sentence ended. The algorithm picks the next ordered triple of the *allw* set (line 49) in order to continue the assessment of words within a sentence (a sentence is delimited by a period). However, the algorithm also checks whether not only if s, a and o were created but also if the POS tag of the next ordered triple of the *allw* set is a period to decide about the creation of a BSAO tuple.

Note that the algorithm for generating BSAO tuples can generate Subject made up of several words. Line 12 of the algorithm shows that the auxiliary variable $subject$ is updated so that the Subject element of the tuple (t_i^s, line 51) can be filled with the correct composition of words. Conditions must be satisfied for a word to be part of the Subject: POS tag of such a word must be in *candSubObj* set; and, Subject must not have been yet created (variable *subCreated*; line 10). There is also a mechanism to avoid the inclusion of words whose POS tags are in the *candSubObj* set but do not really make part of the Subject (lines 11 to 13). Similar remarks are valid for creating Action (lines 24 to 37, and line 51) and Object (lines 39 to 45, and line 51) formed by several words.

In order to illustrate how a BSAO tuple is automatically generated, consider the following requirement:

> If the main software system does not start operating the air conditioning system at midnight, maintenance personnel should be called.

Table 3.4 shows how a BSAO tuple is generated considering each word of this requirement. The POS tag and consequently the lexical category of each word is provided by the *Stanford POS Tagger*. The Behavior (B), Subject (S), Action (A), and Object (O) columns contain the value of each one of these tuple's elements after applying algorithm shown in Figure 3.10.

The conditions that determine behavior (line 7) were satisfied and hence the Behavior element of the tuple was fulfilled accordingly. In addition and as explained above, several words make up the Subject, Action, and Object. The content of Action was derived with the keys previously mentioned. In other words, "4-do" is the key due to the verb "does". The number (4) is the current "counter of Useful Words (*cntUW*)" which uniquely identifies this word within the set of NL requirements. In this case, "do" is the lemma of "does". Notice that the same explanation applies to "6-start" due to the verb "start", and "7-operate" due to "operating".

The NL requirement presented in Table 3.4 might generate another BSAO tuple with Behavior empty, "maintenance personnel" as Subject, and "14-be 15-call" as Action. However, the sentence ends and no Object has been detected. Therefore, a second BSAO tuple is not created because the Object element is missing.

Table 3.4 - Generation of a BSAO tuple. Caption: Cat = Category

Word	**Lexical Cat**	**Tag**	**B**	**S**	**A**	**O**
If	preposition or subordinating conjunction	IN	if			
the	determiner	DT	if			
main	adjective	JJ	if	main		
software	common noun, singular	NN	if	main software		

(Continues)

Table 3.4 - Conclusion

Word	Lexical Cat	Tag	B	S	A	O
system	common noun, singular	NN	if	main software system		
does	verb, present tense, 3rd person singular	VBZ	if	main software system	4-do	
not	adverb	RB	if	main software system	4-do 5-not	
start	verb, base form	VB	if	main software system	4-do 5-not 6-start	
operating	verb, gerund or present participle	VBG	if	main software system	4-do 5-not 6-start 7-operate	
the	determiner	DT	if	main software system	4-do 5-not 6-start 7-operate	
air	common noun, singular	NN	if	main software system	4-do 5-not 6-start 7-operate	air
conditioning	common noun, singular	NN	if	main software system	4-do 5-not 6-start 7-operate	air conditioning
system	common noun, singular	NN	if	main software system	4-do 5-not 6-start 7-operate	air conditioning system
at	preposition or subordinating conjunction	IN				
midnight	common noun, singular	NN				
maintenance	common noun, singular	NN				
personnel	common noun, plural	NNS				
should	modal verb	MD				
be	verb, base form	VB				
called	verb, past participle	VBN				

3.3.1.1 Relation with typed dependencies

This section presents some other important observations about the Subject, Action, and Object of the BSAO tuples. These comments will be presented from the perspective of the *Stanford typed dependencies* (MARNEFFE; MANNING, 2008) which are provided by the *Stanford Parser* (this is another tool, other than the *Stanford POS Tagger*, developed by Stanford University). Typed dependencies are also known as *grammatical relations*.

The goal of these observations is to show the relation between the approach proposed in this PhD thesis to identify Subject, Action, and Object, which was based on the lexical categories of words, with another point of view based on grammatical relations. Thus, according to the proposal presented in this PhD thesis, this section provides preliminary guidelines to identify the Subject, Action, and Object of tuples by using typed dependencies rather than lexical categories.

If all the above conditions are satisfied, both in the active voice and passive voice, the syntactic subject of a sentence or clause will be the Subject of the tuple. In this case, the typed dependencies involved would be *nominal subject* and *passive nominal subject*. However, in order to consider Subjects formed by several words, it is necessary to take into account other typed dependencies such as *noun compound modifier* and *adjectival modifier*.

Typed dependencies *nominal subject* and *passive nominal subject* are also important to identify the Action of the tuple. In order to allow the Action to be composed of several words, various other typed dependencies should be considered such as *auxiliary*, *passive auxiliary*, *negation modifier*, *open clausal complement*, and *phrasal verb particle*.

In the active voice, the main typed dependency related to the Object of the tuple is *direct object* whereas in the passive voice is the typed dependency *agent* (the performer of the action). As in previous cases, other typed dependencies should be considered such as *noun compound modifier*, *adjectival modifier*, and *prepositional modifier*.

3.3.2 Translation from BSAO tuples into behavioral model

The main algorithm supporting this second sub-activity is shown, in form of an activity diagram, in Figure 3.11 while Figure 3.12 shows the same algorithm in form of a procedure. In the sequence, the explanations about the algorithm are given taking into account the form of procedure that describes the algorithm (Figure 3.12). The remarks about notation made in Section 3.3.1 apply to Figure 3.12. One additional remark is about the notation related to the Reactiveness (R) function. The notation $R_n(rie)$ means this is the element of the domain of R ($rie \in R.I_E$) of the nth ordered pair of R. Similar observations apply to the codomain of R ($R.O_E$), and also to the $S_{TM}.Y$ function.

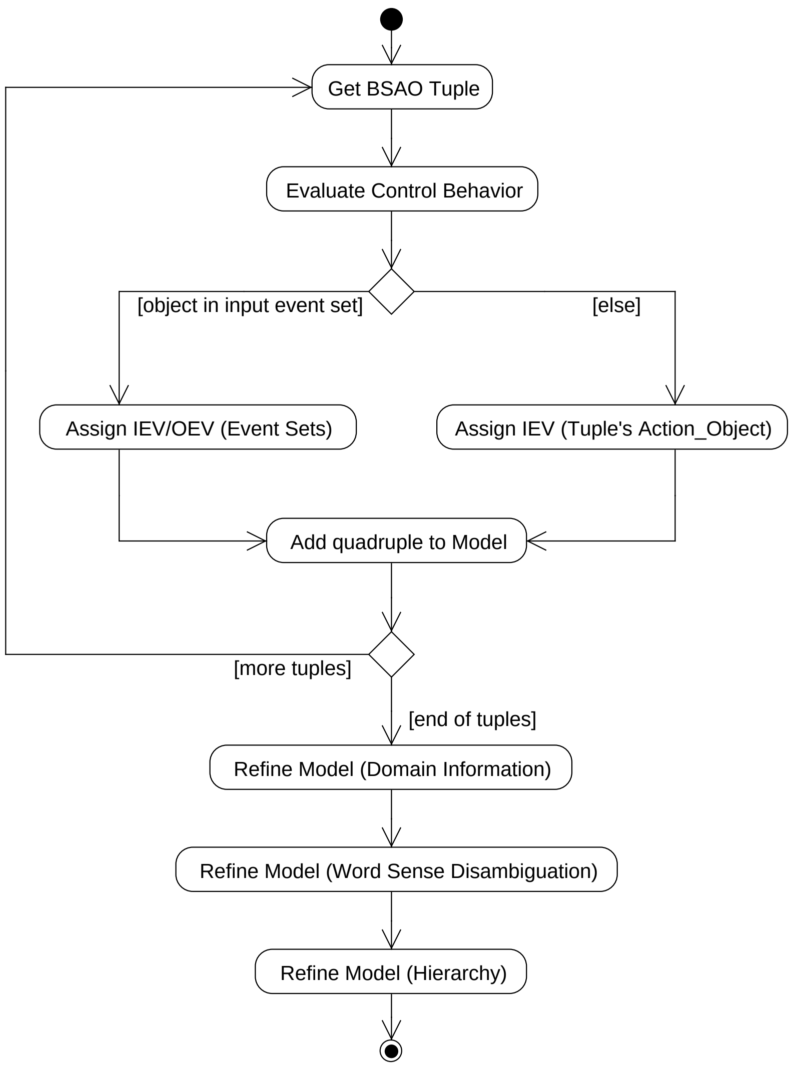

Get BSAO Tuple
Evaluate Control Behavior
[object in input event set]
[else]
Assign IEV/OEV (Event Sets)
Assign IEV (Tuple's Action_Object)
Add quadruple to Model
[more tuples]
[end of tuples]
Refine Model (Domain Information)
Refine Model (Word Sense Disambiguation)
Refine Model (Hierarchy)

Figure 3.11 - Main algorithm for the translation of BSAO tuples into models: activity diagram

As shown in Figure 3.12, the Dictionary and the BSAO tuples are the basis for generating the model. The generated model (mod) is a set of ordered quadruples where each ordered quadruple represents a transition in the model. Hence, each ordered quadruple is composed of: a source state (upper index *src*), an input event (upper index *iev*), an output event (upper index *oev*), and a destination state (upper index *des*). Formally, *mod* is a set such as:

$$mod = \{(m_1^{src}, m_1^{iev}, m_1^{oev}, m_1^{des}), \ldots, (m_z^{src}, m_z^{iev}, m_z^{oev}, m_z^{des})\}.$$

The initial idea is to denote the states of the model with Subjects of the BSAO tuples. This is clearly shown in lines 6, 14 and 31 in Figure 3.12. However, the *checkSubject* procedure (lines 6 and 14) checks whether there is already a source state in the model with the same name of the current BSAO Subject. The *checkSubject* procedure returns a name for the state just adding an underscore followed by an incrementing number after the Subject, if there is already a same state name in the model; otherwise, it will return the same BSAO Subject to be assigned as the name of the source state.

The Reactiveness (R) function of the Dictionary plays an important role in defining input and output events within a transition. As shown from lines 22 to 28, if the Object of the BSAO tuple exists in the input event set ($R.I_E$), then the input event (*iev*) will be assigned to the matched element of the domain of R ($R.I_E$), and the output event (*oev*) will have the value of the corresponding element of the codomain of R ($R.O_E$). However, if the tuple's Object does not match any element of $R.I_E$, the input event becomes a combination of Action_Object of the BSAO tuple, and the output event null. Another remark is that the if-then-else situation in NL sentences is addressed from lines 3 to 21. Hence, more than one transition may be leaving the same source state in the resulting model.

Consider the following two requirements from the SWPDC case study:

- SRS001 - The PDC shall be powered on by the Power Conditioning Unit.
- SRS002 - The PDC shall be in the Initiation Operation Mode after being powered on. The 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.

These requirements produce 6 BSAO tuples as shown in Table 3.5. Piece of the resulting model set based on these tuples is shown in Table 3.6.

```
input : dictionary $dic = \{N, R, S_{TM}.C, S_{TM}.F, S_{TM}.Y\}$
input : tuples $tup = \{t_i^j \mid j = b, s, a, o\}, i = 1..k$
input : verbsWSD $vrbwsd = \{v_p^q \mid q = ivr, inf\}, p = 1..r$
output: model $mod = \{m_x^y \mid y = src, iev, oev, des\}, x = 1..z$
$cond \leftarrow false$;
for $i \leftarrow 1$ to $k$ do
    if $t_i^b \in S_{TM}.C \wedge \neg cond$ then
        $existBeh \leftarrow t_i^b$;
        $existSub \leftarrow t_i^s$;
        $tmod^{src} \leftarrow$ checkSubject($t_i^s$);
        $lastState \leftarrow tmod^{src}$;
        $cond \leftarrow true$;
    else
        if $t_i^b = existBeh \wedge t_i^s = existSub$ then
            $tmod^{src} \leftarrow lastState$;
            $cond \leftarrow false$;
        else
            $tmod^{src} \leftarrow$ checkSubject($t_i^s$);
            if $t_i^b \in S_{TM}.C$ then
                $existBeh \leftarrow t_i^b$;
                $existSub \leftarrow t_i^s$;
                $lastState \leftarrow tmod^{src}$;
            end
        end
    end
    if $t_i^o \in R.I_E$ then
        // $rie \in R.I_E$.
        $tmod^{iev} \leftarrow R_n(rie)$;
        // $roe \in R.O_E$.
        $tmod^{oev} \leftarrow R_n(roe)$;
    else
        $tmod^{iev} \leftarrow t_i^a + “\_” + t_i^o$;
        $tmod^{oev} \leftarrow null$;
    end
    $tmod^{des} \leftarrow null$;
    // add quadruple to model $mod$.
    $x \leftarrow i$;
    $m_x^{src} \leftarrow tmod^{src}$;
    $m_x^{iev} \leftarrow tmod^{iev}$;
    $m_x^{oev} \leftarrow tmod^{oev}$;
    $m_x^{des} \leftarrow tmod^{des}$;
end
// first automated refinement.
$mod \leftarrow$ refineModelDomainInfo($mod, dic$);
// second automated refinement.
$mod \leftarrow$ refineModelWordSense($mod, dic, vrbwsd$);
// third automated refinement.
$mod \leftarrow$ refineModelHierarchy($mod, dic$);
```

Figure 3.12 - Main algorithm for the translation of BSAO tuples into models: procedure

Table 3.5 - BSAO tuples obtained from requirements SRS001 and SRS002

B	S	A	O
-	PDC	1-be 2-power on	Power Conditioning Unit
-	PDC	3-be	Initiation Operation Mode
-	SWPDC	6-then 7-accomplish	post
if	PDC	9-present	irrecoverable problem
-	computer	13-remain	Initiation Operation Mode
-	problem	15-not 16-be 17-propagate	OBDH

Table 3.6 - Piece of the model set obtained from the BSAO tuples in Table 3.5. Caption: #Tr = Transition number

#Tr	Source State	Input Event	Output Event	Destination State
1	PDC	1-be 2-power on_Power Conditioning Unit	null	null
2	PDC_2	3-be_Initiation Operation Mode	null	null
3	SWPDC	6-then 7-accomplish_post	null	null
4	PDC_3	9-present_irrecoverable problem	null	null
5	computer	13-remain_Initiation Operation Mode	null	null
6	problem	15-not 16-be 17-propagate_OBDH	null	null

Up to this point, the number of transitions in the created model is equal to the number of BSAO tuples. Observe that the Subjects of the BSAO tuples define the name of the source states in Table 3.6. However, the *checkSubject* procedure creates different names for source states when finding "PDC" repeatedly by adding "_" followed by an incrementing number. None of the Objects of BSAO tuples are in $R.I_E$. Hence, each input event is a concatenation of Action_Object of the BSAO tuple, and the output event is null. Besides, all destination states are also null.

Some ordered quadruples of the model may be removed. As shown from lines 36 to 38 of the main algorithm (Figure 3.12), there are three types of automated refinements which are applied to the original model so that the final model may be enhanced with respect to the original one. The concept of refinement is defined below.

Definition 3.2. Refinement is a mechanism that aims to better adapt the generated model considering the context of Statechart-based test case generation.

3.3.2.1 Refinement based on domain information

The first automated refinement is to eliminate unnecessary states and transitions of the model, to rename certain states of the model, and to fulfill the destination states of the transitions. It is called a *refinement based on domain information* ($refineModelDomainInfo$ in Figure 3.12). The algorithm to refine the model based on domain information is shown, in form of an activity diagram, in Figure 3.13 while Figure 3.14 shows the same algorithm in form of a procedure. In the sequence, the explanations about the algorithm are given taking into account the form of procedure that describes the algorithm (Figure 3.14).

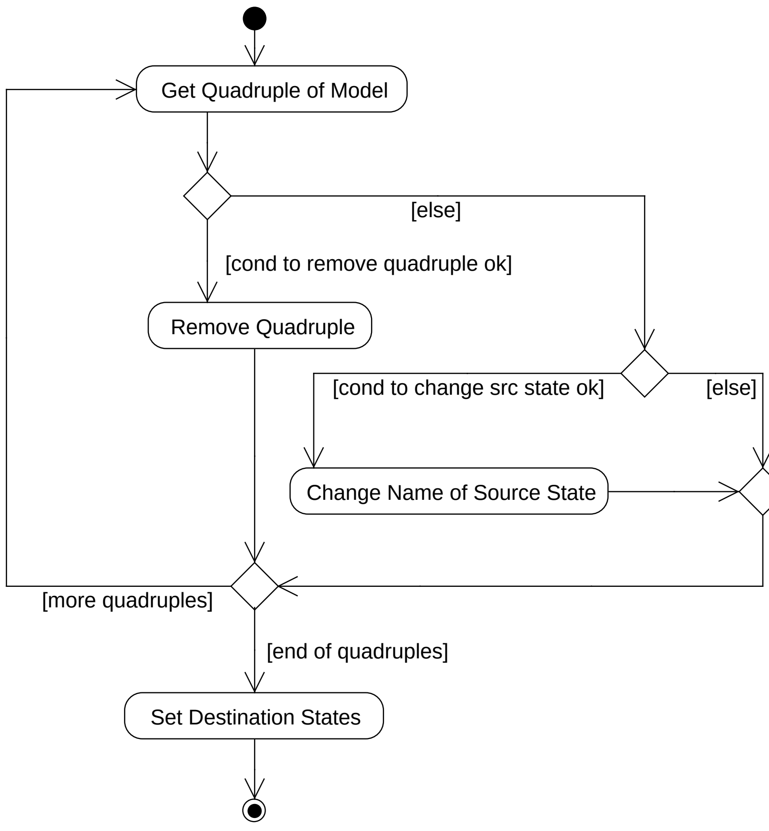


Figure 3.13 - *Refinement based on domain information*: activity diagram

As shown from lines 6 to 8, an ordered quadruple (a transition) is removed from the original model if the source state ($srcSta$) does not exist in N, and the Object part of the input event ($inpObj$) does not exist in N, and the entire input event (m_x^{iev}) does not exist either in $R.I_E$. This is done because not all NL sentences contain relevant information to justify the creation of a transition in the context of model-based test

```
input  : model $mod = \{m_x^y \,|\, y = src, iev, oev, des\}, x = 1..za$
input  : dictionary $dic = \{N, R, S_{TM}.C, S_{TM}.F, S_{TM}.Y\}$
output: model $mod = \{m_x^y \,|\, y = src, iev, oev, des\}, x = 1..zb$
1  $inpObj, inpAct, srcSta \leftarrow empty$;
2  for $x \leftarrow 1$ to $za$ do
3      $srcSta \leftarrow removeUnderlineFromSourceState(m_x^{src})$;
4      $inpAct \leftarrow extractAction(m_x^{iev})$;
5      $inpObj \leftarrow extractObject(m_x^{iev})$;
       // check whether an ordered quadruple must be removed from the model.
6      if $srcSta \notin N \wedge inpObj \notin N \wedge m_x^{iev} \notin R.I_E$ then
7          $mod \leftarrow removeOrderedQuadruple(m_x^{src}, m_x^{iev}, m_x^{oev}, m_x^{des})$;
8          $x \leftarrow x - 1$;
9      else
           // check whether the name of the source state must be changed.
10         if $inpObj \in N \wedge m_x^{iev} \notin R.I_E$ then
11             $m_x^{src} \leftarrow inpObj$;
12             $m_x^{iev} \leftarrow inpAct$;
13         end
14     end
15 end
   // set the destination states.
16 for $x \leftarrow 1$ to $zb$ do
17     if $x \neq zb$ then
18         $m_x^{des} \leftarrow m_{x+1}^{src}$;
19     else
20         $m_x^{des} \leftarrow m_1^{src}$;
21     end
22 end
```

Figure 3.14 - *Refinement based on domain information* algorithm: procedure

case generation.

If an ordered quadruple is not to be removed (lines 10 to 13), the name of its source state may be changed if the Object part of the input event (*inpObj*) exists in N and the entire input event (m_x^{iev}) does not exist in $R.I_E$. This is explained due to the fact that 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. This implies that the name of the states would be basically limited to those names added by a counter (recall the *check_Subject* procedure) such as system, system_1, system_2, and so on. In order to improve this and to provide more meaningful names for states, the new name of the state is changed to a word or words that are in N (Object part of the input event (*inpObj*)).

After the processing presented above, a last feature of the *refinement based on domain information* is setting the destination states. This is simply done considering the source state of the next ordered quadruple as the destination state of the current ordered quadruple (line 18). However, the destination state of the last transition of the model is the first (initial) state (line 20). The FSM test criteria implemented in GTSC (DS, UIO and H-switch cover) require that the flat FSM is strongly connected. In such a machine for each pair of states (s_i, s_j), there is a path[5] connecting s_i to s_j. This explains the logic of setting the destination state of the last transition. Even though the Statechart criteria implemented in GTSC (all-transitions, all-simple-paths, all-paths-k-C0-configuration, all-paths-k-configurations) do not have that restriction, this action makes the model translated from NL requirements more generic in the sense that a test designer may choose any of the seven GTSC test criteria to generate the test suite.

Table 3.7 shows the piece of the model set after the *refinement based on domain information* considering requirements SRS001 and SRS002. Assuming that $N = \{$PDC, SWPDC, Initiation Operation Mode, OBDH, ...$\}$, no transition is eliminated. Note that *computer* and *problem* are not in N but the Object parts of the input events (after "_") of these transitions (in Table 3.6: *Initiation Operation Mode* in transition 5 and *OBDH* in transition 6) are in N. Because the Object parts of the input events are in N, the source states of transitions 2, 5, and 6 are changed according to them. Finally, the destination states are set in accordance with the processing presented earlier.

3.3.2.2 Word Sense Disambiguation refinement

In the SOLIMVA tool, the second automated refinement ($refineModelWordSense$ in Figure 3.12) refers to an adaptation of the graph-based approach proposed by Sinha and Mihalcea (2007) taking into account only one similarity measure, *Jiang and Conrath*, and one graph-based centrality algorithm, *indegree* (Section 2.2.1 presented an overview of the proposal of Sinha and Mihalcea (2007) as well as the *Jiang and Conrath* measure and the *indegree* algorithm). The set $usefulWords$ (Section 3.3.1) attempts to define the lexical categories (common nouns (singular and plural), verbs, adjectives, and adverbs) required for proper operation of this adaptation of the proposal of Sinha and Mihalcea (2007). The goal of such adaptation was to automate the identification of the semantics related to the generated model. Specifically, the idea was to automatically identify self transitions in the resulting Statechart model.

[5]A path is a finite sequence of adjacent transitions. Note that this definition of path is different from the one given in Section 2.1.1.2 where a path is defined as a finite sequence of configurations.

Table 3.7 - Piece of the model set after the *refinement based on domain information.* Caption: #Tr = Transition number

#Tr	**Source State**	**Input Event**	**Output Event**	**Destination State**
1	PDC	1-be 2-power on_Power Conditioning Unit	null	Initiation Operation Mode
2	Initiation Operation Mode	3-be	null	SWPDC
3	SWPDC	6-then 7-accomplish_post	null	PDC_3
4	PDC_3	9-present_irrecoverable problem	null	Initiation Operation Mode
5	Initiation Operation Mode	13-remain	null	OBDH
6	OBDH	15-not 16-be 17-propagate	null	PDC_3

Considering the BSAO tuples and the *refinement based on domain information* (Section 3.3.2.1), the default behavior is: if the current state of the model is s_i, the next state is s_j where $i \neq j$. The goal was to determine in which situations and based on the set of NL requirements the next state is s_i, i.e. when a self transition occurs within the model. To achieve this goal, the *synsets* related to **verbs** in *WordNet* were manually searched to find verb's senses which mean "remain in a same place". The interpretation is that finding a verb with this particular sense implies that the model could exhibit a self transition.

Eleven verbs ("continue", "remain", "stay", etc.) with a total of 21 sense numbers were found which satisfied the conditions for self transition. These 21 sense numbers are the elements of the $S_{TM}.F$ set. The $S_{TM}.F$ set is independent of the application domain and thus the test designer does not need to change it. A sample of the $S_{TM}.F$ set is as follows, where the number indicates the sense number as defined in *WordNet*:

$$S_{TM}.F = \{\text{remain\#v\#1, remain\#v\#2, stay\#v\#1, ..., rest\#v\#6, ...}\}.$$

In order to obtain *Jiang and Conrath* measures between pairs of verbs, version 11.01 of the *Java WordNet::Similarity* (UNIVERSITY OF SUSSEX, 2010), a Java version of the Perl *WordNet::Similarity* package developed by the University of Minessota (PEDERSEN et al., 2004), was used. Both packages use as corpus, by default, SemCor (MILLER et al., 1993) which is a manually sense-tagged subset of the Brown Corpus. In addition, version 2.1 of *WordNet* was selected for use.

Among the four graph-based centrality algorithms used by Sinha and Mihalcea (2007), the *indegree* algorithm was implemented in the SOLIMVA tool. Recall that for an undirected weighted graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, the indegree is defined as

$$indegree(V_a) = \sum_{V_b \in V} wg_{ab}$$

where wg_{ab} is the weight on the edge between V_a and V_b. In the graph generated by the SOLIMVA tool, V_a and V_b are senses of two distinct verbs, and wg_{ab} is the *Jiang and Conrath* measure between these two verb's senses. Furthermore, $wn = 4$ which was the value that provided the best results regarding the correct word sense assignment. In order to identify the sense of a verb, the algorithm only figures out the sense with the highest score among all the senses of a verb.

The reason to choose only the *Jiang and Conrath* similarity measure it was because the goal was to disambiguate the senses of verbs. According to the results shown in Sinha and Mihalcea (2007), the best measure in terms of true positives for verbs was the *Leacock and Chodorow* followed closely by the *Jiang and Conrath* measure (the *Jiang and Conrath* measure was 4.55% worse than the *Leacock and Chodorow* measure; other measures were more than 10% worse than the *Leacock and Chodorow* measure). At first, the *Leacock and Chodorow* measure was tried in the SWPDC's Software Requirements Specification but the results were not very promising. Thus, the *Jiang and Conrath* measure was selected which presented a better performance.

The reasoning behind the selection of the *indegree* graph-based centrality algorithm was also because this was the best algorithm for verb's sense disambiguation outperforming the other three algorithms (SINHA; MIHALCEA, 2007). The initial idea was to implement the four graph-based centrality algorithms and combine them in a voting scheme as proposed by Sinha and Mihalcea (2007). But, the results presented for verbs according to their approach were not very promising and the decision was to implement the algorithm with best performance, the *indegree*.

The main WSD refinement algorithm is shown, in form of an activity diagram, in

Figure 3.15 while Figure 3.16 shows the same algorithm in form of a procedure. In the sequence, the explanations about the algorithm are given taking into account the form of procedure that describes the algorithm (Figure 3.16). The verbs for WSD set (*vrbwsd*) contains all verbs of the set of NL requirements that match the verbs defined in $S_{TM}.F$. Each ordered pair of *vrbwsd* is composed of: the key ("counter of Useful Words (*cntUW*) - word") defined in Section 3.3.1 (upper index *ivr*), and one out of two possible values (upper index *inf*), "self" and "no". This set has been previously generated (Figure 3.10) which informs whether there are verbs within the NL requirements that characterize a self transition in the resulting model (value "self"). Hence, the adaptation of the approach proposed by Sinha and Mihalcea (2007) was in fact implemented as part of the BSAO tuples generation algorithm. However, to make use of the results by adapting the Sinha and Mihalcea (2007) approach to identify self transitions in the Statechart model, additional steps are necessary as described in Figure 3.16. Finally, the importance of the key is due to the fact that naturally the same verb (for instance, stay) may occur several times within the NL requirements indicating or not indicating the presence of several self transitions. So it is necessary to know exactly which word (verb) is being evaluated.

The algorithm in Figure 3.16 takes the *vrbwsd* set as input, and first extracts the Action part (if any) of the input event of a transition and realizes whether it matches a key of *vrbwsd* (lines 7 to 8). Assuming this is true and also that the information related to the key matches the value "self" (line 10), the algorithm performs a backward search until it finds a previous source state which is the same as the current source state where the verb (key) characterizes a self transition (lines 12 to 20). As long as this does not occur, the input events of all transitions backward traversed are stored (*tempIET*, line 14) in order to compose the new input event of a transition in the model (line 24). The new current destination state is exactly the current source state which is the expected behavior in such a situation (line 25). Intermediate and needless states and transitions are removed from the model (line 26), and the new next source state is also set (line 27).

To sum up, there are four conditions to be satisfied so that a self transition is detected and added in the model after the WSD refinement. First, the verb must match the verbs in $S_{TM}.F$. Second, the information must be "self". Third, a previous source state must match the current source state in the backward search. This is important because if there is no match then it is not possible to decide which are the source and destination states of the self transition. Finally, during the backward search, none of the input events must match an element of $R.I_E$. Again, this demonstrates

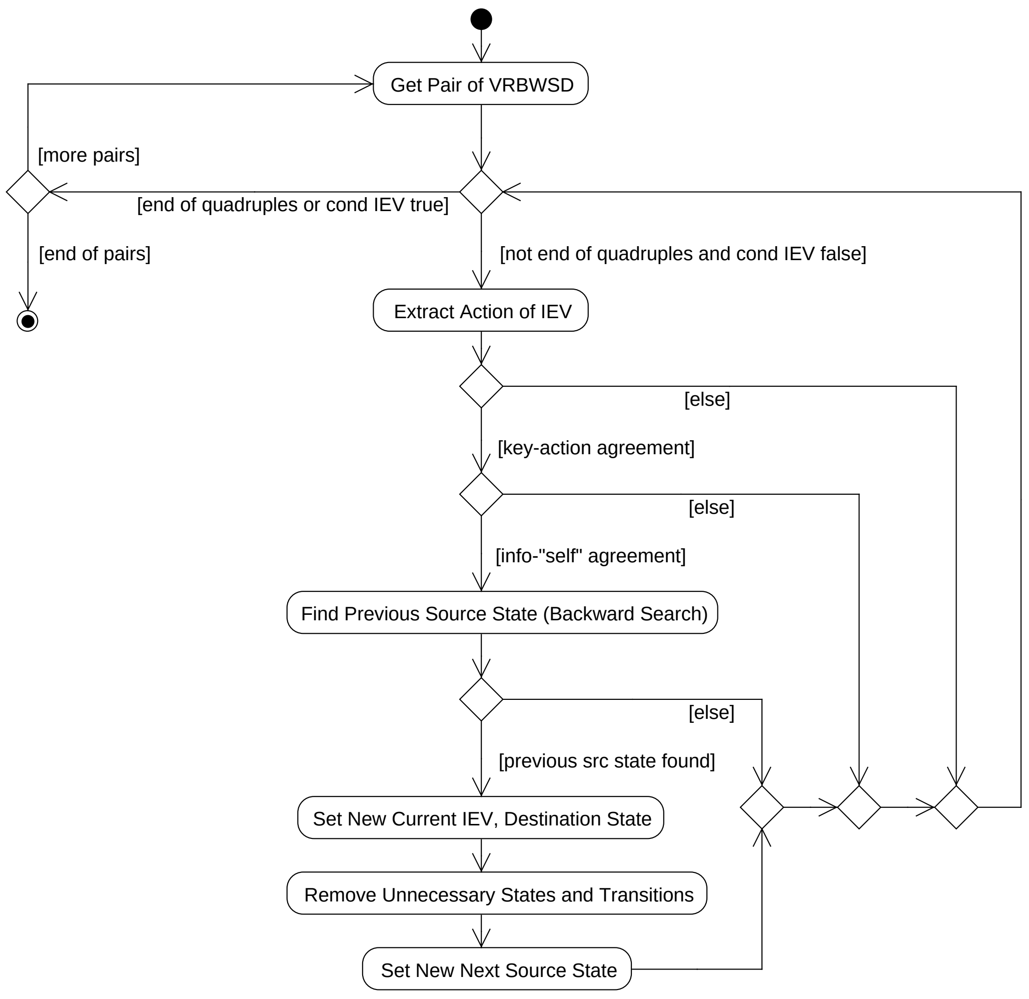

Get Pair of VRBWSD
[more pairs]
[end of quadruples or cond IEV true]
[end of pairs]
[not end of quadruples and cond IEV false]
Extract Action of IEV
[else]
[key-action agreement]
[else]
[info-"self" agreement]
Find Previous Source State (Backward Search)
[else]
[previous src state found]
Set New Current IEV, Destination State
Remove Unnecessary States and Transitions
Set New Next Source State

Figure 3.15 - Main WSD refinement algorithm: activity diagram

the priority of reactiveness over other behaviors in the resulting model.

Requirements SRS001 and SRS002 produce the following keys (verbs): “1-be”, “2-power”, “3-be”, “4-be”, “5-power”, “7-accomplish”, “9-present”, “13-remain”, “16-be”, and “17-propagate”. Within the *generateVerbsWSD* procedure (Figure 3.10), the sense (label) dependency graph is created considering all verbs and with $wn = 4$. For instance, verb “be” has 13 distinct senses (sense numbers: be#v#1, be#v#2, ..., be#v#13), verb “power” has only one sense (sense number: power#v#1), and verb “accomplish”' has 2 senses (sense numbers: accomplish#v#1, accomplish#v#2). As mentioned earlier, each sense of each verb is a vertex of the graph and *Jiang and Conrath* measures are obtained between senses of different verbs since they are encompassed by the value of the window (4). Thus, for verb “1-be”, the measures are obtained considering all senses of “2-power”, “3-be”, “4-be”, “5-power” and the

```
input  : model $mod = \{m_x^y \mid y = src, iev, oev, des\}, x = 1..zb$
input  : dictionary $dic = \{N, R, S_{TM}.C, S_{TM}.F, S_{TM}.Y\}$
input  : verbsWSD $vrbwsd = \{v_p^q \mid q = ivr, inf\}, p = 1..r$
output: model $mod = \{m_x^y \mid y = src, iev, oev, des\}, x = 1..zc$
$inpAct, newIET, tempIET \leftarrow empty$;
$matIET, matSrc \leftarrow false$;
$ib, it \leftarrow 1$;
for $p \leftarrow 1$ to $r$ do
    $x \leftarrow 1$;
    while $x \leq zb \wedge \neg matIET$ do
        $inpAct \leftarrow extractAction(m_x^{iev})$;
        if $v_p^{ivr} = inpAct$ then
            $matIET \leftarrow true$;
            if $v_p^{inf} =$ "$self$" then
                $ib \leftarrow x$;
                while $ib \geq 1 \wedge \neg matSrc$ do
                    $ib \leftarrow ib - 1$;
                    $tempIET_{it} \leftarrow m_{ib}^{iev}$;
                    $it \leftarrow it + 1$;
                    if $m_x^{src} = m_{ib}^{src} \wedge m_{ib}^{iev} \notin R.I_E$ then
                        $newIET \leftarrow reverseTempIETGenerateNewIET(tempIET)$;
                        $matSrc \leftarrow true$;
                    end
                end
                $tempIET \leftarrow empty$;
                $it \leftarrow 1$;
                if $matSrc$ then
                    // new current input event.
                    $m_{ib}^{iev} \leftarrow newIET$;
                    // new current destination state.
                    $m_{ib}^{des} \leftarrow m_{ib}^{src}$;
                    // remove unnecessary states and transitions.
                    $mod \leftarrow removeStatesTransitions(mod, ib + 1)$;
                    // new next Source State.
                    $m_{ib+1}^{src} \leftarrow m_{ib}^{src}$;
                    $matSrc \leftarrow false$;
                    $newIET \leftarrow empty$;
                end
            end
        end
        $x \leftarrow x + 1$;
    end
    $matIET \leftarrow false$;
end
```

Figure 3.16 - Main WSD refinement algorithm: procedure

senses of “1-be” themselves. For “13-remain”, the other verbs to generate *Jiang and Conrath* measures are “9-present”, “16-be”, and “17-propagate”. Figure 3.17 shows a small part of the entire sense dependency graph focusing on key (verb) “13-remain”.

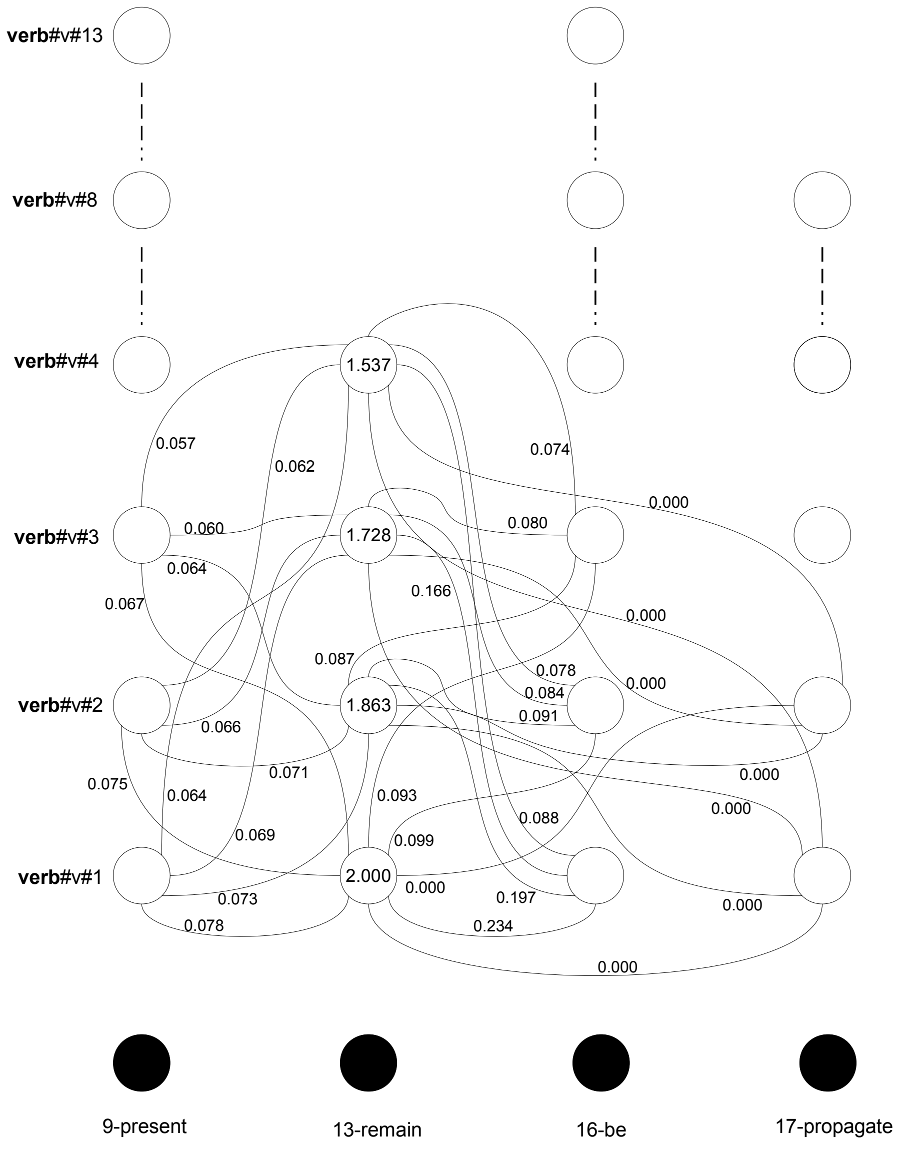


Figure 3.17 - A small part of the entire sense dependency graph due to requirements SRS001 and SRS002

From all verbs present in requirements SRS001 and SRS002, only verb “remain” has some sense numbers in $S_{TM}.F$. In Figure 3.17, the number of different senses of verbs “present”, “remain”, “be”, and “propagate” are 13, 4, 13, and 8, respectively. So, exactly these amounts of vertices (white circles) were created in the graph. Each value on an edge of the graph is the *Jiang and Conrath* measure between two senses of distinct verbs. For instance, the measure between present#v#1 (the first sense of "9-present") and remain#v#1 (the first sense of "13-remain") is 0.078 while the measure between remain#v#3 (the third sense of "13-remain") and be#v#2 (the second sense of "16-be") is 0.084.

The values inside the vertices (senses) of verb "13-remain" are the scores derived from the *indegree* algorithm. Since the highest score is 2.000 then the selected sense of "13-remain" is the first one: remain#v#1. Because remain#v#1 is in $S_{TM}.F$, the information is "self". The current source state, i.e. the state whose input event is "13-remain", is *Initiation Operation Mode* (transition 5 in Table 3.7). During the backward search, the WSD algorithm finds another source state named *Initiation Operation Mode* (transition 2 in Table 3.7). Moreover, during the backward search, none of the input events are in $R.I_E$. Therefore, all conditions are satisfied and a self transition is added into the model. Table 3.8 shows the piece of the model set after the WSD refinement based on requirements SRS001 and SRS002. Transition number 2 is a self transition. Moreover, the input event of transition 2 is the concatenation of previous input events as designed in the main WSD refinement algorithm.

Table 3.8 - Piece of the model set after the WSD refinement. Caption: #Tr = Transition number

#Tr	Source State	Input Event	Output Event	Destination State
1	PDC	1-be 2-power on_Power Conditioning Unit	null	Initiation Operation Mode
2	Initiation Operation Mode	3-be#6-then 7-accomplish_post# 9-present_irrecoverable problem#	null	Initiation Operation Mode
3	Initiation Operation Mode	15-not 16-be 17-propagate	null	PDC_3

3.3.2.3 Refinement for adding hierarchy

The output model obtained after the WSD refinement is no more than an FSM as defined in Chapter 2. In order to obtain a Statechart model, a strategy was created to incorporate hierarchy (depth) into the final model. This is the last automated refinement incorporated into the SOLIMVA tool ($refineModelHierarchy$ in Figure 3.12). Figure 3.18 shows, in form of an activity diagram, this algorithm while Figure 3.19 shows the same algorithm in form of a procedure. In the sequence, the explanations about the algorithm are given taking into account the form of procedure that describes the algorithm (Figure 3.19).

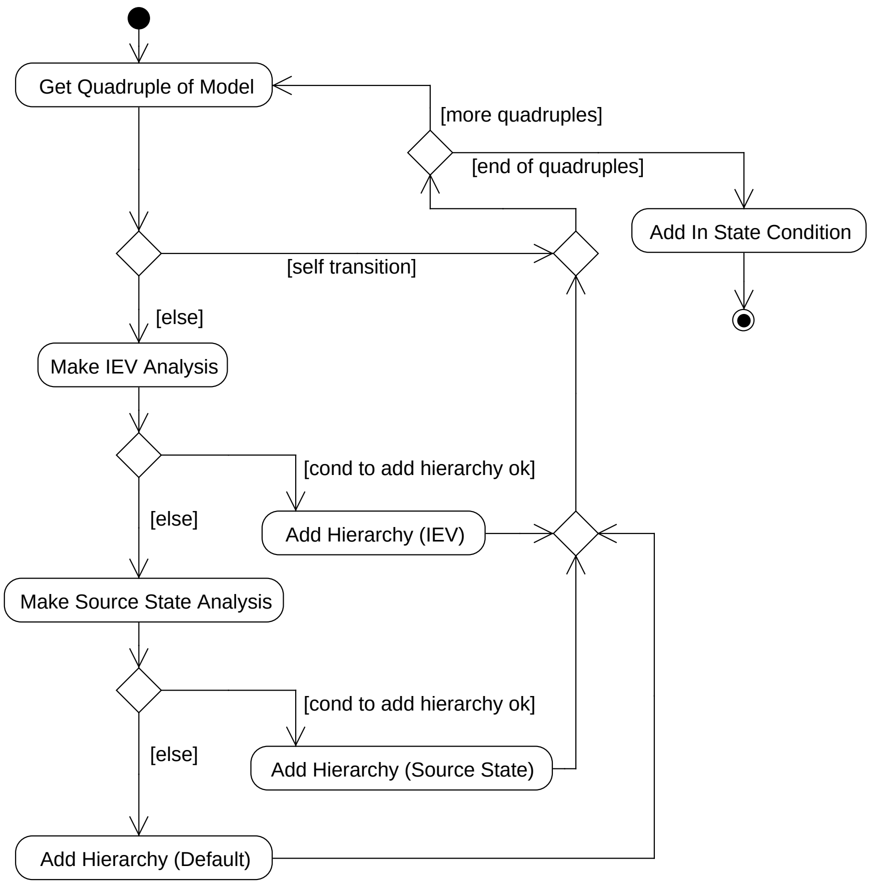


Figure 3.18 - The algorithm to add hierarchy into the model: activity diagram

The algorithm partitions the set of states of the model based on information gathered from the $S_{TM}.Y$ function. Such a function is a mapping among some names of source states (m_x^{src}) and input events (m_x^{iev}) of transitions, which are elements of the input pattern set ($Y.I_P$), and names that will define the COMPOSITE states of the Statechart model, which are elements of the output pattern set ($Y.O_P$). The selected names of source states are in N as well as the names of input events are in $R.I_E$.

One basic assumption is that no COMPOSITE state will be created if a self transi-

```
input  : model $mod = \{m_x^y \mid y = src, iev, oev, des\}, x = 1..zc$
input  : dictionary $dic = \{N, R, S_{TM}.C, S_{TM}.F, S_{TM}.Y\}$
output: model $mod = \{m_x^y \mid y = src, iev, oev, des\}, x = 1..z$
$match \leftarrow false$;
for $x \leftarrow 1$ to $zc$ do
    if $m_x^{src} \neq m_x^{des}$ then
        if $x = zc$ then
            $match \leftarrow checkInputPatternSet(m_x^{iev}, \{\})$;
        else
            if $x = zc - 1$ then
                $match \leftarrow checkInputPatternSet(m_x^{iev}, \{m_{x+1}^{iev}\})$;
            else
                $match \leftarrow checkInputPatternSet(m_x^{iev}, \{m_{x+1}^{iev}, m_{x+2}^{iev}\})$;
            end
        end
        if $match = true$ then
            $mod \leftarrow addHierarchyIEV(mod)$;
        else
            if $x = zc$ then
                $match \leftarrow checkInputPatternSet(m_x^{src}, \{\})$;
            else
                if $x = zc - 1$ then
                    $match \leftarrow checkInputPatternSet(m_x^{src}, \{m_{x+1}^{src}, m_{x+1}^{iev}\})$;
                else
                    $match \leftarrow checkInputPatternSet(m_x^{src}, \{m_{x+1}^{src}, m_{x+1}^{iev}, m_{x+2}^{src}, m_{x+2}^{iev}\})$;
                end
            end
            if $match = true$ then
                $mod \leftarrow addHierarchySRC(mod)$;
            else
                $mod \leftarrow addHierarchyDefault(mod)$;
            end
        end
    end
end
$mod \leftarrow addInStateCondition(mod)$;
```

Figure 3.19 - The algorithm to add hierarchy into the model: procedure

tion exists in a certain state. This is shown in line 3 of Figure 3.19. In other words, if source (m_x^{src}) and destination (m_x^{des}) states are equal then no processing aiming to add hierarchy into the model is accomplished. Moreover, the general idea is to detect whether the source state or the input event of a transition is in $Y.I_P$ and, if so, names of appropriate source and destination states are changed so that depth can be incorporated into the model.

The procedure *checkInputPatternSet* has as arguments an element of a transition that should be checked if it is in $Y.I_P$ and a second argument that is actually a set of elements that should not be in $Y.I_P$. First, an analysis is made considering the input events (m_x^{iev}; lines 4 to 12). In the case of the last transition of the model (line 4) being evaluated, this second argument is not relevant (represented by the empty set, $\{\}$, in line 5). The value of the boolean variable *match* is true if the input event is in $Y.I_P$. For the penultimate transition of the model (line 7), the second argument is the next input event (m_{x+1}^{iev}; line 8). So even if the current input event (m_x^{iev}) is in $Y.I_P$, *match* will be false if the next input event is also in $Y.I_P$. If neither the last nor the the penultimate transition of the model is being assessed (line 10), then the analysis is performed if m_x^{iev} is in $Y.I_P$ and the next two input events ($m_{x+1}^{iev}, m_{x+2}^{iev}$) are not in $Y.I_P$. In other words, *match* will be true if the current input event is in the input pattern set and none of the next two input events are. Hence, a COMPOSITE state is created if *match* is true and thus hierarchy is incorporated into the model (lines 13 to 14).

Similar solution was adopted for the assessment taking into account the source states (m_x^{src}; lines 16 to 26). The basic difference is that in the penultimate transition (line 19) it is evaluated if the current source state is in $Y.I_P$ and both the next source state (m_{x+1}^{src}) and the next input event (m_{x+1}^{iev}) are not in $Y.I_P$ (line 20) to *match* becomes true. From the first to the antepenultimate transition, the absence of the next two source states ($m_{x+1}^{src}, m_{x+2}^{src}$) and the next two input events ($m_{x+1}^{iev}, m_{x+2}^{iev}$) in the input pattern set are requirements for the addition of hierarchy into the model as well as, of course, the current source state (m_x^{src}) is in $Y.I_P$ (line 22).

The reasoning behind the inclusion of hierarchy only if next input events and/or source states are not in the input pattern set is to avoid creating an excessive number of COMPOSITE states and/or COMPOSITE states encompassing few states. Some other requirements were considered when designing the algorithm for incorporating hierarchy. First, depth was limited to two levels, say the main level and a second hierarchy level. Although depth is an important feature of Statecharts, from the point of view of models for system and acceptance test case generation, too many hierarchy levels may make the entire modeling difficult to read. Two levels seem to be enough for this purpose.

There is no priority among COMPOSITE states. In other words, at any time the behavior may be "leave the COMPOSITE state *s*" if the conditions previously described are satisfied. Another feature is that the initial state of the main Statechart model is not allowed to be part of a COMPOSITE state, this is verified inside the procedures to add hierarchy (*addHierarchyIEV*, *addHierarchySRC*, *addHierarchyDefault*), and there is also a mechanism to include the *In State* conditions of Statecharts into the final model (line 33).

The SOLIMVA tool does not consider the creation of orthogonal (parallel) states. Orthogonality (parallelism) is not very relevant in the context of this PhD thesis because the main goal is to automatically create models for system and acceptance test case generation. In summary, use case scenarios are derived and the IUT is stimulated one command at a time. Thus, orthogonality is not very important. Note that this is entirely different if test cases had to be generated based on models created by the development team in which orthogonality is very likely to occur, and important to be accounted for. With this characteristic, models generated by the SOLIMVA methodology and by the SOLIMVA tool are easier to read and the translation from the Abstract Test Suite into the Executable Test Suite is not complex either.

An observation about what was called output event (m_x^{oev}) in this work should be made. According to the designation given by Harel et al. (1987), output events are called *actions* in Statecharts. In Statecharts, an action is not simply sent to the "outside world" as an output such as occurs in FSMs. An action typically affects the behavior modeled in orthogonal components of Statecharts. This is achieved by the broadcasting mechanism. However, as the SOLIMVA tool does not take into account orthogonality as explained above, the mechanism of broadcasting turns out to be ineffective with no action as internal event that could cause firing of transitions in orthogonal components. Therefore, output events (m_x^{oev}) end up being semantically similar to *outputs* in FSMs. With respect to the input events (m_x^{iev}), they are known as *events* (HAREL et al., 1987) in Statecharts and simply *inputs* in the terminology of FSMs (PETRENKO; YEVTUSHENKO, 2005).

Consider the SWPDC case study. Hence, $Y.I_P$, $Y.O_P$, and $S_{TM}.Y$ can be composed of:

- $Y.I_P$ = {Safety Operation Mode, CH-OP-MODE-NOMINAL, ... },
- $Y.O_P$ = {Safety Operation Mode, Nominal Operation Mode, ... },
- $S_{TM}.Y$ = {(Safety Operation Mode, Safety Operation Mode), (CH-OP-MODE-NOMINAL, Nominal Operation Mode), ... }.

3.4 Final remarks about this chapter

This chapter presented version 1.0 of the SOLIMVA methodology which addresses the primary objective of this PhD thesis. First, a brief description of the main case study, SWPDC software product (SANTIAGO et al., 2007), in which the methodology was applied, was made. After that, descriptions of version 1.0 of the SOLIMVA methodology and of the SOLIMVA tool were made. SOLIMVA relies on several theories such as combinatorial designs (MATHUR, 2008), POS Tagging (TOUTANOVA et al., 2003), WSD (NAVIGLI, 2009), and, of course, Model-Based Testing (EL-FAR; WHITTAKER, 2001; UTTING; LEGEARD, 2007) to generate model-based test cases.

Next chapter presents the application of version 1.0 of the SOLIMVA methodology to the SWPDC software product where all the elements associated with the methodology, such as factors, levels, NL requirements, models, Abstract Test Cases, Executable Test Cases and others, are shown in detail. At the end of Chapter 4, other observations about version 1.0 of the SOLIMVA methodology and the SOLIMVA tool are presented.

4 APPLICATION OF THE SOLIMVA METHODOLOGY

This chapter shows in detail the application of version 1.0 of the SOLIMVA methodology and supporting tools (SOLIMVA tool and GTSC environment) to the SWPDC software product (SANTIAGO et al., 2007). In this chapter when used without any explicit indication of the version number, the terms "SOLIMVA methodology" and "SOLIMVA" refer to version 1.0 of the SOLIMVA methodology. The comparison between the SOLIMVA methodology and an expert's (manual) approach under the aspects of coverage of test objectives and characteristics of Executable Test Cases are detailed as well as directives to apply the methodology to a second case study related to the Ground Segment (CARDOSO et al., 2008) are also shown. An abridged version of this chapter can be seen in SANTIAGO JÚNIOR and VIJAYKUMAR (2012).

The test designer starts by defining the Dictionary. Then, he/she can create the Name (N) set and Reactiveness (R) function in accordance with Table 4.1, and one possibility for the *Semantic Translation Model* is shown in Table 4.2: Control ($S_{TM}.C$) set, Self Transition ($S_{TM}.F$) set, and Hierarchy ($S_{TM}.Y$) function. In functions, the representation is $x \rightarrow y$ where x is an element of the domain of the function and y is an element of the codomain of the function. Furthermore, the elements in CAPITAL letters of the domain ($R.I_E$) and codomain ($R.O_E$) of R are simply abbreviations for commands and responses of the PDC-OBDH Communication Protocol Specification. Thus, VER-OP-MODE is an abbreviation for the command VERIFY PDC's OPERATION MODE. As previously stated, $S_{TM}.C$ and $S_{TM}.F$ are already defined in configuration files of the SOLIMVA tool and hence the user does not need to alter them. However, if necessary the user has an option to change them.

Table 4.1 - Sample of the Name set and the Reactiveness function for the SWPDC case study

Name	Reactiveness
PDC	VER-OP-MODE → INFO-OP-MODE
SWPDC	PREP-HK → CMD-REC
Initiation Operation Mode	TX-DATA-SCI-End → SCI-DATA or NO-DATA
Safety Operation Mode	CH-OP-MODE-NOMINAL → CMD-REC
Nominal Operation Mode	CH-OP-MODE-SAFETY → CMD-REC
EPP Hx	Several TX-DATA-HK → Several HK-DATA or NO-DATA
OBDH	...
...	

Table 4.2 - Sample of the *Semantic Translation Model* for the SWPDC case study

Control	**Self Transition**	**Hierarchy**
if	remain#v#1	Initiation Operation Mode → Initiation Operation Mode
...	remain#v#2	Safety Operation Mode → Safety Operation Mode
	stay#v#1	Nominal Operation Mode → Nominal Operation Mode
	stay#v#2	CH-OP-MODE-NOMINAL → Nominal Operation Mode
	stay#v#4	CH-OP-MODE-SAFETY → Safety Operation Mode
	rest#v#1	...
	rest#v#6	
	continue#v#1	
	...	

After that, scenarios are defined using the strategy described in Section 3.2. First, the core elements in this case study are the 37 commands that the OBDH can send to PDC which are defined in the PDC-OBDH Communication Protocol Specification. These 37 commands were grouped into 28 scenarios. These are very simple scenarios consisting essentially in switching the PDC on and send these commands in order to realize whether PDC correctly receives and processes such commands.

For the normal scenarios, Table 4.3 shows a possible choice of factors and levels for the SWPDC case study. The meanings of the factors Cmd, OpMode, and Services as well as the levels HwSwHnd, DtAcqTx, Nom, Sci, and Hk have already been given in Section 3.2. The explanation for the remaining factors and levels follows:

a) Levels of the Cmd factor: processing activities to manage PDC's operation mode (OpMMgm), load new program into PDC's Data Memory on the fly (PrLoad);

b) Levels of the OpMode factor: Initiation/Initialization (Init), Safety (Safe), and Diagnosis (Diag) are other PDC's operation modes;

c) Levels of the Services factor: Services related to acquisition, formatting, and transmission of Test (Tst) and Diagnosis (Dg) Data, generation, formatting, and transmission of Dump (Dmp) Data, loading new program into PDC's Data Memory on the fly (Load);

d) StartMode factor. This factor relates to the way PDC is started: Power On (PwrOn) or Reset (Reset).

Since there are four factors and as explained in the *Define Scenarios* activity (Fig-

Table 4.3 - Factors and levels for the SWPDC case study

Factors	**Levels**						
Cmd	HwSwHnd	OpMMgm	DtAcqTx	PrLoad	Inv		
OpMode	Nom	Init	Safe	Diag	Inv		
Services	Sci	Hk	Dmp	Load	Dg	Tst	Inv
StartMode	PwrOn	Reset	Inv				

ure 3.5), then *strength* = 3. The combinatorial designs algorithm produced 175 factor combinations which shall be interpreted to derive 175 normal scenarios. One example is the factor combination 71: {DtAcqTx, Nom, Sci, Inv}. However, the test designer may neglect the level *Inv* assuming that robustness will be covered by another factor combination. Hence, factor combination 71 becomes: {DtAcqTx, Nom, Sci, -}. The interpretation of such factor combination defines scenario 71 to "acquire, format, and transmit Scientific Data in the Nominal Operation Mode".

All normal scenarios were analyzed to see if they needed to be unfolded. Normal scenarios 73 ({DtAcqTx, Nom, Dmp, -}) whose interpretation is "generation, formatting, and transmission of Dump Data in the Nominal Operation Mode", and 94 ({DtAcqTx, Diag, Dmp, -}) whose interpretation is "generation, formatting, and transmission of Dump Data in the Diagnosis Operation Mode", are two normal scenarios to be unfolded. The Dump Data (Dmp) service refers to get data from pieces of PDC's Program or Data Memories. Organization of PDC's Program Memory is simple with a 64-kByte program address space, from 0000h to FFFFh (h = hexadecimal). However, organization of PDC's Data Memory is more complicated as shown in Figure 4.1. The size of the data address space is also 64 kBytes, from 0000h to FFFFh. However, there is a paging mechanism which enables PDC to access more than 64 kBytes of Data Memory. Each one of the 8 pages in Figure 4.1 has a size of 32 kBytes. Hence, in order to dump data from PDC's memories the user must:

- select the memory from where to dump data. The options are Program and Data Memories. In case the user selects the Data Memory, the page (0 to 7) must be selected too;
- provide the initial and final 16-bit memory addresses.

The selection is accomplished by means of a specific command defined in the PDC-OBDH Communication Protocol Specification. Then, dumping from different pieces of memory areas can be made, and robustness aspects of SWPDC may also be

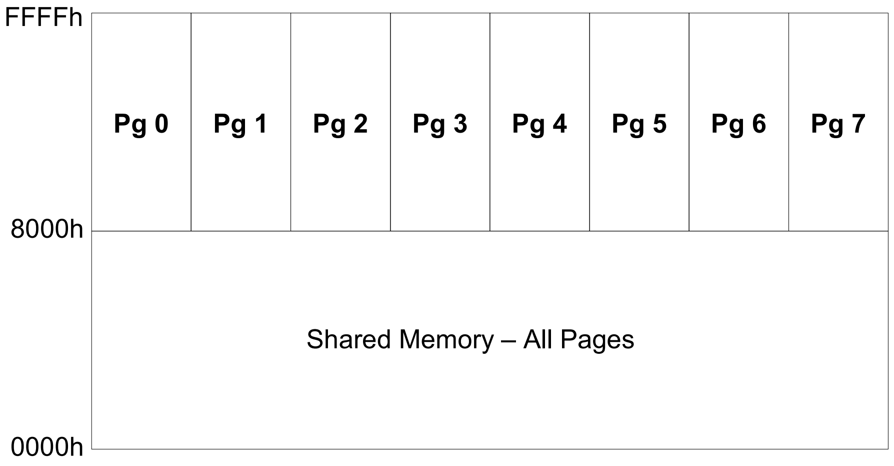


Figure 4.1 - Organization of PDC's Data Memory. Caption: Pg = Page

emphasized. For instance, what is its behavior when the initial address is greater than the final address, the initial or final address is less than the minimum physical address allowed for a certain type of memory and so on. These facts explain the need for unfolding.

Table 4.4 shows the configuration of factors and levels for unfolding normal scenarios 73 and 94. The priority factor in this case is the type of Memory (Mem; Prg = Program Memory, DtP0 = Data Memory - Page 0, ..., DtP7 = Data Memory - Page 7) from where the dumping will take place, and *strength* = 2. The initial memory address (IniAdd) and final memory address (FinalAdd) are the other factors. Since the priority factor has 10 levels, then not only normal scenario 73 but also normal scenario 94 will be replaced with 10 unfolded scenarios each (note that the only difference between normal scenarios 73 and 94 is the PDC's operation mode). These unfolded scenarios are identified as 73.1, 73.2, ..., 73.10, 94.1, 94.2, ..., 94.10.

When running the combinatorial designs algorithm, the factor combinations that are interpreted for deriving each unfolded scenario are shown in Table 4.5. As mentioned in Section 3.2, the test designer adds NL requirements to characterize each unfolded scenario according to the directives provided by the combinatorial designs algorithm.

It is interesting to stress the focus on Robustness testing related to this unfolding process. For instance, unfolded scenarios 73.3 or 94.3 propose the following situations, where all cases relate to Page 1 of PDC's Data Memory (DtP1):

- IniAdd = Min, FinalAdd = LessMin;

- IniAdd = Max, FinalAdd = InRng;
- IniAdd = LessMin, FinalAdd = Min;
- IniAdd = GreatMax, FinalAdd = Min;
- IniAdd = Min, FinalAdd = GreatMax;
- IniAdd = InRng, FinalAdd = Max.

Table 4.4 - Unfolding normal scenarios 73 and 94

Factors	**Levels**									
Mem	Prg	DtP0	DtP1	DtP2	DtP3	DtP4	DtP5	DtP6	DtP7	Inv
IniAdd	InRng	Min	Max	LessMin	GreatMax					
FinalAdd	InRng	Min	Max	LessMin	GreatMax					

SWPDC must process only the last situation (IniAdd = InRng, FinalAdd = Max) because all the others have innapropriate settings of initial and/or final memory addresses. The unfolding process generated 20 additional scenarios, 10 to replace normal scenario 73 and another 10 to replace normal scenario 94.

Four other normal scenarios needed to be unfolded: normal scenario 109 ({PrLoad, Nom, Load, -}; "loading new program into PDC's Data Memory on the fly in the Nominal Operation Mode") which contributed with 6 additional scenarios; normal scenarios 2 ({HwSwHnd, Nom, Hk, -}), 16 ({HwSwHnd, Safe, Hk, -}), 23 ({HwSwHnd, Diag, Hk, -}) whose interpretations are "verification of correct implementation of commands related to software parameters manipulation, and generation, formatting, and transmission of Housekeeping Data in the Nominal (scenario 2), Safety (scenario 16) or Diagnosis (scenario 23) Operation Modes", and that provided 7 additional scenarios each one, adding 21 new scenarios. Hence, the total number of scenarios proposed by the SOLIMVA methodology for the SWPDC case study was 244 as shown in Table 4.6. Due to the fact that six normal scenarios have been unfolded and then substituted for unfolded scenarios the total amount of normal scenarios decreased compared to the value (175) originally obtained.

For each scenario, a set of NL requirements is chosen. As a matter of illustration, Table 4.7 shows the set of NL requirements in order to characterize normal scenario 71: {DtAcqTx, Nom, Sci, -}.

Table 4.5 - Factor combinations for creating unfolded scenarios from normal scenarios 73 and 94. Caption: Min/Max = Minimum/Maximum allowed memory address; LessMin/GreatMax = Less/Greater than Minimum/Maximum allowed memory address; InRng = Address between Min and Max (In Range)

Unfolded Scenario	**Factor Combinations**
73.1 and 94.1	{Prg, InRng, InRng}, {Prg, Min, Min}, {Prg, Max, Max}, {Prg, LessMin, LessMin}, {Prg, GreatMax, GreatMax}
73.2 and 94.2	{DtP0, InRng, Min}, {DtP0, Min, InRng}, {DtP0, Max, LessMin}, {DtP0, LessMin, Max}, {DtP0, GreatMax, InRng}, {DtP0, Min, GreatMax}
73.3 and 94.3	{DtP1, InRng, Max}, {DtP1, Min, LessMin}, {DtP1, Max, InRng}, {DtP1, LessMin, Min}, {DtP1, GreatMax, Min}, {DtP1, Min, GreatMax}
73.4 and 94.4	{DtP2, InRng, LessMin}, {DtP2, Min, Max}, {DtP2, Max, Min}, {DtP2, LessMin, InRng}, {DtP2, GreatMax, Max}, {DtP2, Min, GreatMax}
73.5 and 94.5	{DtP3, InRng, GreatMax}, {DtP3, Min, InRng}, {DtP3, Max, Min}, {DtP3, LessMin, Max}, {DtP3, GreatMax, LessMin}
73.6 and 94.6	{DtP4, InRng, InRng}, {DtP4, Min, GreatMax}, {DtP4, Max, Min}, {DtP4, LessMin, Max}, {DtP4, GreatMax, LessMin}
73.7 and 94.7	{DtP5, InRng, InRng}, {DtP5, Min, Min}, {DtP5, Max, GreatMax}, {DtP5, LessMin, Max}, {DtP5, GreatMax, LessMin}
73.8 and 94.8	{DtP6, InRng, InRng}, {DtP6, Min, Min}, {DtP6, Max, Max}, {DtP6, LessMin, GreatMax}, {DtP6, GreatMax, LessMin}
73.9 and 94.9	{DtP7, InRng, InRng}, {DtP7, Min, Min}, {DtP7, Max, Max}, {DtP7, LessMin, LessMin}, {DtP7, GreatMax, GreatMax}
73.10 and 94.10	{Inv, InRng, InRng}, {Inv, Min, Min}, {Inv, Max, Max}, {Inv, LessMin, LessMin}, {Inv, GreatMax, GreatMax}

Table 4.6 - Total number of scenarios for the SWPDC case study

Type of Scenario	**Quantity**
Simple	28
Normal	169
Unfolded	47
Total	244

After the selection of the requirements, the next step is to generate the Statechart model. Figure 4.2 shows the main model generated by the SOLIMVA tool for normal scenario 71. Note the self transition in state *Initiation Operation Mode*. This behavior occurs due to the verb "remain" in requirement SRS002 as explained in the WSD refinement section (Section 3.3.2.2). There are three COMPOSITE states (symbol ∞) in the main Statechart model: *Nominal Operation Mode* (Figure 4.3), *Safety Operation Mode_2* (Figure 4.4), and *Safety Operation Mode* (Figure 4.5). When

Table 4.7 - Set of NL requirements that characterize normal scenario 71. Caption: Req = Requirement; Id = Identification

Req Id	**Req Description**
SRS001	The PDC shall be powered on by the Power Conditioning Unit.
SRS002	The PDC shall be in the Initiation Operation Mode after being powered on. The 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.
SRS003	If PDC does not present any irrecoverable problem, after the initiation process, the PDC shall automatically enter into the Safety Operation Mode.
POCP001	The PDC can only respond to requests (commands) from OBDH after the PDC has been energized for at least 1 minute. If OBDH sends commands within less than 1 minute, the OBDH shall not receive any response from PDC.
RB001	The OBDH shall send VER-OP-MODE to PDC.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH sends distinct commands to perform such actions.
PECP001	Each EPP Hx can only respond to requests (commands) from PDC after each EPP Hx has been energized for at least 30 seconds. If PDC sends commands within less than 30 seconds to a certain EPP Hx, the PDC shall not receive any response from this EPP Hx.
SRS004	The OBDH should wait 600 seconds before asking for a Housekeeping Data frame.
SRS005	Housekeeping data transmission shall start with PREP-HK. After that, the OBDH can send several TX-DATA-HK to PDC. The transmission shall be ended with TX-DATA-SCI-End.
RB003	The OBDH shall send CH-OP-MODE-NOMINAL to PDC.
RB001	The OBDH shall send VER-OP-MODE to PDC.
POCP002	The OBDH should wait 10 seconds before asking for a Scientific Data frame.
SRS006	The SWPDC shall obtain and handle scientific data from each EPP Hx. The SWPDC shall also accept scientific data transmission requests from OBDH.
RB004	The OBDH shall send CH-OP-MODE-SAFETY to PDC. After that, the PDC shall be in the Safety Operation Mode.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH sends distinct commands to perform such actions.
RB005	After switching both EPPHxs off via PDC, the OBDH shall switch the PDC off via the Power Conditioning Unit.

omitted, the output event within a transition is *null*. Moreover, some transitions are of the form *input event[In(A.B)]/output event*.[1] For these situations, A means a COMPOSITE state and B the substate inside COMPOSITE state A. In Figure 4.2, transition *CH-OP-MODE-NOMINAL[In(SOM.OBDH_7)]/CMD-REC* is fired if the input event *CH-OP-MODE-NOMINAL* occurs but also if the current active state of

[1]Alternatively, following the notation in Harel et al. (1987): *event[in(state) condition]/action*.

the model is *OBDH_7* of the COMPOSITE state *Safety Operation Mode* (*SOM*). Similar remarks are valid for other transitions where the abbreviations for COMPOSITE states should be accounted for: *Nomimal Operation Mode* (*NOM*), *Safety Operation Mode_2* (*SOM_2*).

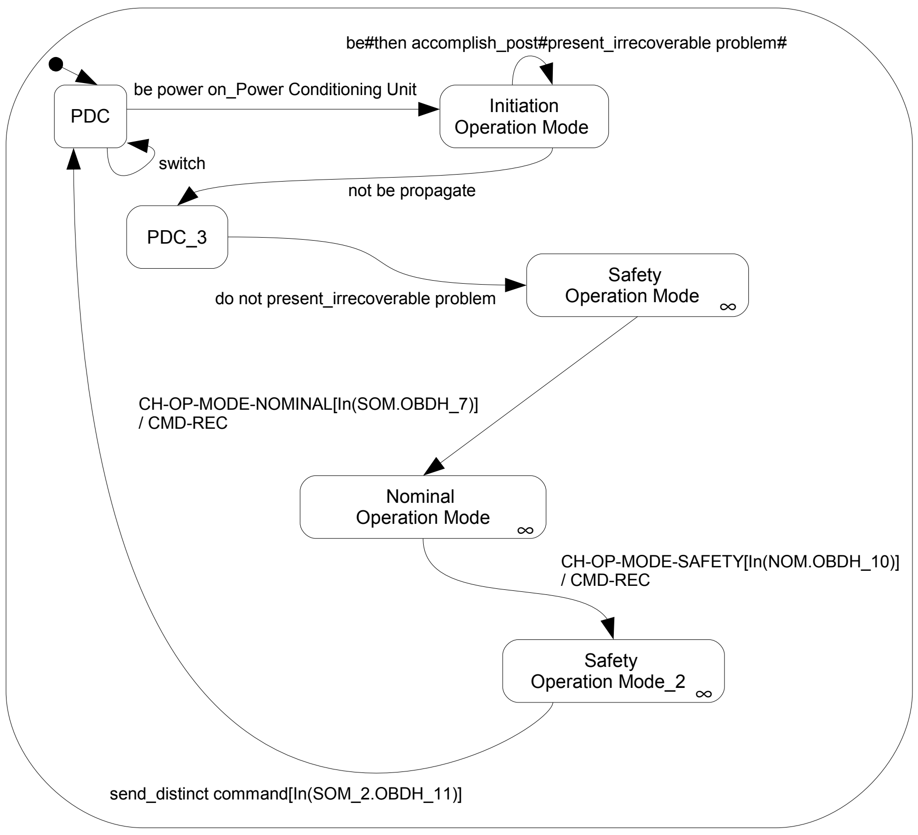

be#then accomplish_post#present_irrecoverable problem#
be power on_Power Conditioning Unit
PDC
Initiation Operation Mode
switch
not be propagate
PDC_3
Safety Operation Mode
do not present_irrecoverable problem
CH-OP-MODE-NOMINAL[In(SOM.OBDH_7)] / CMD-REC
Nominal Operation Mode
CH-OP-MODE-SAFETY[In(NOM.OBDH_10)] / CMD-REC
Safety Operation Mode_2
send_distinct command[In(SOM_2.OBDH_11)]

Figure 4.2 - The main Statechart model derived from NL requirements that characterize normal scenario 71

In the sequence, GTSC is run to generate Abstract Test Cases. The Abstract Test Suites created from all-transitions, all-simple-paths, and all-paths-k-C0-configuration test criteria are presented in Table 4.8. An Abstract Test Case is composed of a set of *input/output* pairs and it is delimited by left and right braces.

Abstract Test Cases must be translated into Executable Test Cases in order to stimulate the IUT. Tables 4.9 and 4.10 show examples of such translations considering the second Abstract Test Case of the all-transitions Abstract Test Suite. It is interesting to realize that in some cases, there is no specific *test input data* to stimulate the IUT but only *actions* shall be performed and certain behaviors shall occur. Besides, the

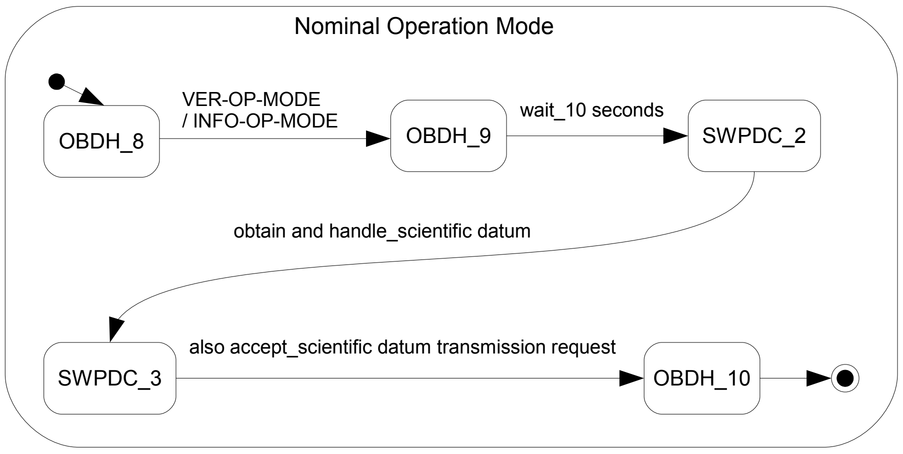


Figure 4.3 - Normal scenario 71: COMPOSITE state *Nominal Operation Mode*

mapping is not one to one, i.e. not necessarily one *input/output* pair of an Abstract Test Case will derive exactly one *test input data/expected result* of an Executable Test Case.

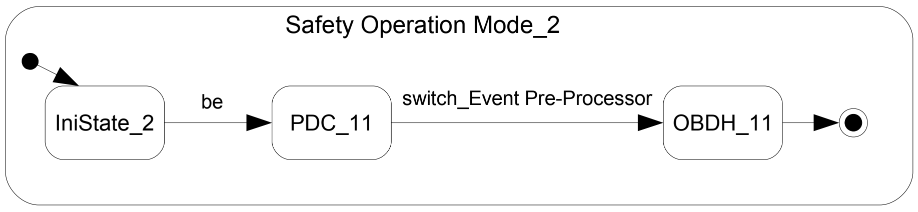


Figure 4.4 - Normal scenario 71: COMPOSITE state *Safety Operation Mode_2*

Another remark is that most of the translations shown in Tables 4.9 and 4.10 are applied not only to other Abstract Test Suites derived from other test criteria for normal scenario 71 but also to other Abstract Test Suites due to the all-transitions or other test criteria regarding another scenario. The point is that it is not necessary to do all the mapping when considering other scenarios: most of the translations are reusable.

The complete translation of the Abstract Test Suites for normal scenario 71 into Executable Test Suites is shown in Table 4.11. Similarly, an Executable Test Case consists of a set of *test input data/expected result* and it is delimited by left and right braces. However, since *actions* can also be present in Executable Test Cases then such test cases provide a clear correspondence with the sequence of test steps of the testing procedures which are, in fact, the instructions so that Executable Test Cases can be executed. Therefore, the term "test step" will be used to represent both the *test input data/expected result* of an Executable Test Case, meaning that the IUT must be stimulated by such input data and should be expected the indicated result, as to represent an *action* that must be performed.

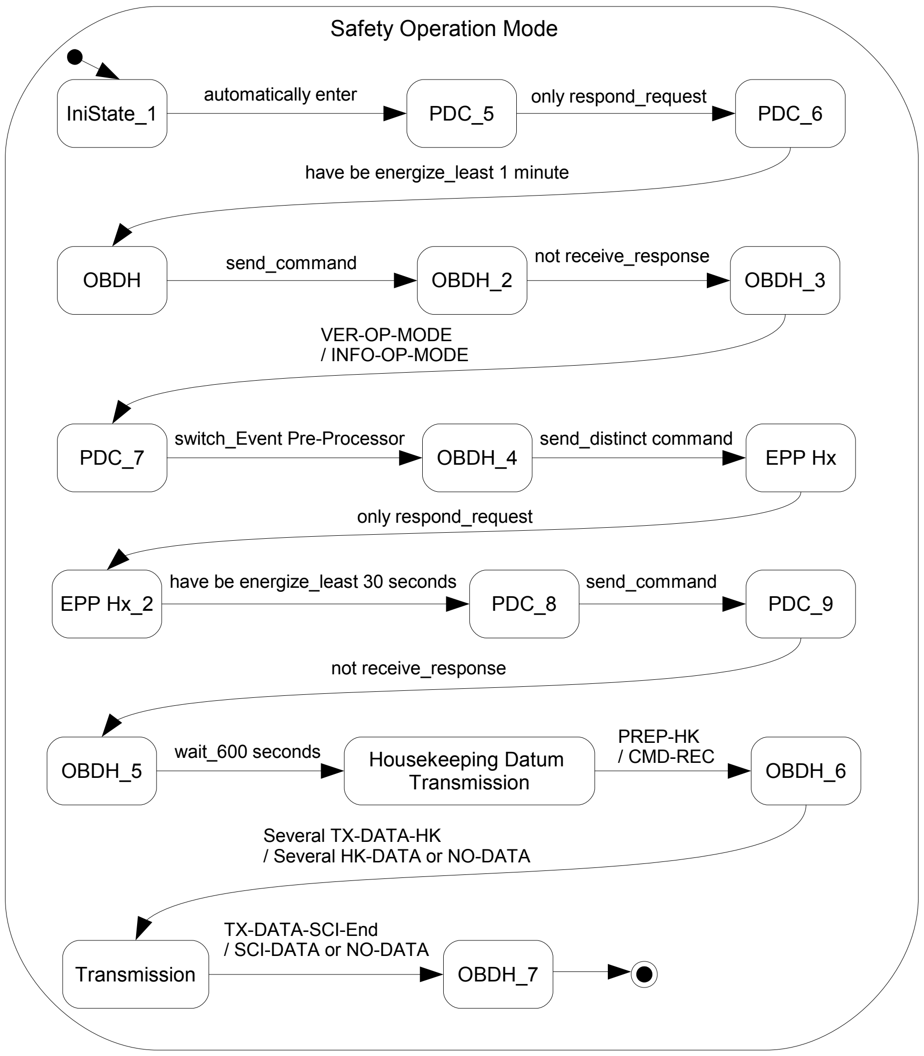


Figure 4.5 - Normal scenario 71: COMPOSITE state *Safety Operation Mode*

Action1, *Action2*, *Action3*, *Action4*, and *Action5* are defined in Tables 4.9 and 4.10. *Action6* is the translation of the abstract *input/output* pair *be#then accomplish_-post#present_irrecoverable problem#/null*. Hence, *Action6* directs the test designer to simulate an irrecoverable problem during the intitiation (initialization) process of PDC and realize whether PDC remains in the *Initiation Operation Mode*, as presented in Figure 4.2. *Action7* is to switch PDC off. The idea then is just to substitute one or more *input/output* pairs of the Abstract Test Suite for the corresponding *test input data/expected result* or *actions* of the Executable Test Suite as shown in Tables 4.9 and 4.10. Translations into other Executable Test Suites due to other test criteria follow the same principles.

Table 4.8 - Abstract Test Suites for normal scenario 71

Test Criterion	**Abstract Test Cases**
all-transitions	{be power on_Power Conditioning Unit/null, be#then accomplish_-post#present_irrecoverable problem#/null}, {be power on_Power Conditioning Unit/null, not be propagate/null, do not present_irrecoverable problem/null, automatically enter/null, only respond_request/null, have be energize_least 1 minute/null, send_command/null, not receive_response/null, VER-OP-MODE/INFO-OP-MODE, switch_Event Pre-Processor/null, send_distinct command/null, only respond_request/null, have be energize_least 30 seconds/null, send_command/null, not receive_response/null, wait_600 seconds/null, PREP-HK/CMD-REC, Several TX-DATA-HK/Several HK-DATA or NO-DATA, TX-DATA-SCI-End/SCI-DATA or NO-DATA, CH-OP-MODE-NOMINAL/CMD-REC, VER-OP-MODE/INFO-OP-MODE, wait_10 seconds/null, obtain and handle_scientific datum/null, also accept_scientific datum transmission request/null, CH-OP-MODE-SAFETY/CMD-REC, be/null, switch_Event Pre-Processor/null, send_distinct command/null}, {switch/null}
all-simple-paths	{be power on_Power Conditioning Unit/null, not be propagate/null, do not present_irrecoverable problem/null, automatically enter/null, only respond_-request/null, have be energize_least 1 minute/null, send_command/null, not receive_response/null, VER-OP-MODE/INFO-OP-MODE, switch_-Event Pre-Processor/null, send_distinct command/null, only respond_request/null, have be energize_least 30 seconds/null, send_command/null, not receive_response/null, wait_600 seconds/null, PREP-HK/CMD-REC, Several TX-DATA-HK/Several HK-DATA or NO-DATA, TX-DATA-SCI-End/SCI-DATA or NO-DATA, CH-OP-MODE-NOMINAL/CMD-REC, VER-OP-MODE/INFO-OP-MODE, wait_10 seconds/null, obtain and handle_scientific datum/null, also accept_scientific datum transmission request/null, CH-OP-MODE-SAFETY/CMD-REC, be/null, switch_Event Pre-Processor/null, send_distinct command/null}, {switch/null}

(Continues)

Table 4.8 - Conclusion

Test Criterion	Abstract Test Cases
all-paths-k-C0	{be power on_Power Conditioning Unit/null, be#then accomplish_-post#present_irrecoverable problem#/null, not be propagate/null, do not present_irrecoverable problem/null, automatically enter/null, only respond_-request/null, have be energize_least 1 minute/null, send_command/null, not receive_response/null, VER-OP-MODE/INFO-OP-MODE, switch_-Event Pre-Processor/null, send_distinct command/null, only respond_re-quest/null, have be energize_least 30 seconds/null, send_command/null, not receive_response/null, wait_600 seconds/null, PREP-HK/CMD-REC, Several TX-DATA-HK/Several HK-DATA or NO-DATA, TX-DATA-SCI-End/SCI-DATA or NO-DATA, CH-OP-MODE-NOMINAL/CMD-REC, VER-OP-MODE/INFO-OP-MODE, wait_10 seconds/null, obtain and han-dle_scientific datum/null, also accept_scientific datum transmission re-quest/null, CH-OP-MODE-SAFETY/CMD-REC, be/null, switch_Event Pre-Processor/null, send_distinct command/null}

Table 4.9 - Examples of the translation from the Abstract Test Suite into the Executable Test Suite. Caption: Abs = Abstract

Abs Input/Output	Test Input Data	Expected Result	Action
be power on_Power Conditioning Unit/null	Action1		Action1: Switch PDC on
not be propagate/null + do not present_irrecoverable problem/null + automatically enter/null	Action2		Action2: Realize whether the PDC is in the Safety Operation Mode after the initiation process
only respond_request/null + have be energize_least 1 minute/null + send_command/null + not receive_response/null	VER-OP-MODE Action3	Timeout	OBDH shall send any command within less than 1 minute since the PDC has been energized. Action3: OBDH should wait 60 seconds to send a command

(Continues)

Table 4.9 - Conclusion

Abs Input/Output	Test Input Data	Expected Result	Action
VER-OP-MODE/INFO-OP-MODE	VER-OP-MODE	INFO-OP-MODE	OBDH shall send this command
only respond_request/null + have be energize_least 30 seconds/null + send_command/null + not receive_response/null	PREP-TST Action4 TX-DATA-TST TX-DATA-TST TX-DATA-SCI	CMD-REC NO-DATA NO-DATA NO-DATA	OBDH shall send/perform all these commands/action within less than 30 seconds since each EPP Hx has been energized. Hence, PDC will try to get Test (TST) Data from each EPP Hx but a timeout will occur in the PDC side of the communication. The result is that no Test Data frame (NO-DATA) will be sent to the OBDH. Action4: OBDH should wait 10 seconds to send a command

Table 4.10 - More examples of the translation from the Abstract Test Suite into the Executable Test Suite. Caption: Abs = Abstract

Abs Input/Output	Test Input Data	Expected Result	Action
wait_600 seconds/null	Action5		Action5: OBDH should wait 600 seconds to send a command
TX-DATA-SCI-End/SCI-DATA or NO-DATA	TX-DATA-SCI	NO-DATA	OBDH shall send this command
wait_10 seconds/null	Action4		Action4: OBDH should wait 10 seconds to send a command

(Continues)

Table 4.10 - Conclusion

Abs Input/Output	**Test Input Data**	**Expected Result**	**Action**
obtain and handle_scientific datum/null + also accept_scientific datum transmission request/null	TX-DATA-SCI	SCI-DATA	After waiting for 10 seconds, the OBDH can send this command. In case that many Scientific Data frames shall be requested, this process is repeated as many times as needed (wait 10 seconds, send the command)

Table 4.11 - Executable Test Suites translated from the Abstract Test Suites for normal scenario 71

Test Criterion	**Executable Test Cases**
all-transitions	{Action1, Action6}, {Action1, Action2, VER-OP-MODE/Timeout, Action3, VER-OP-MODE/INFO-OP-MODE, ACT-HW-EPPH1ON/CMD-REC, ACT-HW-EPPH2ON/CMD-REC, PREP-TST/CMD-REC, Action4, TX-DATA-TST/NO-DATA, TX-DATA-TST/NO-DATA, TX-DATA-SCI/NO-DATA, Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA, CH-OP-MODE-NOMINAL/CMD-REC, VER-OP-MODE/INFO-OP-MODE, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, CH-OP-MODE-SAFETY/CMD-REC, VER-OP-MODE/INFO-OP-MODE, ACT-HW-EPPH1OFF/CMD-REC, ACT-HW-EPPH2OFF/CMD-REC}, {Action7}

(Continues)

Table 4.11 - Conclusion

Test Criterion	**Executable Test Cases**
all-simple-paths	{Action1, Action2, VER-OP-MODE/Timeout, Action3, VER-OP-MODE/INFO-OP-MODE, ACT-HW-EPPH1ON/CMD-REC, ACT-HW-EPPH2ON/CMD-REC, PREP-TST/CMD-REC, Action4, TX-DATA-TST/NO-DATA, TX-DATA-TST/NO-DATA, TX-DATA-SCI/NO-DATA, Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA, CH-OP-MODE-NOMINAL/CMD-REC, VER-OP-MODE/INFO-OP-MODE, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, CH-OP-MODE-SAFETY/CMD-REC, VER-OP-MODE/INFO-OP-MODE, ACT-HW-EPPH1OFF/CMD-REC, ACT-HW-EPPH2OFF/CMD-REC}, {Action7}
all-paths-k-C0	{Action1, Action6, Action2, VER-OP-MODE/Timeout, Action3, VER-OP-MODE/INFO-OP-MODE, ACT-HW-EPPH1ON/CMD-REC, ACT-HW-EPPH2ON/CMD-REC, PREP-TST/CMD-REC, Action4, TX-DATA-TST/NO-DATA, TX-DATA-TST/NO-DATA, TX-DATA-SCI/NO-DATA, Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA, CH-OP-MODE-NOMINAL/CMD-REC, VER-OP-MODE/INFO-OP-MODE, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, Action4, TX-DATA-SCI/SCI-DATA, CH-OP-MODE-SAFETY/CMD-REC, VER-OP-MODE/INFO-OP-MODE, ACT-HW-EPPH1OFF/CMD-REC, ACT-HW-EPPH2OFF/CMD-REC}

The concrete *test input data/expected result* of an Executable Test Case is obtained just by replacing the abbreviations of commands/responses with the values as specified in the PDC-OBDH Communication Protocol Specification. For instance, the first VER-OP-MODE/INFO-OP-MODE of the second Executable Test Case of the all-transitions Executable Test Suite, of the first Executable Test Case of the all-simple-paths Executable Test Suite, and of the first (single) Executable Test Case of the all-paths-k-C0-configuration Executable Test Suite (Table 4.11) is substituted for (all values are in hexadecimal):

- VER-OP-MODE: EB 80 00 00 00 00 04 00 00 00 FE 91;
- INFO-OP-MODE: EB 20 XX XX XX XX 83 00 00 01 01 XX XX.

In the expected result, XX represents values that are very difficult to predict in advance such as the time stamp that indicates the exact value of the clock of the PDC computer when PDC is assembling a response to be sent back to OBDH. In these cases, it is not necessary to worry about these values.

4.1 Comparing the SOLIMVA methodology with an expert's approach

This section compares the models and Executable Test Cases generated by version 1.0 of the SOLIMVA methodology and by the SOLIMVA tool with others manually generated by a test designer who is an expert in the SWPDC software product. The 20 scenarios and their respective models generated by the expert can be seen in SANTIAGO JÚNIOR et al. (2010). The purposes of such a comparison are twofold: (i) to know if the SOLIMVA methodology is able to cover the test objectives of the set of scenarios manually developed by an expert with respect to a certain IUT; (ii) to verify whether the Executable Test Cases generated according to the SOLIMVA methodology have the same characteristics of the test cases developed according to the expert's models.

4.1.1 Coverage of test objectives

The results of the comparison to check the coverage of test objectives are shown in Tables 4.12 and 4.13. In both tables, the leftmost column shows the *Test Objectives* associated with a scenario, *Expert* shows the number of the manually generated scenario, and the *SOLIMVA* column shows the corresponding factor combinations whose interpretations derive the scenarios which have associated the same test objectives of the manual (expert) scenarios. The number of the scenario in the *SOLIMVA* column refers to those generated only by the main execution of the combinatorial designs algorithm, i.e. one that resulted in 175 normal scenarios.

Table 4.12 - Scenarios 1 to 10 of the expert: a comparison between the expert's approach and SOLIMVA from the perspective of coverage of test objectives

Test Objectives	**Expert**	**SOLIMVA**
PDC initiation process	1	{OpMMgm, Init, -, -} = 43, 44, 45, 46, 47, 48, 49
Switching EPP Hxs on and off	2	{HwSwHnd, Safe, -, -} = 15, 16, 17, 18, 19, 20, 21
Changing software parameters in the Safety Operation Mode	3	{HwSwHnd, Safe, Hk, -} = 16
Processes of Power On and Reset	4	{HwSwHnd, Safe, -, PwrOn} = 16 {HwSwHnd, Safe, -, Reset} = 17, 18 {OpMMgm, Safe, -, PwrOn} = 50, 53, 54, 55, 56 {OpMMgm, Safe, -, Reset} = 52
Scientific Data Acquisition and Transmission in the Nominal Operation Mode	5	{DtAcqTx, Nom, Sci, -} = 71
Scientific Data Acquisition and Transmission in the Nominal Operation Mode, Robustness (command)	6	{DtAcqTx, Nom, Sci, -} = 71 {Inv, Nom, Sci, -} = 141
Houseekeeping Data Transmission in the Nominal Operation Mode	7	{DtAcqTx, Nom, Hk, -} = 72
Houseekeeping Data Transmission in the Nominal Operation Mode, Robustness (reception), Load new programs	8	{DtAcqTx, Nom, Hk, -} = 72 {PrLoad, Nom, Load, -} = 109 {Inv, Nom, Hk, -} = 142
Dump Data of Program Memory in the Nominal Operation Mode	9	{DtAcqTx, Nom, Dmp, -} = 73
Dump Data of Program Memory in the Nominal Operation Mode, Robustness (command and reception)	10	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143

Table 4.13 - Scenarios 11 to 20 of the expert: a comparison between the expert's approach and SOLIMVA from the perspective of coverage of test objectives

Test Objectives	**Expert**	**SOLIMVA**
Dump Data of Data Memory (Pages 0 - 3) in the Nominal Operation Mode	11	Bad strategy
Dump Data of Data Memory (Page 0) in the Nominal Operation Mode, Robustness (command and reception)	12	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143
Dump Data of Data Memory (Page 1) in the Nominal Operation Mode, Robustness (command and reception)	13	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143
Dump Data of Data Memory (Page 2) in the Nominal Operation Mode, Robustness (command and reception)	14	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143
Dump Data of Data Memory (Page 3) in the Nominal Operation Mode, Robustness (command and reception)	15	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143
Dump Data of Data Memory (Pages 4 - 7) in the Nominal Operation Mode	16	Bad strategy
Dump Data of Data Memory (Page 4) in the Nominal Operation Mode, Robustness (command and reception)	17	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143
Dump Data of Data Memory (Page 5) in the Nominal Operation Mode, Robustness (command and reception)	18	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143

(Continues)

Table 4.13 - Conclusion

Test Objectives	Expert	SOLIMVA
Dump Data of Data Memory (Page 6) in the Nominal Operation Mode, Robustness (command and reception)	19	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143
Dump Data of Data Memory (Page 7) in the Nominal Operation Mode, Robustness (command and reception)	20	{DtAcqTx, Nom, Dmp, -} = 73 {Inv, Nom, Dmp, -} = 143

The first goal of the comparison was achieved. All test objectives of the expert's scenarios were satisfactorily covered by SOLIMVA's scenarios. Observe that a single expert's scenario can be addressed by more than one corresponding SOLIMVA's scenario. For instance, the test objective associated with expert's scenario 1 can be addressed by any of the following SOLIMVA's scenarios: 43, 44, 45, 46, 47, 48, 49. Besides, a factor combination may be interpreted in different ways. In Table 4.12, three different test objectives are related to factor combination 16: "Switching EPP Hxs on and off", "Changing software parameters in the Safety Operation Mode", and "Process of Power On". It is up to the test designer to choose one of these test objectives for SOLIMVA's scenario 16, and leave the others to be addressed by interpreting other factor combinations.

Besides covering all the test objectives, SOLIMVA's philosophy also pointed out some problems in the expert's strategy. Expert's scenario 4 could be broken into two different scenarios: one addressing the process of power on and another related to the reset of PDC. As expert's scenario 4 can be mapped to four distinct groups of scenarios in SOLIMVA, this demonstrates that the test objectives of the expert were confused trying to deal with many different aspects into a single scenario. Figure 4.6 shows the main Statechart model derived by the expert for scenario 4. This model is an AND state with three substates: *Initiation*, *Timing*, and *PowerState*. Let the sequence of transitions be (only input events and *In State* conditions are shown): {*switchPDCOn, POSTOk, tsinc, VER_OP_MODE[In (Power)]*}, where *tsinc* is an internal event. After such a sequence of transitions, the COMPOSITE state *SafeM_-PowerVer* is entered and, within it, all behavior modeling activities related to the process of switching PDC on (power on process) is considered.

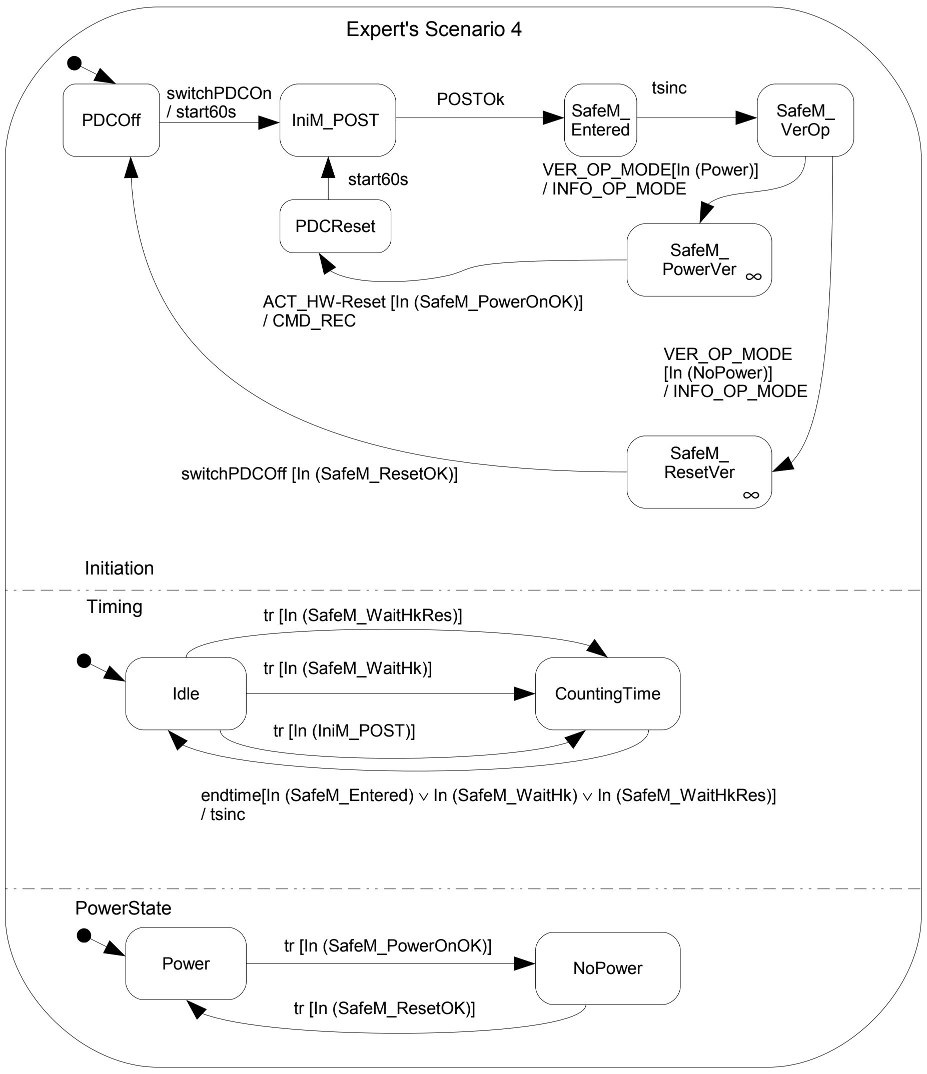

Expert's Scenario 4
PDCOff
switchPDCOn / start60s
IniM_POST
POSTOk
SafeM_ Entered
tsinc
SafeM_ VerOp
start60s
PDCReset
VER_OP_MODE[In (Power)] / INFO_OP_MODE
SafeM_ PowerVer ∞
ACT_HW-Reset [In (SafeM_PowerOnOK)] / CMD_REC
VER_OP_MODE [In (NoPower)] / INFO_OP_MODE
SafeM_ ResetVer ∞
switchPDCOff [In (SafeM_ResetOK)]
Initiation
Timing
tr [In (SafeM_WaitHkRes)]
tr [In (SafeM_WaitHk)]
Idle
CountingTime
tr [In (IniM_POST)]
endtime[In (SafeM_Entered) ∨ In (SafeM_WaitHk) ∨ In (SafeM_WaitHkRes)] / tsinc
PowerState
Power
tr [In (SafeM_PowerOnOK)]
NoPower
tr [In (SafeM_ResetOK)]

Figure 4.6 - Expert's scenario 4: main Statechart model
SOURCE: Adapted from SANTIAGO JÚNIOR et al. (2010)

After the verification of power on, this second sequence of transitions {*ACT_-HW-Reset[In (SafeM_PowerOnOk)], start60s, POSTOk, tsinc, VER_OP_MODE[In (NoPower)]*} allows the system to enter into the COMPOSITE state *SafeM_ResetVer* in order to verify whether the SWPDC performed correctly the tasks related to the reset process. A better approach is to separate these two test objectives into two distinct scenarios. SOLIMVA's normal scenario 50 (Figures 4.7, 4.8) addresses the power on process while SOLIMVA's normal scenario 17 covers the reset process (Figures 4.9, 4.10, 4.11).

The first difference among the models for expert's scenario 4 and those for

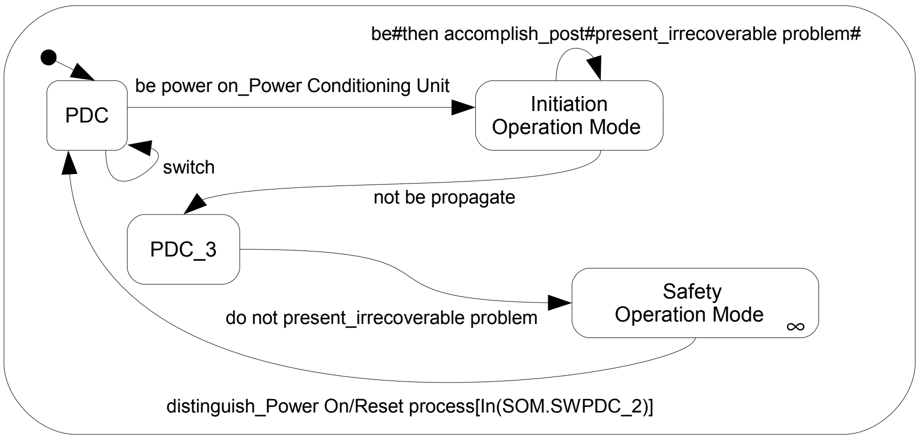

be#then accomplish_post#present_irrecoverable problem#
PDC
be power on_Power Conditioning Unit
Initiation Operation Mode
switch
not be propagate
PDC_3
Safety Operation Mode
do not present_irrecoverable problem
distinguish_Power On/Reset process[In(SOM.SWPDC_2)]

Figure 4.7 - SOLIMVA's normal scenario 50: main Statechart model

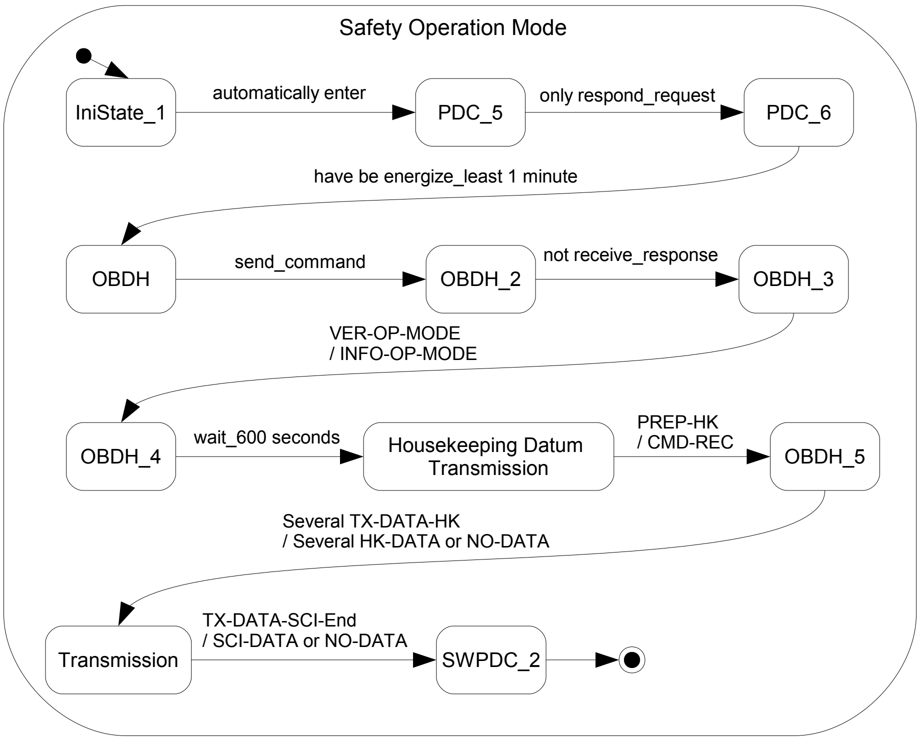


Figure 4.8 - SOLIMVA's normal scenario 50: *Safety Operation Mode*

SOLIMVA's scenarios 50 and 17 is the lack of orthogonality (parallelism) in the models generated by the SOLIMVA tool as explained in Section 3.3.2.3. Actually, all main Statechart models of all expert's scenarios are AND states and hence orthogonality is a characteristic of the expert's modeling. Besides, the expert's model missed one important transition that represents severe problems during the PDC initiation (initialization) process (the self transition in the *Initiation Operation Mode* state in Figures 4.7, 4.9).

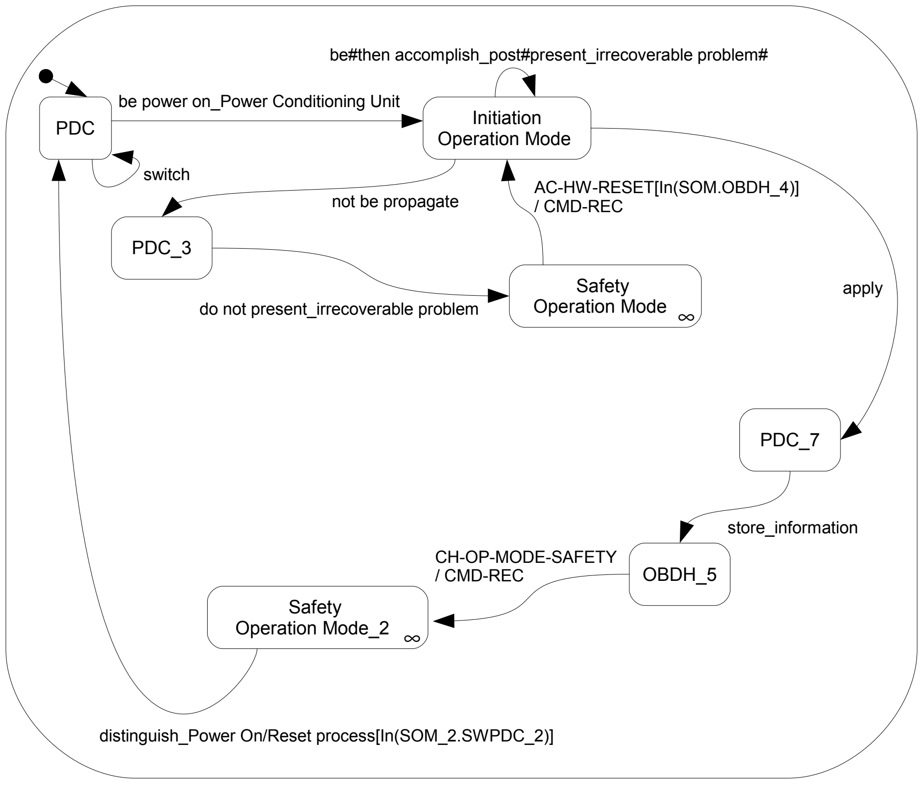

be#then accomplish_post#present_irrecoverable problem#
be power on_Power Conditioning Unit
PDC
Initiation Operation Mode
switch
not be propagate
AC-HW-RESET[In(SOM.OBDH_4)] / CMD-REC
PDC_3
Safety Operation Mode
apply
do not present_irrecoverable problem
PDC_7
store_information
CH-OP-MODE-SAFETY / CMD-REC
OBDH_5
Safety Operation Mode_2
distinguish_Power On/Reset process[In(SOM_2.SWPDC_2)]

Figure 4.9 - SOLIMVA's normal scenario 17: main Statechart model

The separation of test objectives into two different scenarios provides a clear benefit in terms of strategy. Models generated for SOLIMVA's normal scenario 50, which have associated the power on verification, are very simple models. Checking whether the processes of power on and reset were successful is by means of Housekeeping Data where SWPDC adds information to form a log. Hence, in the models for SOLIMVA's normal scenario 50, the *Safety Operation Mode* COMPOSITE state contains behavior to transmit Housekeeping Data, i.e. the sequence of transitions {*wait_600 seconds, PREP-HK, Several TX-DATA-HK, TX-DATA-SCI-End*} in Figure 4.8. However, for reset process addressed by SOLIMVA's normal scenario 17, the *Safety Operation Mode* COMPOSITE state has no such behavior because right after the initiation (initialization) process, the system shall be reset (*AC-HW-RESET[In(SOM.OBDH_4)]* in Figure 4.9) returning to the *Initiation Operation Mode* state. The behavior to

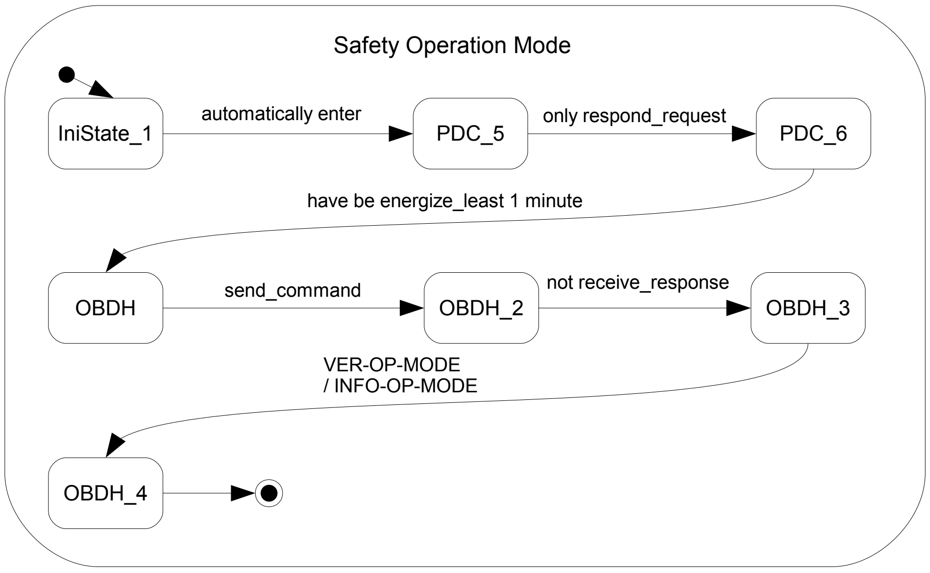


Figure 4.10 - SOLIMVA's normal scenario 17: *Safety Operation Mode*

transmit Housekeeping Data appears in the *Safety Operation Mode_2* COMPOSITE state. Again, these differences favor simplicity. In order to contemplate these characteristics, the expert's models for scenario 4 were created with three hierarchy levels, say the main, the second and the third levels, adding complexity to understand such models.

Expert's scenario 8 is another issue. In this case, the test objectives aimed at two completely and unrelated goals such as Houseekeeping Data transmission and loading new programs on the fly into PDC's Data Memory, and a third goal related to the robustness of SWPDC in situations where a command is not entirely received by PDC (reception). Loading new programs on the fly is a complex process in which the entire executable code is substituted for a new one during satellite operation. Figure 4.12 shows the the *NomM_HkData* COMPOSITE state developed by the expert for scenario 8 where the three test objectives are addressed. The sequence of transitions {*start600s, tsinc, TX_DATA-Hk, PREP_HK[In(Idle)], TX_DATA-Hk, TX_DATA-Sci*} addresses the test objective "Housekeeping Data Transmission in the Nominal Operation Mode". This sequence of transitions is similar to ones presented in some SOLIMVA's COMPOSITE states (Figures 4.5, 4.8, 4.11). In other words, to receive Housekeeping Data from PDC, the OBDH should wait 600 seconds (*start600s*). After that a command to PREPARE HOUSEKEEPING DATA (*PREP_HK*) shall be sent to PDC followed by one or more commands to

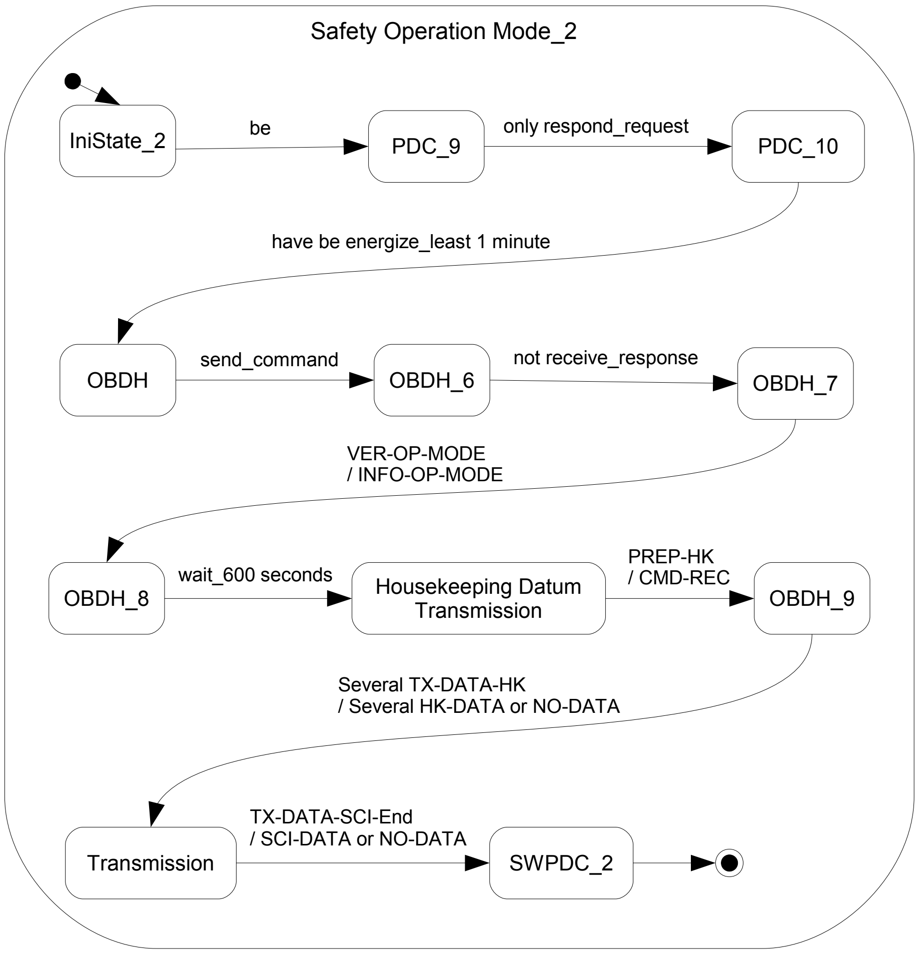


Figure 4.11 - SOLIMVA's normal scenario 17: *Safety Operation Mode_2*

TRANSMIT HOUSEKEEPING DATA (*TX_DATA-Hk*) and, to finish this process, the last command shall be TRANSMIT SCIENTIFIC DATA (*TX_DATA-Sci*).

The test objective "Robustness addressing problem of incomplete reception of commands (reception)" is modeled by the following sequence of transitions: {*start60s, tsinc, PREP_HK[In(Idle)], TX_DATA-Hk, RET_ANSW[In(Idle)]*}. Note the expected result *timeout* due to the test input data TRANSMIT HOUSEKEEPING DATA (*TX_DATA-Hk*). The test designer assumed that a problem, probably in the physical transmission medium, occurred and only part of the command was received by SWPDC. The OBDH can then send a command to RETRANSMIT THE LAST DATA RESPONSE (*RET_ANSW*). In this case, the reception of this last command is successful but the OBDH will not receive Housekeeping Data as indicated in the expert's model (expected result *HK_DATA-RSC0* in Figure 4.12). This model has a

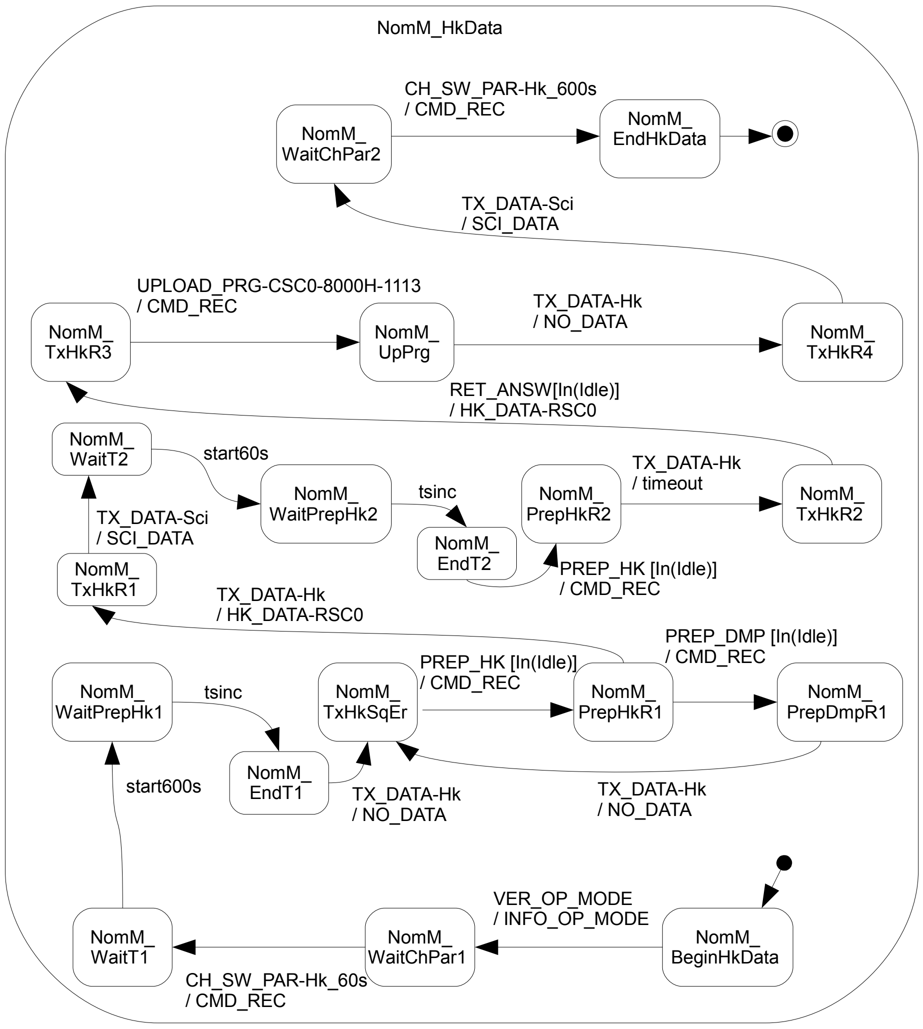


Figure 4.12 - Expert's scenario 8: *NomM_HkData*
SOURCE: Adapted from SANTIAGO JÚNIOR et al. (2010)

minor defect. To explain this fact, consider the following NL requirements, one from the PDC-OBDH Communication Protocol Specification (POCP021) and the other from the Software Requirements Specification (SRS013):

- POCP021 - The PDC may not receive a command sent in its entirety. After identifying the beginning of a command frame, the PDC shall wait two times MAX-TRANSM-DELAY for the rest of the command. If this stipulated time expires, a timeout shall occur, the PDC shall abort the communication, the command shall be discarded, an event report shall be generated, and the PDC shall wait for a new OBDH's command.

- SRS013 - The SWPDC shall always maintain temporarily stored the last data response sent to the OBDH because the OBDH can demand the retransmission of this last data response.

Since the command to TRANSMIT HOUSEKEEPING DATA (*TX_DATA-Hk*) is not completely received by the PDC, the response for *RET_ANSW* is the last data response sent to the OBDH which is Scientific Data (note the transition *TX_DATA-Sci/SCI_DATA* from state *NomM_TxHkR1* to state *NomM_WaitT2*). Thus, the correct sequence of *test input data/expected result* to address this test objective should be something like this: {*start60s, tsinc, PREP_HK[In(Idle)]/CMD_REC, TX_DATA-Hk/timeout, RET_ANSW[In(Idle)]/SCI_DATA, PREP_HK/CMD_-REC, TX_DATA-Hk/HK_DATA-RSC0, TX_DATA-Sci/SCI_DATA*}.

The test objective "Loading new programs on the fly into PDC's Data Memory" is covered by the following transition: *UPLOAD_PRG-CSC0-8000H-1113*. This single command represents, in fact, all the commands to load a small new program into PDC's Data Memory (a concise representation was used in this case). And as mentioned earlier, the combination of these three test objectives makes expert's scenario 8 very confusing and inadequate.

The SOLIMVA methodology will certainly drive the test designer to separate these three test objectives leading to a more coherent solution. For the test objective "Housekeeping Data Transmission in the Nominal Operation Mode", SOLIMVA's normal scenario 72 can be used. Figure 4.13 shows the COMPOSITE state *Nominal Operation Mode* of normal scenario 72. The following sequence of transitions is related to this test objective: {*wait_600 seconds, PREP-HK, stop_scientific datum acquisition, Several TX-DATA-HK, TX-DATA-SCI-End*}.

SOLIMVA's normal scenario 142 can be used to cover "Robustness addressing problem of incomplete reception of commands (reception)". Figure 4.14 shows the COMPOSITE state *Nominal Operation Mode* of this scenario where the following sequence of transitions addresses the test objective: {*wait_600 seconds, PREP-HK, stop_scientific datum acquisition, Several TX-DATA-HK, TX-DATA-SCI-End, not receive_command, wait_two time MAX-TRANSM-DELAY, abort_communication, wait_new OBDH, always maintain temporarily store_last datum response, demand_retransmission*}. Some of these abstract *input/output* pairs were derived from NL requirements POCP021 and SRS013 shown above.

The last test objective, "Loading new programs on the fly into PDC's Data Memory"

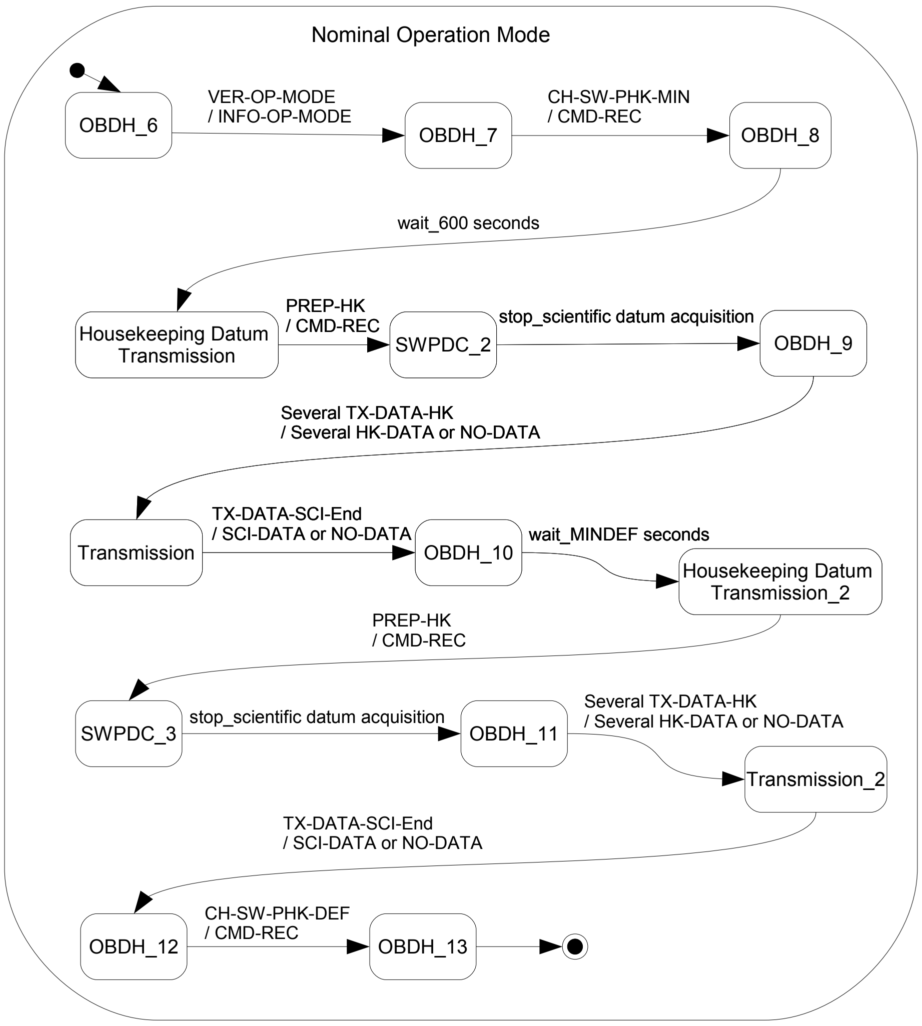

Nominal Operation Mode
OBDH_6
VER-OP-MODE / INFO-OP-MODE
OBDH_7
CH-SW-PHK-MIN / CMD-REC
OBDH_8
wait_600 seconds
Housekeeping Datum Transmission
PREP-HK / CMD-REC
SWPDC_2
stop_scientific datum acquisition
OBDH_9
Several TX-DATA-HK / Several HK-DATA or NO-DATA
Transmission
TX-DATA-SCI-End / SCI-DATA or NO-DATA
OBDH_10
wait_MINDEF seconds
Housekeeping Datum Transmission_2
PREP-HK / CMD-REC
SWPDC_3
stop_scientific datum acquisition
OBDH_11
Several TX-DATA-HK / Several HK-DATA or NO-DATA
Transmission_2
TX-DATA-SCI-End / SCI-DATA or NO-DATA
OBDH_12
CH-SW-PHK-DEF / CMD-REC
OBDH_13

Figure 4.13 - SOLIMVA's normal scenario 72: *Nominal Operation Mode*

is addressed by SOLIMVA's normal scenario 109. However, as mentioned earlier, normal scenario 109 needs to be unfolded and contributes with 6 unfolded scenarios replacing this normal scenario. Table 4.14 shows factors and levels for unfolding normal scenario 109. The primary factor is the initial memory address (IniAdd) so that, from this address, the piece of executable code contained within a data frame (command) can be loaded. Inside each LOAD PROGRAM (*LD-PRG*) command there is the initial memory address.

Command Sequence Control (CSC) is a field of the command frame that indicates the sequence number of a command of a series of commands to be sent and which are related to a certain service supported by SWPDC. This field is primarily used for the service of loading new programs. In short, CSC tells SWPDC which is the number

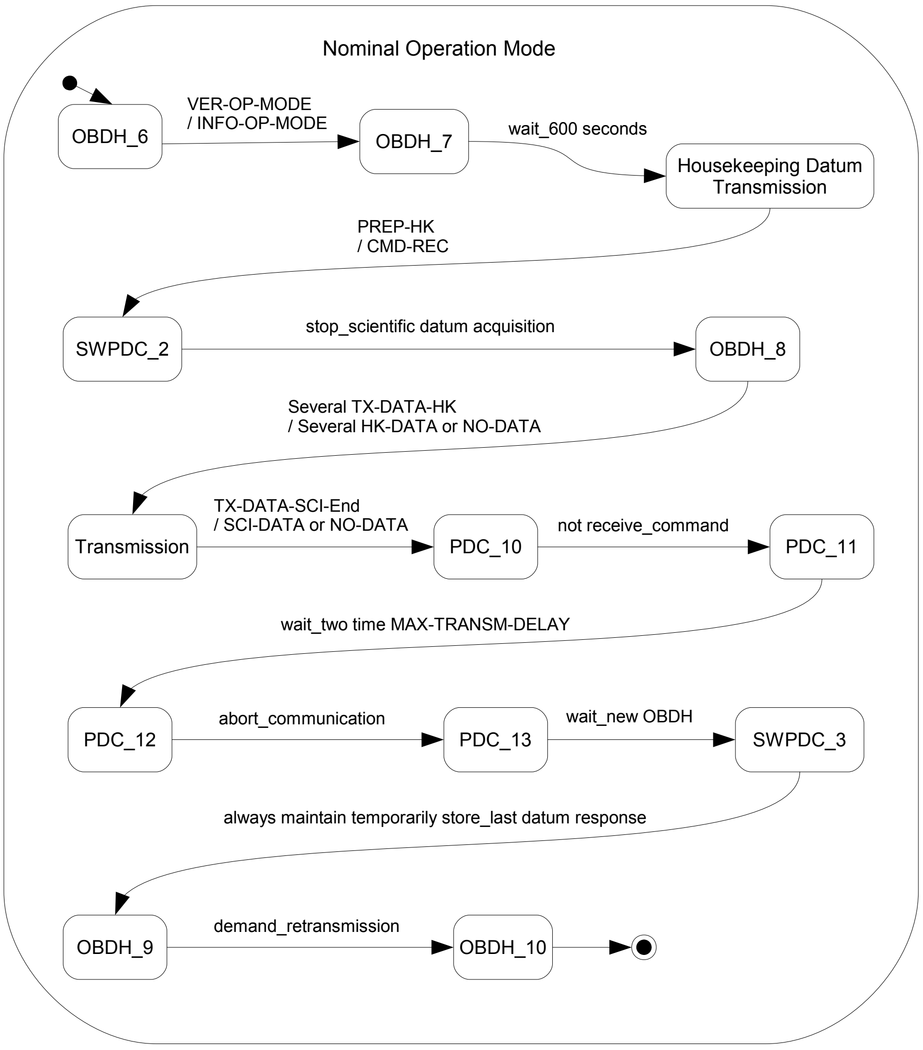

Nominal Operation Mode
OBDH_6
VER-OP-MODE / INFO-OP-MODE
OBDH_7
wait_600 seconds
Housekeeping Datum Transmission
PREP-HK / CMD-REC
SWPDC_2
stop_scientific datum acquisition
OBDH_8
Several TX-DATA-HK / Several HK-DATA or NO-DATA
Transmission
TX-DATA-SCI-End / SCI-DATA or NO-DATA
PDC_10
not receive_command
PDC_11
wait_two time MAX-TRANSM-DELAY
PDC_12
abort_communication
PDC_13
wait_new OBDH
SWPDC_3
always maintain temporarily store_last datum response
OBDH_9
demand_retransmission
OBDH_10

Figure 4.14 - SOLIMVA's normal scenario 142: *Nominal Operation Mode*

Table 4.14 - Unfolding normal scenario 109. Caption: Min/Max = Minimum/Maximum allowed memory address; LessMin/GreatMax = Less/Greater than Minimum/Maximum allowed memory address; InRng = Address between Min and Max (In Range); LessPrev = Less than the last memory address previously loaded with executable code

Factors	**Levels**					
IniAdd	InRng	LessPrev	Min	Max	LessMin	GreatMax
CSC	Ok	Inv				
CKS	Ok	Inv				

of a *LD-PRG* command, and this number varies from N to 0 (this being the last command). So SWPDC checks whether a command received has the correct CSC, that is if after the command whose CSC is N, the next received command has CSC equals to $N - 1$, and so on until the receipt of the last command (CSC = 0). This is a flow control mechanism similar to mechanisms used in data link layer protocols in computer networks. A Checksum (CKS) value shall be sent by the OBDH after all *LD-PRG* commands. Hence, SWPDC checks whether this received checksum is equal to the recalculated checksum after receiving and loading all new executable code into PDC's Data Memory. Only if the recalculated and received checksums are identical is that the executable code will be actually executed.

When running the combinatorial designs algorithm with $strength = 2$, 12 factor combinations are generated which are grouped in 6 unfolded scenarios. Unfolded scenario 109.3 is obtained by the interpretation of the following factor combinations: {Min, Ok, Ok}, {Min, -, Inv}. Thus, this unfolded scenario has two associated test objectives. The first one covers what the expert intended to scenario 8. In other words, the factor combination {Min, Ok, Ok} drives the test designer to add requirements to address the successful operation of loading a new program. Thus, within the selected requirements the initial memory address in the first *LD-PRG* command is the minimum allowed memory address (8000h; new programs are loaded only on page 0 of the Data Memory). If other *LD-PRG* commands are needed, initial memory addresses are all consistent where, for example, no code will be loaded into memory area previously loaded. Moreover, the sequence of CSC related to all *LD-PRG* commands is also coherent and the recalculated and received checksums are identical.

The second test objective comes from the interpretation of {Min, -, Inv}. The observations just made related to initial memory address and CSC are also valid in this second test objective (behavior is the same). The difference is that the recalculated and received checksums are supposed to be unequal (note the *Inv* level) and then the new program is not in fact executed. This test objective relates to Robustness testing.

The set of NL requirements that characterize unfolded scenario 109.3 is shown in Table 4.15. The combinatorial designs direct the test designer to add requirements in accordance with the factor combinations. NL requirement SRS015 was added due to factor combination {Min, Ok, Ok} as well as NL requirements RB008, RB009, RB010, RB011, and RB012 were chosen due to factor combination {Min, -, Inv}.

Figure 4.15 shows the COMPOSITE state *Nominal Operation Mode* for SOLIMVA's unfolded scenario 109.3. The sequence of transitions {*be_capable, STOP-DT-ACQ,*

Table 4.15 - Set of NL requirements that characterize unfolded scenario 109.3. Caption: Req = Requirement; Id = Identification

Req Id	**Req Description**
SRS001	The PDC shall be powered on by the Power Conditioning Unit.
SRS002	The PDC shall be in the Initiation Operation Mode after being powered on. The 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.
SRS003	If PDC does not present any irrecoverable problem, after the initiation process, the PDC shall automatically enter into the Safety Operation Mode.
POCP001	The PDC can only respond to requests (commands) from OBDH after the PDC has been energized for at least 1 minute. If OBDH sends commands within less than 1 minute, the OBDH shall not receive any response from PDC.
RB001	The OBDH shall send VER-OP-MODE to PDC.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH sends distinct commands to perform such actions.
PECP001	Each EPP Hx can only respond to requests (commands) from PDC after each EPP Hx has been energized for at least 30 seconds. If PDC sends commands within less than 30 seconds to a certain EPP Hx, the PDC shall not receive any response from this EPP Hx.
RB003	The OBDH shall send CH-OP-MODE-NOMINAL to PDC.
RB001	The OBDH shall send VER-OP-MODE to PDC.
SRS015	The SWPDC shall be capable to load and execute new programs in the Nominal Operation Mode. First, the OBDH shall send STOP-DT-ACQ so that the SWPDC can interrupt the acquisition of scientific data. After this command, the PDC is expected to receive LD-PRG-MIN-OK and, after that, the OBDH shall send Several LD-PRG-INRNG-OK. Then, the OBDH shall submit EXEC-CKS-OK to PDC so that the loaded program is executed. At the end of the process of loading and executing a new program, the OBDH shall send RSTART-DATA-ACQ to PDC so that PDC can restart acquiring scientific data from EPP Hxs.
RB008	The OBDH shall send STOP-DT-ACQ to PDC.
RB009	The OBDH shall send LD-PRG-MIN-OK to PDC.
RB010	The OBDH shall send Several LD-PRG-INRNG-OK to PDC.
RB011	The OBDH shall send EXEC-CKS-INV to PDC.
RB012	The OBDH shall send RSTART-DATA-ACQ to PDC.
RB004	The OBDH shall send CH-OP-MODE-SAFETY to PDC. After that, the PDC shall be in the Safety Operation Mode.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH sends distinct commands to perform such actions.
RB005	After switching both EPPHxs off via PDC, the OBDH shall switch the PDC off via the Power Conditioning Unit.

interrupt_acquisition, LD-PRG-MIN-OK, Several LD-PRG-INRNG-OK, EXEC-CKS-OK, RSTART-DATA-ACQ, restart acquire_scientific datum} addresses the test objective of successful loading which was derived from the interpretation of factor combination {Min, Ok, Ok}. This other sequence of transitions {*STOP-DT-ACQ, LD-PRG-MIN-OK, Several LD-PRG-INRNG-OK, EXEC-CKS-INV, RSTART-DATA-ACQ*} is related to the test objective which realizes what is the behavior of SWPDC when the recalculated and received checksums are different and thus the interpretation of factor combination {Min, -, Inv} was responsible for this fact.

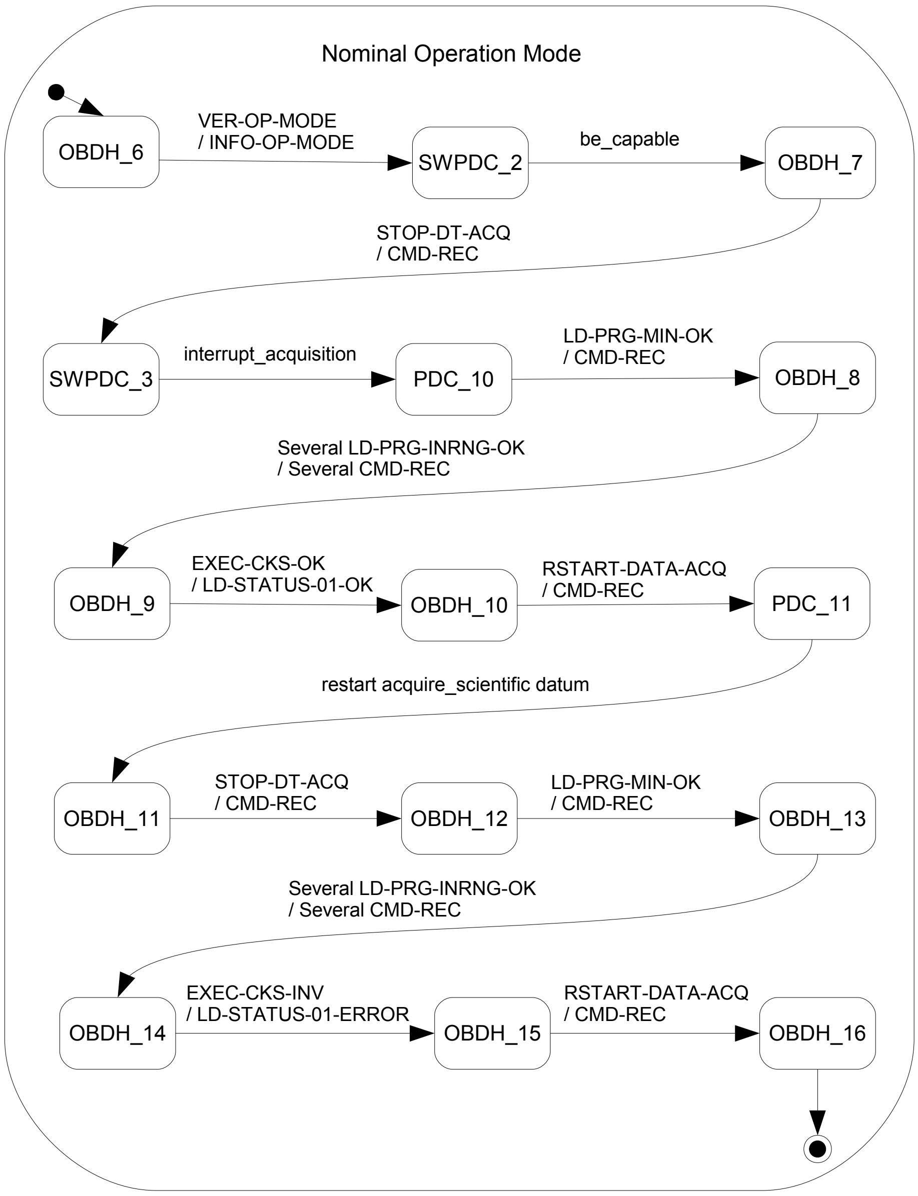

Nominal Operation Mode
OBDH_6
VER-OP-MODE / INFO-OP-MODE
SWPDC_2
be_capable
OBDH_7
STOP-DT-ACQ / CMD-REC
SWPDC_3
interrupt_acquisition
PDC_10
LD-PRG-MIN-OK / CMD-REC
OBDH_8
Several LD-PRG-INRNG-OK / Several CMD-REC
OBDH_9
EXEC-CKS-OK / LD-STATUS-01-OK
OBDH_10
RSTART-DATA-ACQ / CMD-REC
PDC_11
restart acquire_scientific datum
OBDH_11
STOP-DT-ACQ / CMD-REC
OBDH_12
LD-PRG-MIN-OK / CMD-REC
OBDH_13
Several LD-PRG-INRNG-OK / Several CMD-REC
OBDH_14
EXEC-CKS-INV / LD-STATUS-01-ERROR
OBDH_15
RSTART-DATA-ACQ / CMD-REC
OBDH_16

Figure 4.15 - SOLIMVA's unfolded scenario 109.3: *Nominal Operation Mode*

The model shown in Figure 4.15 is not fully coherent with the actual behavior related to the loading of new programs. But the problem is not due to the SOLIMVA methodology/SOLIMVA tool, but that is because the Software Requirements Specification was not updated, and it is therefore incomplete in relation to the final version of the SWPDC software product. So this is a problem of requirements evolution/traceability. Since this specification is a key reference document for the use of the methodology, then this justifies the incoherence. SOLIMVA's normal scenarios 72, 142 and unfolded scenario 109.3 all have a main Statechart model as well as COMPOSITE states *Safety Operation Mode* and *Safety Operation Mode_2*. However, these models are quite similar to the ones generated for normal scenario 71 (Figures 4.2, 4.5, 4.4).

In Table 4.13, expert's scenarios 11 and 16 are "marked" as Bad strategy. Indeed, the test objective related to expert's scenario 11 is covered by expert's scenarios 12 to 15. Similarly, the test objective associated with expert's scenario 16 is addressed by expert's scenarios 17 to 20. In case of the SOLIMVA methodology, the Dump Data service proposed in expert's scenarios 9 to 20 is covered by unfolding normal scenario 73. Continuing with the comparison between the expert's approach and SOLIMVA, consider expert's scenario 12 with the following test objectives: "Dump Data of Data Memory (Page 0) in the Nominal Operation Mode", "Robustness taking into account problems (inconsistent values) within commands sent to PDC (command)", and "Robustness addressing problem of incomplete reception of commands (reception)". The models related to expert's scenario 12 are shown in Figures 4.16, 4.17.

In the COMPOSITE state *NomM_DmpPg0* (Figure 4.17), the sequence of transitions {*P_DMP-DataP0-7FFFH-BFFFH, TX_DATA-Dmp*} relates to the test objective "Robustness taking into account problems (inconsistent values) within commands sent to PDC (command)". *P_DMP-DataP0-7FFFH-BFFFH* is an abbreviation for the following command defined in the PDC-OBDH Communication Protocol Specification: PREPARE DUMP DATA FROM PAGE 0 OF DATA MEMORY, WITH INITIAL ADDRESS = 7FFFH AND FINAL ADDRESS = BFFFH. However, as shown in Figure 4.1, the minimum memory address allowed for all pages of the Data Memory is 8000h. Thus, this command must not be processed by SWPDC due to the incorrect value of the initial address, i.e. lower than the minimum memory address allowed, and the expected result is a *timeout*. If a TRANSMIT DUMP DATA (*TX_DATA-Dmp*) is then sent, SWPDC must respond NO DATA (*NO_DATA*).

The sequence of transitions {*P_DMP-DataP0-8000H-BFFFH, TX_DATA-Dmp_14-*

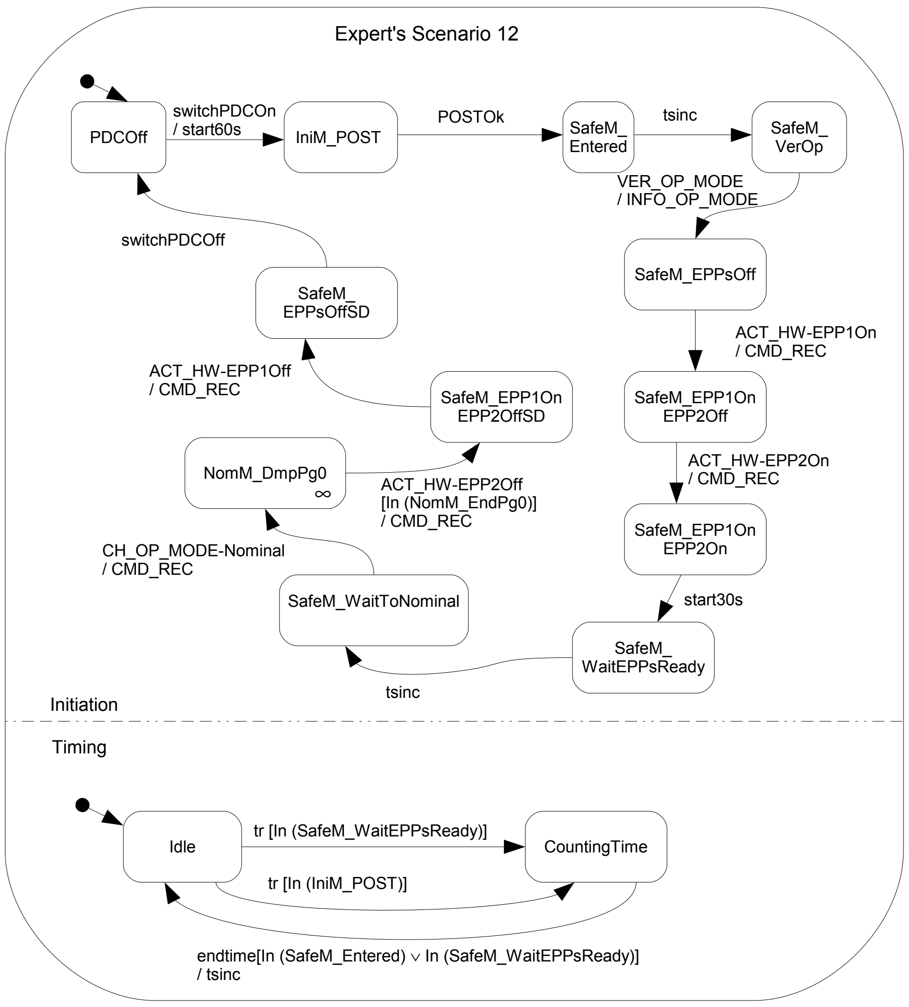


Figure 4.16 - Expert's scenario 12: main Statechart model
SOURCE: Adapted from SANTIAGO JÚNIOR et al. (2010)

1, TX_DATA-Dmp_0, TX_DATA-Sci} relates to the test objective "Dump Data of Data Memory (Page 0) in the Nominal Operation Mode". Now, the initial address has an acceptable value (8000h is equal to the minimum memory address allowed) and data can be dumped normally from page 0. Then, SWPDC responds with Dump Data (*DMP_DATA-RSC14-1_1111* and *DMP_DATA-RSC0_830*).

The test objective "Robustness addressing problem of incomplete reception of commands (reception)" is modeled by the following sequence of transitions: {*P_DMP-DataP0-8000H-BFFFH, TX_DATA-Dmp_14-1, TX_DATA-Dmp_0, RET_ANSW, TX_DATA-Sci*}. Note the expected result *timeout* due to the test input data TRANSMIT LAST DUMP DATA (*TX_DATA-Dmp_0*). As in expert's

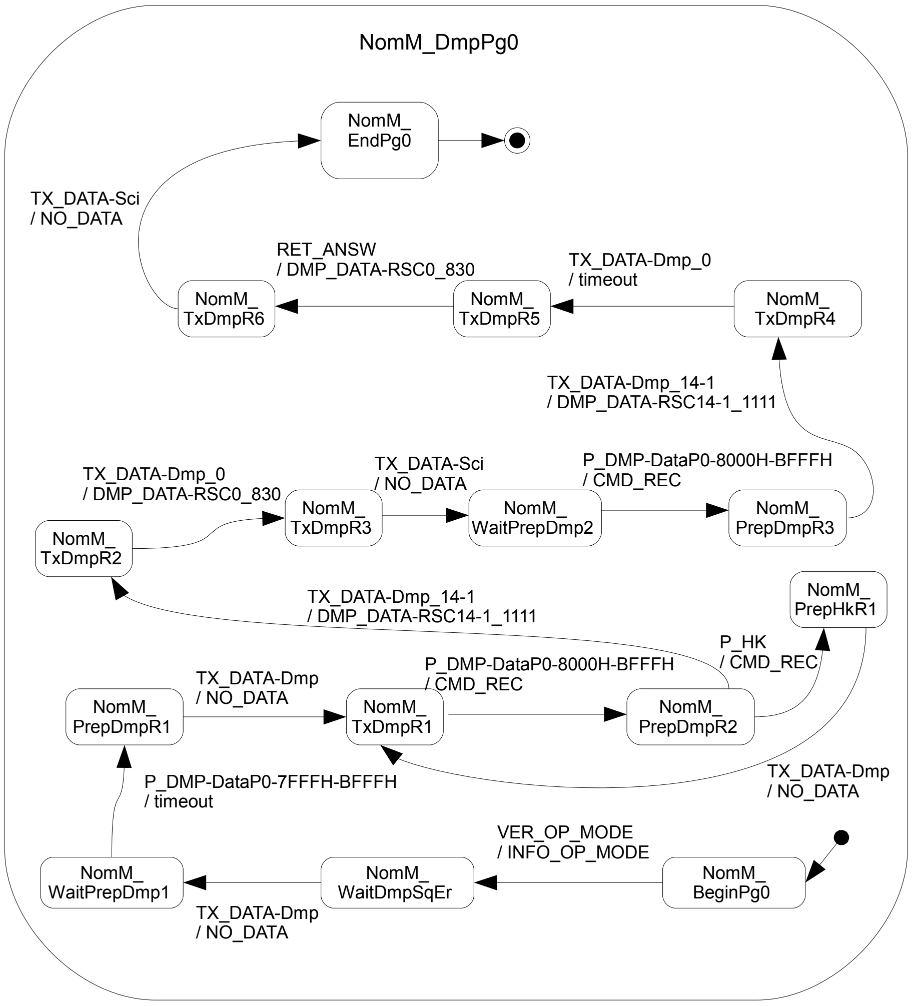


Figure 4.17 - Expert's scenario 12: *NomM_DmpPg0*
SOURCE: Adapted from SANTIAGO JÚNIOR et al. (2010)

scenario 8, the test designer assumed that a problem occurred and only part of the command was received by SWPDC. The OBDH can then send a command to RETRANSMIT THE LAST DATA RESPONSE (*RET_ANSW*). Also like the COMPOSITE state *NomM_HkData* in expert's scenario 8 (Figure 4.12), there is a minor defect in the model. The correct sequence of *test input data/expected result* to address this test objective should be: {*P_DMP-DataP0-8000H-BFFFH/CMD_REC, TX_DATA-Dmp_14-1/DMP_DATA-RSC14-1_1111, TX_DATA-Dmp_0/timeout, RET_ANSW/DMP_DATA-RSC1_1111, TX_DATA-Dmp_0/DMP_DATA-RSC0_830, TX_DATA-Sci/NO_DATA*}. The same three test objectives apply to all other types of PDC's memory, i.e. Program Memory (expert's scenario 10) and the remaining pages of the Data Memory (expert's scenarios 13, 14, 15, 17, 18, 19, 20). The models for these other expert's scenarios are quite similar to the ones for expert's scenario 12.

The SOLIMVA methodology covers expert's scenario 12 as follows. First, according to Table 4.5, unfolded scenario 73.2 (note DtP0 in every factor combination) is the one that relates to the Dump Data service from page 0 of the Data Memory. The set of factor combinations that is interpreted for generating unfolded scenario 73.2 is: {DtP0, InRng, Min}, {DtP0, Min, InRng}, {DtP0, Max, LessMin}, {DtP0, LessMin, Max}, {DtP0, GreatMax, InRng}, {DtP0, Min, GreatMax}. All factor combinations except the second ({DtP0, InRng, Min}, {DtP0, Max, LessMin}, {DtP0, LessMin, Max}, {DtP0, GreatMax, InRng}, {DtP0, Min, GreatMax}) drive the test designer to add requirements in which inconsistent values of initial and/or final memory addresses are set. In other words, {DtP0, InRng, Min} implies that the initial memory address is In Range (between the Minimum and Maximum allowed memory addresses), and the final memory address is the Minimum allowed memory address. Thus, the initial address is greater than the final address and this is an incorrect setting. The conclusion is that these five factor combinations relate to the test objective "Robustness taking into account problems (inconsistent values) within commands sent to PDC (command)".

Nevertheless, the second factor combination ({DtP0, Min, InRng}) is consistent and it addresses the test objective "Dump Data of Data Memory (Page 0) in the Nominal Operation Mode". When translating the Abstract Test Suite into the Executable Test Suite, the test designer can substitute Min for 8000h (which is the only possible value for Min) and InRng for BFFFh, which is one of many possible values for InRng. Then, the same initial and final memory addresses proposed in the model for expert's scenario 12 can be selected.

The set of NL requirements that characterize unfolded scenario 73.2 is shown in Table 4.16. As previously stated, the combinatorial designs direct the test designer to add requirements in accordance with the factor combinations. NL requirement SRS022 has the command PREPARE DUMP DATA FROM PAGE 0 OF DATA MEMORY, WITH INITIAL ADDRESS = INRNG AND FINAL ADDRESS = MIN (*PREP-DMP-DTP0-INRNG-MIN*). This requirement was added due to factor combination {DtP0, InRng, Min}. NL requirements SRS023, SRS024, SRS025, SRS026, and SRS027 were added due to factor combinations {DtP0, Min, InRng}, {DtP0, Max, LessMin}, {DtP0, LessMin, Max}, {DtP0, GreatMax, InRng}, {DtP0, Min, GreatMax}, respectively.

Table 4.16 - Set of NL requirements that characterize unfolded scenario 73.2. Caption: Req = Requirement; Id = Identification

Req Id	**Req Description**
SRS001	The PDC shall be powered on by the Power Conditioning Unit.
SRS002	The PDC shall be in the Initiation Operation Mode after being powered on. The 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.
SRS003	If PDC does not present any irrecoverable problem, after the initiation process, the PDC shall automatically enter into the Safety Operation Mode.
POCP001	The PDC can only respond to requests (commands) from OBDH after the PDC has been energized for at least 1 minute. If OBDH sends commands within less than 1 minute, the OBDH shall not receive any response from PDC.
RB001	The OBDH shall send VER-OP-MODE to PDC.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH sends distinct commands to perform such actions.
PECP001	Each EPP Hx can only respond to requests (commands) from PDC after each EPP Hx has been energized for at least 30 seconds. If PDC sends commands within less than 30 seconds to a certain EPP Hx, the PDC shall not receive any response from this EPP Hx.
RB003	The OBDH shall send CH-OP-MODE-NOMINAL to PDC.
RB001	The OBDH shall send VER-OP-MODE to PDC.
SRS022	Memory Dump data transmission shall start with PREP-DMP-DTP0-INRNG-MIN. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send One TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.
SRS023	Memory Dump data transmission shall start with PREP-DMP-DTP0-MIN-INRNG. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send Several TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.
SRS024	Memory Dump data transmission shall start with PREP-DMP-DTP0-MAX-LESSMIN. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send One TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.

(Continues)

Table 4.16 - Conclusion

Req Id	**Req Description**
SRS025	Memory Dump data transmission shall start with PREP-DMP-DTP0-LESSMIN-MAX. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send One TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.
SRS026	Memory Dump data transmission shall start with PREP-DMP-DTP0-GREATMAX-INRNG. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send One TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.
SRS027	Memory Dump data transmission shall start with PREP-DMP-DTP0-MIN-GREATMAX. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send One TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.
RB004	The OBDH shall send CH-OP-MODE-SAFETY to PDC. After that, the PDC shall be in the Safety Operation Mode.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH sends distinct commands to perform such actions.
RB005	After switching both EPPHxs off via PDC, the OBDH shall switch the PDC off via the Power Conditioning Unit.

Figures 4.18 and 4.19 show the COMPOSITE state *Nominal Operation Mode* for SOLIMVA's unfolded scenario 73.2. For this unfolded scenario, the main Statechart model is quite similar to the one generated for normal scenario 71 (Figure 4.2) as well as are similar the COMPOSITE states *Safety Operation Mode* (Figure 4.5) and *Safety Operation Mode_2* (Figure 4.4). The following sequences of transitions

- {*PREP-DMP-DTP0-INRNG-MIN, not stop_scientific datum acquisition, One TX-DATA-DMP, TX-DATA-SCI-End*},
- {*PREP-DMP-DTP0-MAX-LESSMIN, not stop_scientific datum acquisition, One TX-DATA-DMP, TX-DATA-SCI-End*},
- {*PREP-DMP-DTP0-LESSMIN-MAX, not stop_scientific datum acquisition, One TX-DATA-DMP, TX-DATA-SCI-End*},
- {*PREP-DMP-DTP0-GREATMAX-INRNG, not stop_scientific datum acquisition, One TX-DATA-DMP, TX-DATA-SCI-End*},

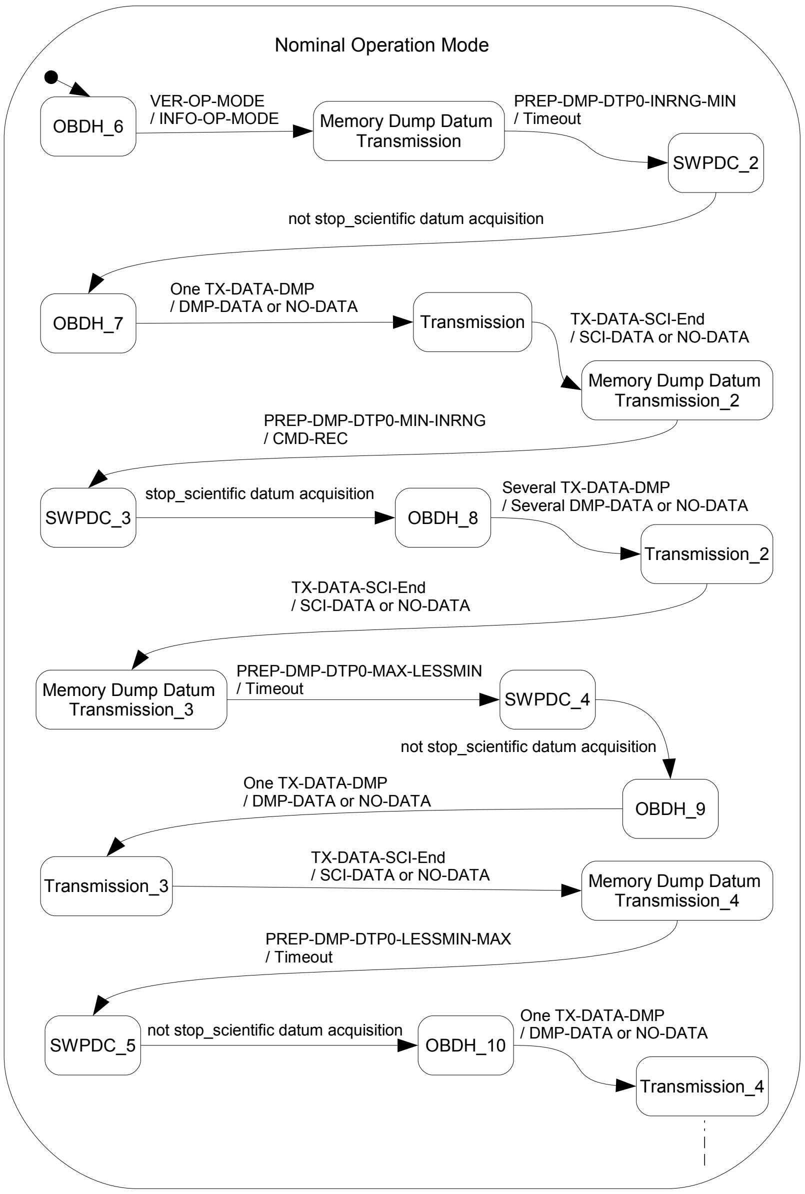
- {*PREP-DMP-DTP0-MIN-GREATMAX, not stop_scientific datum acquisition, One TX-DATA-DMP, TX-DATA-SCI-End*},

deal with situations where inconsistent values of initial and/or final memory addresses are accounted for. They were created as a result of the addition of the requirements due to all factor combinations except the second (Table 4.5). In these cases, all "expected results" related to commands PREPARE DUMP DATA FROM PAGE 0 OF DATA MEMORY (*PREP-DMP-DTP0*) are *Timeout* because of the wrong setting of addresses. These sequences of requirements address the test objective "Robustness taking into account problems (inconsistent values) within commands sent to PDC (command)".

On the other hand, the sequence of transitions {*PREP-DMP-DTP0-MIN-INRNG, stop_scientific datum acquisition, Several TX-DATA-DMP, TX-DATA-SCI-End*} is consistent and it addresses the test objective "Dump Data of Data Memory (Page 0) in the Nominal Operation Mode". The second factor combination was responsible for such sequence of transitions.

However, the test objective "Robustness addressing problem of incomplete reception of commands (reception)" is not covered not only by unfolded scenario 73.2 but also by any other unfolded scenario for other PDC's memories (73.1, 73.3, 73.4, etc.). In order to solve this problem, factor combination 143 ({Inv, Nom, Dmp, -}) can be used. Its interpretation defines normal scenario 143 and it deals with the incomplete reception of commands but considering all PDC's memories, i.e. the Program Memory and all pages of the Data Memory. This approach concentrates in a single scenario the aforementioned test objective instead of modeling in every scenario for each PDC's memory as the expert proposed. A sample of the set of NL requirements that characterize normal scenario 143 is shown in Table 4.17.

In Table 4.17, NL requirements SRS017, POCP021, SRS013 relate to the incomplete reception of commands when dumping data from the Program (PRG) Memory while SRS023, POCP021, SRS013 refer to page 0 of the Data Memory (DtP0). For the other pages of the Data Memory, it is enough to change the first of the three requirements and repeat the other two. Figure 4.20 shows a piece of the COMPOSITE state *Nominal Operation Mode* for normal scenario 143 in which it is possible to see the transitions that cover the test objective "Robustness addressing problem of incomplete reception of commands (reception)" when dealing with page 0 of the Data Memory (DtP0).



Nominal Operation Mode
OBDH_6
VER-OP-MODE / INFO-OP-MODE
Memory Dump Datum Transmission
PREP-DMP-DTP0-INRNG-MIN / Timeout
SWPDC_2
not stop_scientific datum acquisition
OBDH_7
One TX-DATA-DMP / DMP-DATA or NO-DATA
Transmission
TX-DATA-SCI-End / SCI-DATA or NO-DATA
Memory Dump Datum Transmission_2
PREP-DMP-DTP0-MIN-INRNG / CMD-REC
SWPDC_3
stop_scientific datum acquisition
OBDH_8
Several TX-DATA-DMP / Several DMP-DATA or NO-DATA
Transmission_2
TX-DATA-SCI-End / SCI-DATA or NO-DATA
Memory Dump Datum Transmission_3
PREP-DMP-DTP0-MAX-LESSMIN / Timeout
SWPDC_4
not stop_scientific datum acquisition
OBDH_9
One TX-DATA-DMP / DMP-DATA or NO-DATA
Transmission_3
TX-DATA-SCI-End / SCI-DATA or NO-DATA
Memory Dump Datum Transmission_4
PREP-DMP-DTP0-LESSMIN-MAX / Timeout
SWPDC_5
not stop_scientific datum acquisition
OBDH_10
One TX-DATA-DMP / DMP-DATA or NO-DATA
Transmission_4

Figure 4.18 - SOLIMVA's unfolded scenario 73.2: *Nominal Operation Mode* (Part 1)

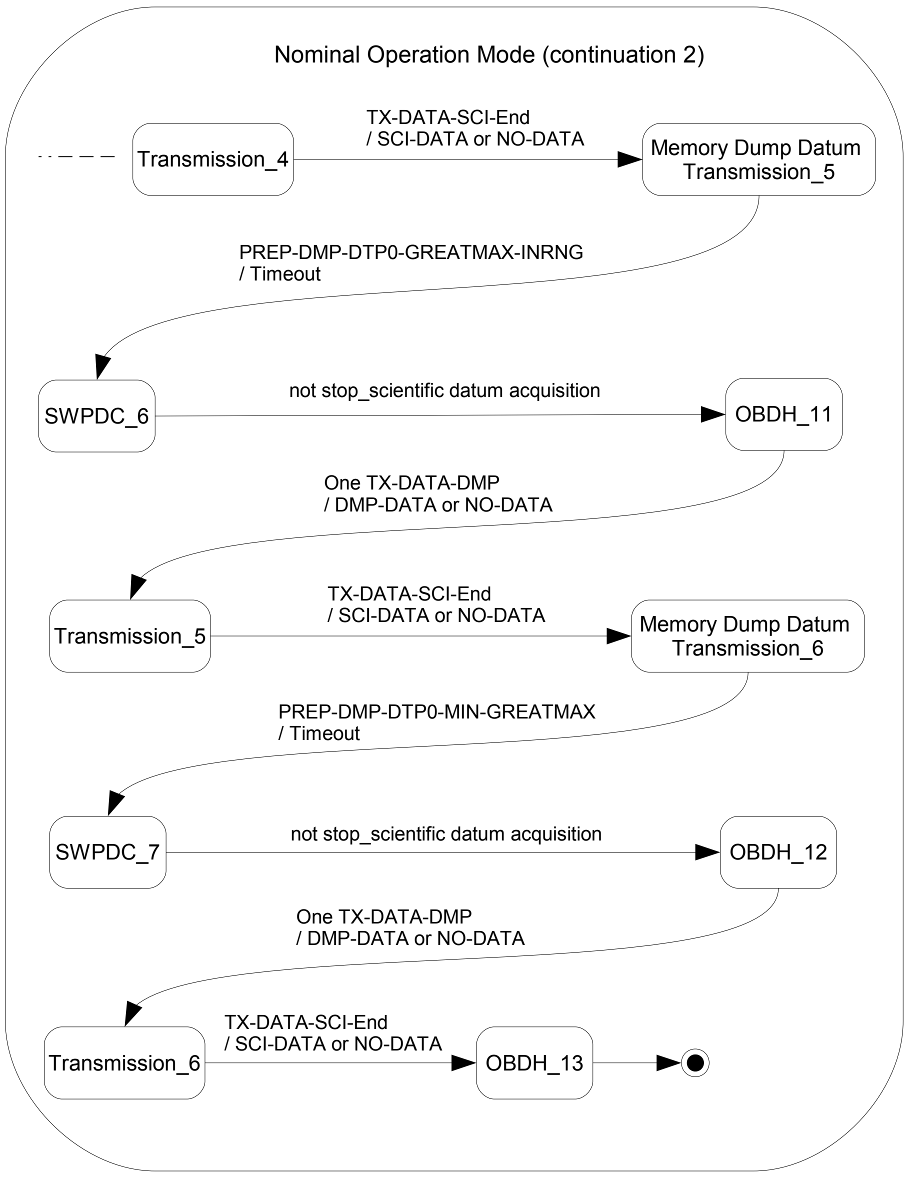

Nominal Operation Mode (continuation 2)
Transmission_4
TX-DATA-SCI-End / SCI-DATA or NO-DATA
Memory Dump Datum Transmission_5
PREP-DMP-DTP0-GREATMAX-INRNG / Timeout
SWPDC_6
not stop_scientific datum acquisition
OBDH_11
One TX-DATA-DMP / DMP-DATA or NO-DATA
Transmission_5
TX-DATA-SCI-End / SCI-DATA or NO-DATA
Memory Dump Datum Transmission_6
PREP-DMP-DTP0-MIN-GREATMAX / Timeout
SWPDC_7
not stop_scientific datum acquisition
OBDH_12
One TX-DATA-DMP / DMP-DATA or NO-DATA
Transmission_6
TX-DATA-SCI-End / SCI-DATA or NO-DATA
OBDH_13

Figure 4.19 - SOLIMVA's unfolded scenario 73.2: *Nominal Operation Mode* (Part 2)

Table 4.17 - Sample of the set of NL requirements that characterize normal scenario 143. Caption: Req = Requirement; Id = Identification

Req Id	**Req Description**
...	...
SRS017	Memory Dump data transmission shall start with PREP-DMP-PRG-INRNG-INRNG. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send Several TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.
POCP021	The PDC may not receive a command sent in its entirety. After identifying the beginning of a command frame, the PDC shall wait two times MAX-TRANSM-DELAY for the rest of the command. If this stipulated time expires, a timeout shall occur, the PDC shall abort the communication, the command shall be discarded, an event report shall be generated, and the PDC shall wait for a new OBDH's command.
SRS013	The SWPDC shall always maintain temporarily stored the last data response sent to the OBDH because the OBDH can demand the retransmission of this last data response.
SRS023	Memory Dump data transmission shall start with PREP-DMP-DTP0-MIN-INRNG. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send Several TX-DATA-DMP to PDC. The transmission shall be ended with TX-DATA-SCI-End.
POCP021	The PDC may not receive a command sent in its entirety. After identifying the beginning of a command frame, the PDC shall wait two times MAX-TRANSM-DELAY for the rest of the command. If this stipulated time expires, a timeout shall occur, the PDC shall abort the communication, the command shall be discarded, an event report shall be generated, and the PDC shall wait for a new OBDH's command.
SRS013	The SWPDC shall always maintain temporarily stored the last data response sent to the OBDH because the OBDH can demand the retransmission of this last data response.
...	...

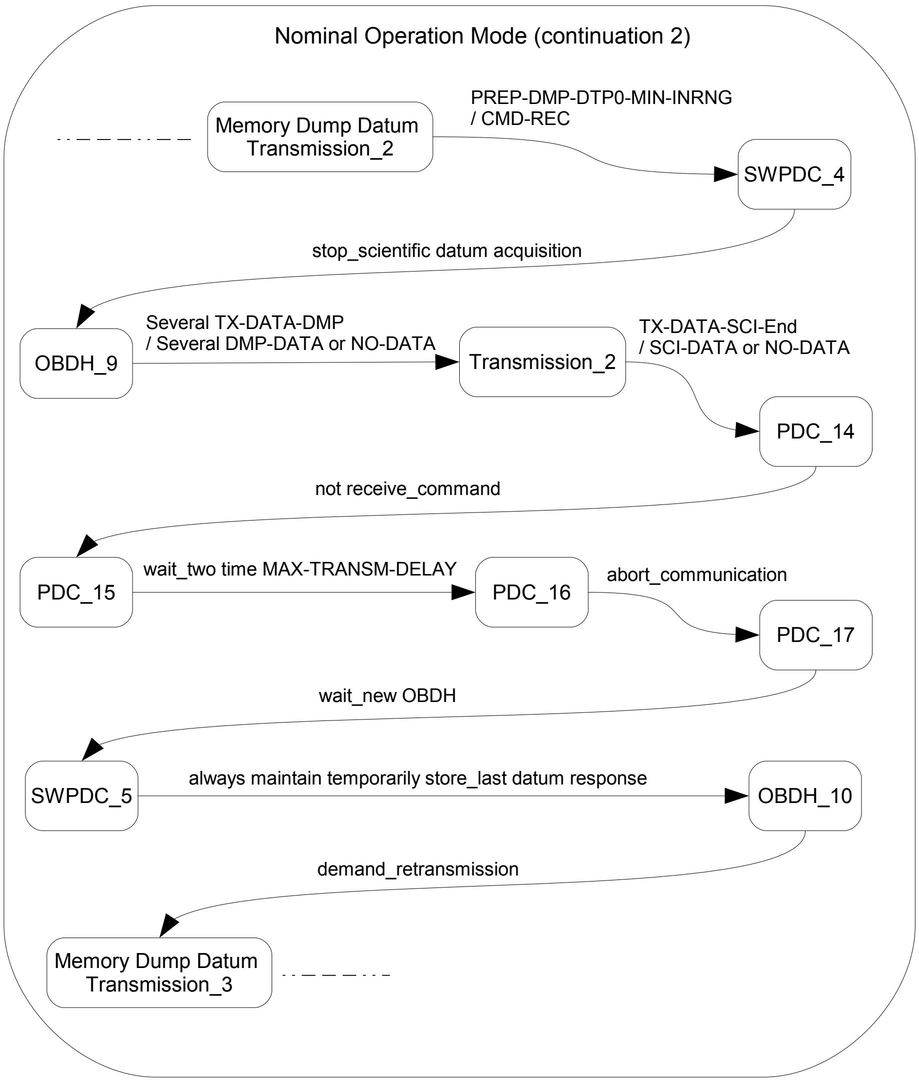

Nominal Operation Mode (continuation 2)
Memory Dump Datum Transmission_2
PREP-DMP-DTP0-MIN-INRNG / CMD-REC
SWPDC_4
stop_scientific datum acquisition
OBDH_9
Several TX-DATA-DMP / Several DMP-DATA or NO-DATA
Transmission_2
TX-DATA-SCI-End / SCI-DATA or NO-DATA
PDC_14
not receive_command
PDC_15
wait_two time MAX-TRANSM-DELAY
PDC_16
abort_communication
PDC_17
wait_new OBDH
SWPDC_5
always maintain temporarily store_last datum response
OBDH_10
demand_retransmission
Memory Dump Datum Transmission_3

Figure 4.20 - SOLIMVA's normal scenario 143: a piece of the COMPOSITE state *Nominal Operation Mode*

Comparisons with other expert's scenarios are similar to those presented in this section. Therefore, it was demonstrated how the SOLIMVA methodology adequately covered the test objectives associated with the expert's scenarios. As previously mentioned, the SOLIMVA methodology proposed a better strategy with test objectives clearly separated according to the directives of the combinatorial designs.

4.1.2 Characteristics of Executable Test Cases

The second goal of the comparison was also achieved. For the sake of demonstration, comparative analysis will be made involving two expert's scenarios. Consider expert's scenario 5 whose main Statechart model is shown in Figure 4.21. As already discussed, this scenario is covered by SOLIMVA's normal scenario 71 whose models were previously shown (Figures 4.2, 4.3, 4.5, 4.4). Choosing the all-transitions test criterion, there is only one Executable Test Case that composes the Executable Test Suite based on the expert's model:

> {*switchPDCOn/start60s, POSTOk/null, endtime/null, VER_OP_-MODE/INFO_OP_MODE, ACT_HW-EPP1On/CMD_REC, ACT_HW-EPP2On/CMD_REC, start30s/null, endtime/null, CH_OP_MODE-Nominal/CMD_REC, VER_OP_MODE/INFO_OP_MODE, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, start10s/null, endtime/null, TX_DATA-Sci/SCI_DATA, ACT_HW-EPP2Off/CMD_REC, ACT_HW-EPP1Off/CMD_REC, switchPDCOff/null*}.

In order to compare the Executable Test Cases of both approaches, consider Table 4.18. Columns *Expert* and *SOLIMVA* show the test steps of the Executable Test Cases based on the expert's and SOLIMVA's strategies, respectively. Notice that all test steps of the single Executable Test Case of the expert's approach were satisfactorily covered by the three Executable Test Cases by employing the SOLIMVA methodology. Moreover, SOLIMVA provides several benefits over the expert's approach. First, the test steps {*start30s/null, endtime/null*} of the expert's approach were covered by a more detailed set of test steps. These test steps were based on NL requirement PECP001 (Table 4.7):

> PECP001 - Each EPP Hx can only respond to requests (commands) from PDC after each EPP Hx has been energized for at least 30 seconds. If PDC sends commands within less than 30 seconds to a certain EPP Hx, the PDC shall not receive any response from this EPP Hx.

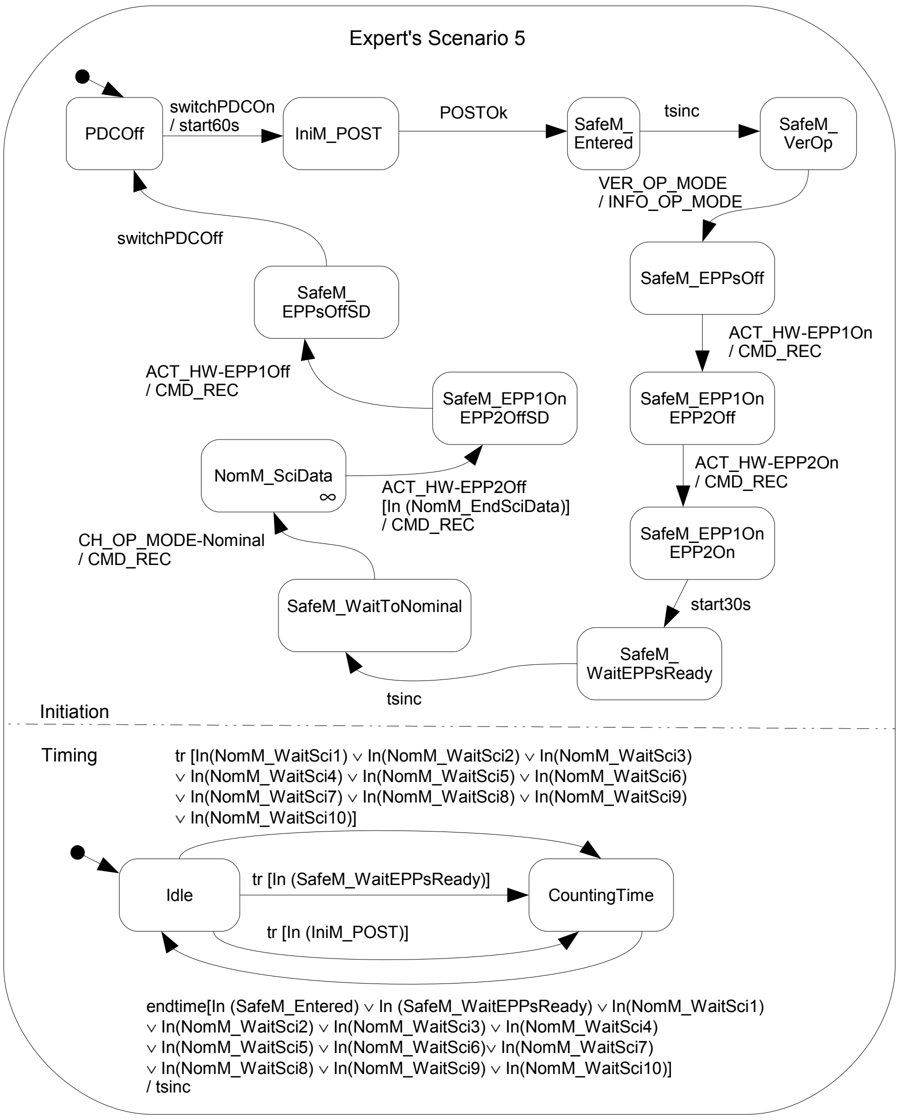

Expert's Scenario 5
PDCOff
switchPDCOn / start60s
IniM_POST
POSTOk
SafeM_ Entered
tsinc
SafeM_ VerOp
VER_OP_MODE / INFO_OP_MODE
SafeM_EPPsOff
ACT_HW-EPP1On / CMD_REC
SafeM_EPP1On EPP2Off
ACT_HW-EPP2On / CMD_REC
SafeM_EPP1On EPP2On
start30s
SafeM_ WaitEPPsReady
tsinc
SafeM_WaitToNominal
CH_OP_MODE-Nominal / CMD_REC
NomM_SciData ∞
ACT_HW-EPP2Off [In (NomM_EndSciData)] / CMD_REC
SafeM_EPP1On EPP2OffSD
ACT_HW-EPP1Off / CMD_REC
SafeM_ EPPsOffSD
switchPDCOff
Initiation
Timing
tr [In(NomM_WaitSci1) ∨ In(NomM_WaitSci2) ∨ In(NomM_WaitSci3) ∨ In(NomM_WaitSci4) ∨ In(NomM_WaitSci5) ∨ In(NomM_WaitSci6) ∨ In(NomM_WaitSci7) ∨ In(NomM_WaitSci8) ∨ In(NomM_WaitSci9) ∨ In(NomM_WaitSci10)]
Idle
tr [In (SafeM_WaitEPPsReady)]
CountingTime
tr [In (IniM_POST)]
endtime[In (SafeM_Entered) ∨ In (SafeM_WaitEPPsReady) ∨ In(NomM_WaitSci1) ∨ In(NomM_WaitSci2) ∨ In(NomM_WaitSci3) ∨ In(NomM_WaitSci4) ∨ In(NomM_WaitSci5) ∨ In(NomM_WaitSci6)∨ In(NomM_WaitSci7) ∨ In(NomM_WaitSci8) ∨ In(NomM_WaitSci9) ∨ In(NomM_WaitSci10)] / tsinc

Figure 4.21 - Expert's scenario 5: main Statechart model
SOURCE: Adapted from SANTIAGO JÚNIOR et al. (2010)

Table 4.18 - A comparison between expert's and SOLIMVA's Executable Test Suites: expert's scenario 5 and SOLIMVA's normal scenario 71. Caption: #ETC-E/-S = number of Executable Test Case within the Executable Test Suite obtained from expert's (E) and SOLIMVA's (S) approaches

#ETC-E	Expert	#ETC-S	SOLIMVA
1	switchPDCOn/start60s	1, 2	Action1, Action3
1	POSTOk/null	2	Action2
1	endtime/null	2	Action3
1	VER_OP_MODE/INFO_OP_-MODE	2	VER-OP-MODE/INFO-OP-MODE
1	ACT_HW-EPP1On/CMD_REC	2	ACT-HW-EPPH1ON/CMD-REC
1	ACT_HW-EPP2On/CMD_REC	2	ACT-HW-EPPH2ON/CMD-REC
1	start30s/null, endtime/null	2	PREP-TST/CMD-REC, Action4, TX-DATA-TST/NO-DATA, TX-DATA-TST/NO-DATA, TX-DATA-SCI/NO-DATA, Action5
1	CH_OP_MODE-Nominal/CMD_-REC	2	CH-OP-MODE-NOMINAL/CMD-REC
1	start10s/null, endtime/null, TX_-DATA-Sci/SCI_DATA	2	Action4, TX-DATA-SCI/SCI-DATA
1	ACT_HW-EPP2Off/CMD_REC	2	ACT-HW-EPPH2OFF/CMD-REC
1	ACT_HW-EPP1Off/CMD_REC	2	ACT-HW-EPPH1OFF/CMD-REC
1	switchPDCOff/null	3	Action7
-	-	1	Action6
-	-	2	VER-OP-MODE/Timeout
-	-	2	Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA
-	-	2	CH-OP-MODE-SAFETY/CMD-REC, VER-OP-MODE/INFO-OP-MODE

In case of the expert's approach, all that should be done is to start a timer and wait 30 seconds. After this time interval, the PDC can begin to request data (Scientific, Test, Diagnosis) from EPP Hxs under OBDH request. However, the Abstract Test Cases derived by applying the SOLIMVA methodology contain the following abstract *input/output* pairs from PECP001: {*only respond_request/null + have be* *energize_least 30 seconds/null + send_command/null + not receive_response/null*} (Table 4.9). When looking at these *input/output* pairs, the test designer realized that there are more relevant behaviors to be considered in the Executable Test Case, say the fact that if an EPP Hx receives commands within less than 30 seconds, the PDC shall not receive any response from this EPP Hx. Although this requirement is more related to EPP Hxs, it is important to verify whether PDC (SWPDC) acts adequately in such a situation. Then, instead of only waiting 30 seconds to start interacting with EPP Hxs, the test designer translated the abstract *input/output* pairs into a set of executable *test input data/expected result* aiming at the transmission of Test (TST) Data (one of the three types of data generated by EPP Hxs) within less than 30 seconds after each EPP Hx has been powered on (*ACT-HW-EPPH1ON*, *ACT-HW-EPPH2ON*). As shown in Table 4.18, when the OBDH asks PDC to transmit Test Data (*TX-DATA-TST*) the expected result is NO DATA (*NO-DATA*) because EPP Hxs are not yet available for communication. This improvement in the Executable Test Case due to the SOLIMVA methodology shows that looking at the Abstract Test Cases is more interesting rather than looking at a set of NL requirements in deliverables because the Abstract Test Cases provide a concise notation and emphasize the most relevant NL sentences, and some command/responses of Communication Protocol Specifications (in this particular case study) so that the test designer can derive more suitable Executable Test Cases.

Although all test steps of the expert's Executable Test Case were addressed by SOLIMVA's Executable Test Suite, the opposite is not true. One very important test step missed in the expert's Executable Test Case is *Action6* which deals with severe problems during the PDC initiation (initialization) process. This test step relates to the translation of the self transition *be#then accomplish_post#present_irrecoverable problem#/null* in the *Initiation Operation Mode* state (Figures 4.2, 4.7, 4.9). As presented in Figure 4.21, no self transition exists in the state that represents the Initiation Operation Mode (*IniM_POST*) in the expert's model. Then, one must not expect that any expert's Executable Test Case covers this behavior. The automated reading of NL requirements and the translation from the Abstract Test Suite into the Executable Test Suite make the SOLIMVA methodology an interesting solution to minimize problems related to the incomplete creation of models for Model-Based Testing.

Another situation that is not present in the expert's Executable Test Suite is verifying the behavior of PDC if a command is sent by the OBDH within less than 1 minute since PDC has been energized (*VER-OP-MODE/Timeout*). This behavior relates to

NL requirement POCP001 (Table 4.7). Furthermore, it is interesting to check some Housekeeping Data ({*Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA*}) before entering into the Nominal Operation Mode in order to request Scientific Data. Finally, the expert's model did not consider the fact that in order to switch PDC off, it is recommended (there are requirements related to this recommendation) to put PDC in the Safety Operation Mode ({*CH-OP-MODE-SAFETY/CMD-REC, VER-OP-MODE/INFO-OP-MODE*}). Hence, states *SafeM_EPP1OnEPP2OffSD* and *SafeM_EPPsOffSD* in Figure 4.21 are incorrectly named as being in the Safety Operation Mode (*SafeM*). Although this last issue might not be a huge problem for Statechart model-based test case generation (more explanation about this in Section 4.3), again this shows that the SOLIMVA methodology presents advantages over the expert's (manual) approach.

The comparative analysis is similar if the Executable Test Suites derived from expert's scenario 5 and SOLIMVA's normal scenario 71 and generated according to the all-simple-paths test criterion are considered. Likewise the all-transitions test criterion in the expert's approach, the Executable Test Suite due to the all-simple-paths test criterion is composed of only one Executable Test Case and this test case has exactly the same test steps shown earlier in this section. As shown in Table 4.11, the all-simple-paths Executable Test Suite generated according to the models for SOLIMVA's normal scenario 71 is composed of two Executable Test Cases. Executable Test Cases 1 and 2 of the all-simple-paths suite have the same test steps of Executable Test Cases 2 and 3 of the all-transitions Executable Test Suite, respectively. But, the first Executable Test Case ({*Action1, Action6*}) of the all-transitions suite does not belong to the all-simple-paths suite. This happens due to the models generated by the SOLIMVA tool and the all-simple-paths test criterion. This implies that the all-simple-paths Executable Test Suite does not predict that the IUT can be stimulated in situations where severe problems occur during the PDC initiation process since *Action6* is not present. In this case, the Executable Test Suite due to the SOLIMVA methodology is not better than the one from the expert's strategy. However, all the other advantages of SOLIMVA over the expert's approach previously presented for the all-transitions criterion are also advantages for the all-simple-paths test criterion.

Consider expert's scenario 4 whose main Statechart model was presented in Figure 4.6. This scenario is addressed by SOLIMVA's normal scenario 50 (Figures 4.7, 4.8) and 17 (Figures 4.9, 4.10, 4.11). Table 4.19 shows the Executable Test Suites due to both approaches and considering the all-transitions test criterion.

Table 4.19 - Executable Test Suites due to expert's scenario 4, SOLIMVA's normal scenarios 50 and 17, and the all-transitions test criterion

Approach	**Executable Test Cases**
Expert - 4	{switchPDCOn/start60s, POSTOk/null, endtime/null, VER_OP_MODE/INFO_OP_MODE, CH_SW_PAR-Hk_60s/CMD_REC, start600s/null, endtime/null, PREP_HK/CMD_REC, TX_DATA-Hk/HK_DATA-RSC0, va_HK_DATA/er_POST-PowerOn, ACT_HW-Reset/CMD_REC, start60s/null, POSTOk/null, endtime/null, VER_OP_MODE/INFO_OP_MODE, start60s/null, endtime/null, PREP_HK/CMD_REC, TX_DATA-Hk/HK_DATA-RSC0, va_HK_DATA/er_POST-Reset, switchPDCOff/null}
SOLIMVA - 50	{Action1, Action6}, {Action1, Action2, VER-OP-MODE/Timeout, Action3, VER-OP-MODE/INFO-OP-MODE, Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA, Action10}, {Action7}
SOLIMVA - 17	{Action1, Action6}, {Action1, Action2, VER-OP-MODE/Timeout, Action3, VER-OP-MODE/INFO-OP-MODE, ACT-HW-RESET/CMD-REC}, {Action1, Action2, VER-OP-MODE/Timeout, Action3, VER-OP-MODE/INFO-OP-MODE, Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA, Action9}, {Action7}

The amount of Executable Test Cases that compose the Executable Test Suites are 1, 3, 4 for expert's scenario 4, SOLIMVA's normal scenarios 50 and 17, respectively. Table 4.20 shows a comparison among the Executable Test Suites shown in Table 4.19. Observe that column *#ETC-S* shows the number of Executable Test Case within the Executable Test Suite derived by applying the SOLIMVA methodology but also, in parentheses, the normal scenario to which the Executable Test Case is related to. For instance, the test step *switchPDCOn/start60s* of the first (unique) test case due to the expert's approach is covered by *Action1, Action3* which are test steps of Executable Test Cases 1 and 2 of both normal scenarios (50 and 17). Moreover, checking event reports within Housekeeping Data in order to realize whether SWPDC correctly updated information related to the reset (*Action9*) and power on (*Action10*) processes are important actions to be accomplished during execution of the test cases.

One test step of the expert's Executable Test Case was not addressed by SOLIMVA's Executable Test Cases: the test step whose *test input data* is CHANGE SOFTWARE PARAMETERS - SET THE TIME TO GENERATE HOUSEKEEPING DATA TO THE MINIMUM VALUE (*CH_SW_PAR-Hk_60s*). The minimum value of this parameter is 60 seconds while the default value is 600 seconds. The expert changed this parameter to the minimum value because he/she needs to verify, in this single scenario, the contents of Housekeeping Data twice: first time to see if SWPDC properly carried out activities related to the power on (*va_HK_DATA/er_POST-PowerOn*) process and a second time to see the behavior of SWPDC related to the reset (*va_HK_DATA/er_POST-Reset*) process. This change aims to decrease the waiting time for a Housekeeping Data frame is fully ready to be transmitted to the OBDH in the second time, i.e. rather than waiting 600 seconds the OBDH needs to wait only 60 seconds. Housekeeping Data is then supposed to be transmitted twice in this scenario (*TX_DATA-Hk/HK_DATA-RSC0* in Table 4.19).

However, the SOLIMVA methodology suggests dividing expert's scenario 4 into two other scenarios, normal scenarios 50 and 17. Due to this separation, in each normal scenario Housekeeping Data is supposed to be transmitted to the OBDH only once (*TX-DATA-HK/HK-DATA* in Table 4.19). Then, the test designer intentionally did not bother to change this parameter to the minimum value (60 seconds) due to the fact that only one transmission of Housekeeping Data was predicted per scenario. However, if he/she wanted to have the same effect as expert's scenario 4, it was necessary just to have added the following NL requirement before resetting PDC in SOLIMVA's normal scenario 17:

POCP003 - The OBDH shall send CH-SW-PHK-MIN to PDC.

Table 4.20 - A comparison between expert's and SOLIMVA's Executable Test Suites: expert's scenario 4 and SOLIMVA's normal scenarios 50 and 17. Caption: #ETC-E/-S = number of Executable Test Case within the Executable Test Suite obtained from expert's (E) and SOLIMVA's (S) approaches

#ETC-E	Expert	#ETC-S	SOLIMVA
1	switchPDCOn/start60s	1 (50,17), 2 (50,17)	Action1, Action3
1	POSTOk/null	2 (50,17)	Action2
1	endtime/null	2 (50,17)	Action3
1	VER_OP_MODE/INFO_OP_-MODE	2 (50,17)	VER-OP-MODE/INFO-OP-MODE
1	CH_SW_PAR-Hk_60s/CMD_REC	-	-

(Continues)

Table 4.20 - Conclusion

#ETC-E	Expert	#ETC-S	SOLIMVA
1	start600s/null, endtime/null, PREP_HK/CMD_REC, TX_DATA-Hk/HK_DATA-RSC0	2 (50)	Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA
1	va_HK_DATA/er_POST-PowerOn	2 (50)	Action10
1	ACT_HW-Reset/CMD_REC	2 (17)	ACT-HW-RESET/CMD-REC
1	start60s/null	3 (17)	Action3
1	POSTOk/null	3 (17)	Action2
1	endtime/null	3 (17)	Action3
1	VER_OP_MODE/INFO_OP_-MODE	3 (17)	VER-OP-MODE/INFO-OP-MODE
1	start60s/null, endtime/null, PREP_HK/CMD_REC, TX_DATA-Hk/HK_DATA-RSC0	3 (17)	Action5, PREP-HK/CMD-REC, TX-DATA-HK/HK-DATA, TX-DATA-SCI/NO-DATA
1	va_HK_DATA/er_POST-Reset	3 (17)	Action9
1	switchPDCOff/null	3 (50), 4 (17)	Action7
-	-	1 (50,17)	Action6
-	-	2 (50,17), 3 (17)	VER-OP-MODE/Timeout

All remaining test steps of the expert's Executable Test Case were addressed by SOLIMVA's Executable Test Suites. As in the previous analysis, some important test steps presented in SOLIMVA's Executable Test Cases are not in the expert's Executable Test Case: *Action6* which deals with severe problems during the PDC initiation (initialization) process; *VER-OP-MODE/Timeout* to verify the behavior of PDC if a command is sent by the OBDH within less than 1 minute since PDC has been energized.

For the all-simple-paths test criterion, the comparative analysis is like the previous case involving expert's scenario 5 and SOLIMVA's normal scenario 71. In the expert's strategy, the Executable Test Suite due to the all-simple-paths test criterion is composed of only one Executable Test Case and this test case has exactly the same test steps of the test case due to the all-transitions test criterion. The all-simple-paths Executable Test Suite generated according to the models for SOLIMVA's normal scenario 50 is composed of two Executable Test Cases: Executable Test Cases 1 and 2 of the all-simple-paths suite have the same test steps of Executable Test Cases 2 and 3 of the all-transitions Executable Test Suite, respectively. For SOLIMVA's normal scenario 17, the all-simple-paths suite consists of two Executable Test Cases too, 1 and 2, which are identical to Executable Test Cases 3 and 4 of the all-transitions suite, respectively. Again, the first Executable Test Case ({*Action1, Action6*}) of the all-transitions suite is not in the all-simple-paths Executable Test Suite and therefore the Executable Test Suite due to the SOLIMVA methodology is not better than the expert's approach, with regard to investigating the behavior of SWPDC upon the occurrence of severe problems during the initialization of PDC. The other advantage of SOLIMVA over the expert's approach is also valid for the all-simple-paths test criterion.

The conclusion is that the Executable Test Cases derived in accordance with the SOLIMVA methodology not only possessed similar characteristics with the expert's Executable Test Cases but also predicted behaviors that did not exist in the expert's strategy. Then, the models automatically created by the SOLIMVA methodology and by the SOLIMVA tool are suitable for generating test cases. The suitability of the models are relevant because the test criteria used for test case generation based on FSMs and/or Statecharts are basically state-transition traversal rules. If the model is poorly developed, the Executable Test Cases might dictate a sequence of stimuli that are incoherent and in many situations impossible to be executed taking into account the real system.

4.2 A second case study: Satellite Control System

Threats to external validity (BASILI et al., 1996) exist with respect to version 1.0 of the SOLIMVA methodology. Threats to external validity imply limitations to generalizing the results. The methodology has been applied to only one case study (SWPDC) which is not enough and by a single professional. However, recall that the characteristics of the SWPDC software product are representative of an important class of complex software products in space application domain, more precisely in the Space Segment (ECSS, 2008). In addition, for a different class of software products in the space domain, it is feasible to apply the SOLIMVA methodology and the amount of rework to do that tends to be lower than in other application domains due to general similarities of the software products. For instance, the concept of *operation mode* is common to both the Space Segment and Ground

Segment systems/subsystems. As shown in the SWPDC case study, PDC's operation modes were elements of the Name (N) set, they were also in the ordered pairs of the Hierarchy ($S_{TM}.Y$) function, and they were selected to be levels for the combinatorial designs algorithm.

This section shows guidelines to apply the SOLIMVA methodology to a second case study of the space domain related to the Ground Segment. The *Satellite Control System* (SATCS) is intended to be a product line to provide a common Ground Segment software infrastructure for controlling satellites. This infrastructure is composed of several software elements such as components, class libraries, frameworks, and databases which will be customized to comply with the operation requirements for each space mission of INPE. Furthermore, the design of SATCS proposes a logical architecture template to be used as a standard in the design of its subsystems (CARDOSO et al., 2008). For activities related to the control of a particular satellite, a customized version of the SATCS software product will be installed in the *Centro de Controle de Satélites* (CCS - Satellite Control Center) of INPE and, if necessary, it will also be installed in ground stations as a spare unit. Figure 4.22 shows the functional architecture of SATCS.

Note that in the CCS Node, SATCS can be run on multiple computers connected via a Local Area Network. Moreover, the number of computers where the SATCS application will be run is not fixed depending on the satellite operational phase. One of the constituents of the logical architecture of the SATCS software product is the Commanding (CMD) framework (subsystem). CMD implements functions for preparing, validating, sending, verifying, and logging of command messages. These functions apply to any entity that can be commanded such as a satellite (telecommand) or a ground station (remote command) for which the command messages can be defined.

The transmission of commands can be performed in three operation modes: Manual, Automatic, and Supervised. In the Manual Operation Mode, the transmission of commands is entirely controlled by the satellite's operator. In the Automatic Operation Mode, all transmissions of commands are executed automatically based on the Flight Operation Plan (FOP). The Supervised Operation Mode also accomplishes the transmission of commands based on the FOP, as with the Automatic Operation Mode. The difference is that the satellite's operator must confirm each operation before it can be executed. If an error occurs in Supervised or Automatic Operation Modes, CMD should go back to the Manual Operation Mode and the satellite's

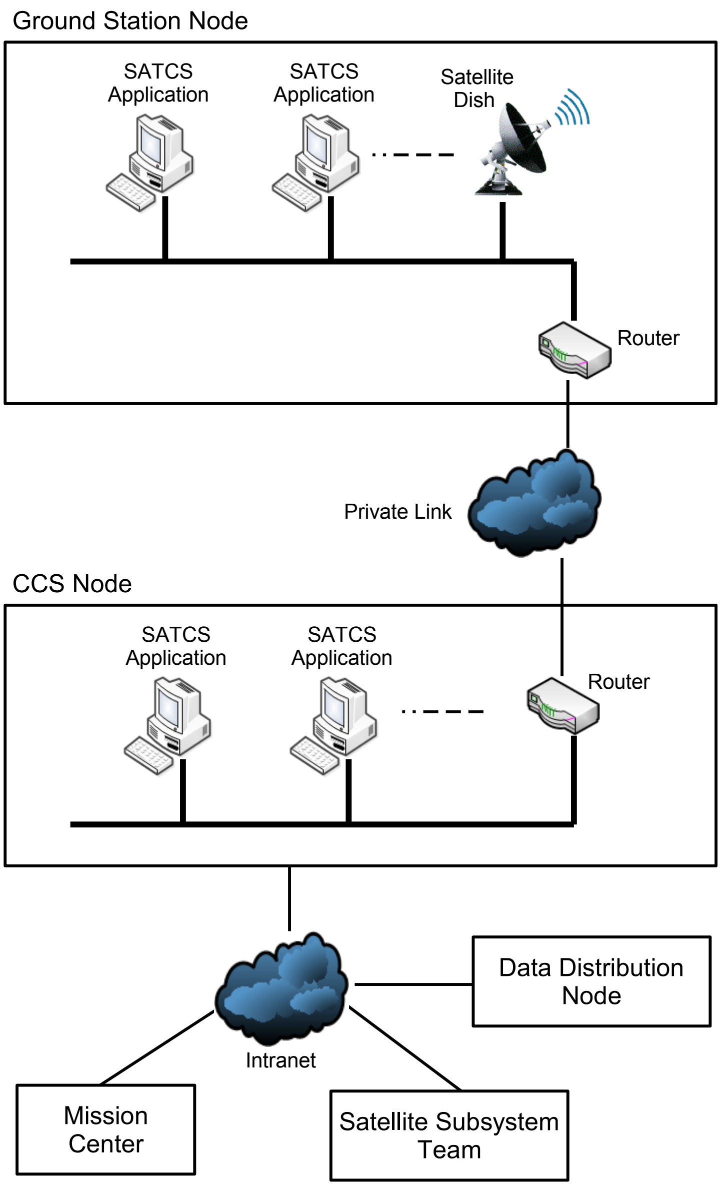


Figure 4.22 - Functional architecture of SATCS
SOURCE: Adapted from INPE.DSS (2010)

operator must decide whether or not to return to the Supervised or Automatic Operation Modes. Changes in operation modes can be: from Manual to Automatic and vice versa, and from Manual to Supervised and vice versa. But it is not possible to change from the Automatic Operation Mode to the Supervised Operation Mode and the opposite, from Supervised to Automatic.

Suppose that a test designer wants to apply SOLIMVA considering as a system the CMD subsystem. Hence, the Name (N) set can be composed of: $N =$ {CMD, Manual Operation Mode, Automatic Operation Mode, Supervised Operation Mode, ...}. One possible selection of factors and levels for the CMD/SATCS case study is shown in Table 4.21. The meanings of factors and levels are as follows:

a) OpMode factor: This factors represents the CMD's operation modes: Manual (Man), Automatic (Auto), and Supervised (Super);

b) ConfOpt factor: Regardless of the operation mode, CMD has two configurations (Conf): Satellite (Sat) and Test (Test). In the Satellite configuration, commands are really transmitted to the satellite while in the Test configuration commands are not sent to the satellite, i.e. they are only sent to the ground station. The purpose of this configuration is to support communication tests with the ground station. In addition, there are also configuration options (Opt): no checking associated with a command is performed, i.e. Normal situation (Norm); a command must be verified before its transmission (PreVer). If the real-time values of the related telemetries are not the expected ones, the command is not transmitted; verification whether a command was correctly transmitted (Tx) by means of a telemetry value received during real-time operation; and verification whether an immediate command was executed (Ex) by means of values of the related telemetries received during real-time operation. Hence, ConfOpt is a factor that combines configuration with configuration option. Thus, the SatNorm level means that the command is indeed transmitted to the satellite and no checking is performed. Similar interpretations apply to the other levels;

c) Editing factor: This factor is related to several tasks that can be performed with the queue of commands: insert a single command (InsCmd), insert a group of commands (InsGrp), insert a sequence of commands (InsSeq), insert commands from the FOP (InsFOP), update the date/time to send a command (UpTCmd), update the date/time to send a set of commands by giving a uniform increment in their times (UpTInc), and delete commands (DelCmd).

In Table 4.21, note the *Inv* level to address Robustness testing. When running the combinatorial designs algorithm with *strength* = 2, 48 factor combinations were produced and thus 48 normal scenarios were derived. Table 4.22 shows 10 of these

Table 4.21 - Factors and levels for the CMD/SATCS case study

Factors	**Levels**							
OpMode	Man	Auto	Super	Inv				
ConfOpt	SatNorm	SatPreVer	SatTx	SatEx	TestNorm	Inv		
Editing	InsCmd	InsGrp	InsSeq	InsFOP	UpTCmd	UpTInc	DelCmd	Inv

48 SOLIMVA's normal scenarios. In order to generate the Executable Test Cases, the activities of the SOLIMVA methodology as discussed in the previous chapter must be followed. That is, generating test cases for this second case study by using the SOLIMVA methodology would simply be a repetitive process.

4.3 Final remarks about this chapter

This chapter presented the application of version 1.0 of the SOLIMVA methodology and supporting tools (SOLIMVA tool and GTSC environment) to the SWPDC (SANTIAGO et al., 2007) case study, a software product related to the Space Segment (ECSS, 2008). The SOLIMVA methodology was compared with a previous manual approach developed by an expert in the SWPDC software product. The SOLIMVA methodology demonstrated several advantages over the approach of the specialist. Also, guidelines have been shown to apply SOLIMVA to a second case study related to the Ground Segment (CARDOSO et al., 2008). The application of SOLIMVA therefore to this second case study is also feasible.

This section presents additional and important remarks about version 1.0 of the SOLIMVA methodology and the SOLIMVA tool. Each activity of the SOLIMVA methodology requires some sort of manual intervention but most of these activities have also a degree of automation. However, as mentioned in Chapter 1, Software Testing automation is in fact a semi-automated process which still relies heavily on manual tasks performed by professionals. With respect to the activities of version 1.0 of the SOLIMVA methodology (Figure 3.3), the degree of manual intervention and automation is as follows:

a) *Define and Input Dictionary*, *Update Dictionary*. As previously mentioned, the test designer needs only to define and input, via Graphical User Interface of the SOLIMVA tool, the Name (N) set, the Reactiveness (R) function, and the Hierarchy ($S_{TM}.Y$) function. The other sets of the Dictionary (Control, Self Transition) and any other auxiliary sets, for instance the sets that help in the generation of the BSAO tuples (Section 3.3.1), do

Table 4.22 - CMD/SATCS case study: normal scenarios due to the factors and levels of Table 4.21. Caption: #Scn = Scenario number

Factor Combination	#Scn	Scenario
{Man, SatNorm, InsCmd}	1	Insert a single command to be transmitted in the Manual Operation Mode, with configuration = Satellite and configuration option = Normal
{Man, SatPreVer, InsGrp}	2	Insert a group of commands to be transmitted in the Manual Operation Mode, with configuration = Satellite and configuration option = Pre-Verification
{Man, TestNorm, UpTCmd}	5	Update the date/time to send a command. The command will be transmitted in the Manual Operation Mode, with configuration = Test and configuration option = Normal
{Auto, SatTx, -}	9	Insert commands from the FOP to be transmitted in the Automatic Operation Mode, with configuration = Satellite and configuration option = Transmission
{Auto, TestNorm, -}	11	Insert commands from the FOP to be transmitted in the Automatic Operation Mode, with configuration = Test and configuration option = Normal
{Super, SatPreVer, -}	14	Insert commands from the FOP to be transmitted in the Supervised Operation Mode, with configuration = Satellite and configuration option = Pre-Verification
{Super, TestNorm, -}	17	Insert commands from the FOP to be transmitted in the Supervised Operation Mode, with configuration = Test and configuration option = Normal
{Inv, SatNorm, -}	19	Change the operation mode from Supervised to Automatic or from Automatic to Supervised, with configuration = Satellite and configuration option = Normal (Robustness testing)
{Inv, SatTx, -}	21	Change the operation mode from Manual to Supervised or from Manual to Automatic when the time for transmitting the first command of the queue of commands has expired, with configuration = Satellite and configuration option = Transmission (Robustness testing)
{-, SatEx, Inv}	47	Update the date/time to send a command with configuration = Satellite and configuration option = Execution, but without firstly selecting the command with these characteristics in the queue of commands (Robustness testing)

not need to be changed and do not require manual intervention. Besides, the test designer does such a definition entirely in NL and thus no formalism is imposed to him/her;

b) *Define Scenarios.* The test designer must select factors and levels and interpret the factor combinations to derive scenarios. However, the factor combinations are automatically generated by an open source tool, *TConfig* (UNIVERSITY OF OTTAWA, 2008);

c) *Select and Input NL Requirements.* This is done manually by the test designer. But, handling of NL requirements is automatically accomplished by the SOLIMVA tool;

d) *Generate Model*, *Clear Requirements and Model.* These two activities are completely and automatically performed by the SOLIMVA tool. The test designer needs only to start these activities via Graphical User Interface of the SOLIMVA tool;

e) *Generate Abstract Test Cases.* This is automatically accomplished by the GTSC environment;

f) *Generate Executable Test Cases.* This is done by the test designer.

The SOLIMVA methodology requires basically three types of efforts from the test designer. First, it is the creation of a Dictionary which defines the application domain. However, from the point of view of test case generation addressing system and acceptance testing, in several situations the name of states in the Statechart models are not very relevant. What matters is the *input/output* pairs within the transitions in the Statechart model that will be translated into *test input data/expected result* in the Executable Test Suite. Hence, the cardinality of the Name set does not need to be high. Moreover, as SOLIMVA uses the "divide and conquer" approach, i.e. splitting the interactions with the IUT into smaller scenarios, it is not mandatory to define an extensive Hierarchy function (ordered pairs) in order to predict different types of COMPOSITE states. Hence, cardinalities of the domain ($Y.I_P$) and codomain ($Y.O_P$) of $S_{TM}.Y$ do not need to be high either. As an example, in order to address all 20 scenarios proposed by the expert for the SWPDC case study, N has cardinality equal to 9 while the cardinality of $Y.I_P$ and $Y.O_P$ is equal to 8 for each set. Therefore, very small cardinalities.

The Reactiveness function might or might not be huge. It depends on the way the deliverables are written. For instance, the SWPDC's Software Requirements Specification has several NL requirements where commands defined in the PDC-OBDH Communication Protocol Specification are explicitly mentioned. Commands such as PREPARE HOUSEKEEPING DATA which were abbreviated to PREP-HK, and TRANSMIT HOUSEKEEPING DATA abbreviated to TX-DATA-HK. In such situations, the test designer may prefer to add the ordered pair (*command, response*) in the Reactiveness function in order to make it easier the translation from the Abstract Test Suite into the Executable Test Suite. The explanation for this fact is that, since the ordered pair is in the Reactiveness function, it is very probable that they appear in the Statechart (abstract) model and thus they will be part of the Abstract Test Cases. Since commands are executable then the translation from the abstract perspective into the executable one is simple. In the SWPDC case study, the cardinality of both the domain, $R.I_E$, and codomain, $R.O_E$, of R is equal to 94.

However, if a requirements specification for space or other application domain has little explicit mention to low level commands or name of methods/routines then the Reactiveness function may not have many ordered pairs. In this case, the input events of the transitions of the Statechart model are pieces of NL sentences such as *only respond_request*, *have be energize_least 1 minute*, and the test designer shall translate them into executable *test input data/expected result.* Finally, the Dictionary is important because the lack of domain knowledge limits the applicability of systems based on unrestricted NL requirements (AMBRIOLA; GERVASI, 2006).

The second type of effort is the need to translate Abstract Test Suites into Executable Test Suites. However, it is more interesting for the test designer to analyze a set of abstract *input/output* pairs within the Abstract Test Cases and identify *test input data/expected results* of the Executable Test Cases rather than trying to find out what are the elements of a Executable Test Case directly from NL requirements documents. Besides, most of the translations accomplished for one scenario are reusable for other scenarios.

One may assert that the need to translate Abstract Test Suites into Executable Test Suites is a disadvantage of the SOLIMVA methodology. The point is that if the model is precise enough, some Model-Based Testing tools, including the GTSC tool, enable the generation of directly Executable Test Cases. Then, there would be no need to make such a translation. However, if NL requirements deliverables like the ones in the SWPDC case study are considered then it is important to realize that if a model has enough information so that a Model-Based Testing tool can generate Executable Test Cases it is because the test designer has translated from the NL requirements into the notation accepted by the tool before using the tool itself. Based on published literature and also taking into account commercial software products, it is very unlikely that there is a Model-Based Testing tool that can accept NL requirements documents and can, without any assistance from the user, generate directly Executable Test Suites. In other words, the translation from the abstract level into the executable level is manually done by the test designer before or after using the Model-Based Testing tool. In this PhD thesis, the decision was to make such a translation after using GTSC because the intention was to use NL requirements as closely as possible in their original form due to the fact that this is a more realistic approach.

The last type of effort is the definition of scenarios by using combinatorial designs. This is without any doubt the most demanding effort. The test designer must not only define factors and their respective levels but also to interpret the factor combinations to create the scenarios. He/she must decide whether a level must or must not be discarded in a certain factor combination and provide a set of scenarios that can cover most of the relevant interactions with the IUT. However, this is one of the main roles of a test designer within a Verification and Validation team and such a professional must have a significant knowledge of the application domain to perform scenarios identification. But, knowledge of the application domain is mandatory: it is very hard to envisage a test designer producing coherent/effective test cases with no or little expertise in the application domain. In short, the SOLIMVA methodology essentially requires knowledge of the application domain from the test designer.

Another important observation refers to the strategy of separation of test objectives proposed by the SOLIMVA methodology as shown in Section 4.1. The question is whether the fact of disjoining behaviors, related to test objectives, as proposed by the methodology will not result in sets of Executable Test Cases that do not take into account each individual behavior. This particularly relates to Robustness testing. The approach of combinatorial designs coupled with the expertise of the test designer in the application domain can avoid this kind of situation. For example, recall that the three test objectives of the expert's scenario 12, two of them related to Robustness testing, were covered by SOLIMVA as follows: two of the test objectives were addressed by unfolded scenario 73.2, and one by normal scenario 143. As a matter of fact, for instance, there are 35 factor combinations derived by the combinatorial designs algorithm with *Inv* as the level for the command (Cmd) factor. Therefore, there are 35 normal scenarios to address problems like inconsistent values within the command frames, incomplete reception of commands or any other robustness feature the test designer might wish. These 35 scenarios spread over different PDC's operation modes (OpMode), Services, ways of initializing PDC (StartMode).

Although the SOLIMVA methodology proposes differently, the test designer may not want to separate test objectives and/or to neglect the interpretation of a level within a factor combination. Considering the SWPDC case study, the factor combination 170 is: {Inv, Inv, Hk, Reset}. One interpretation would generate normal scenario 170 as follows: "verification of behavior of SWPDC when receiving commands with inconsistent values or receiving commands incompletely, changing PDC's operation mode to an unspecified operation mode, generation, formatting, and transmission of Housekeeping Data, and verification of the reset process of PDC." The many unrelated test objectives denote a bad approach. But, the decision is up to the test designer.

Regarding scalability, the number of normal and unfolded scenarios derived by the test designer can be huge if too many factors and/or levels were selected. It is the responsibility of the test designer to choose an appropriate number of factors and levels so that not many scenarios are created. For instance, the level HwSwHnd is related to handling of hardware and software parameters. This level could be broken into two, one for hardware parameters and other for software parameters. Joining these two features into one level decreases the number of scenarios. Although the number of scenarios and consequently Statechart models and Executable Test Suites can be huge, it is nowadays very common the use of frameworks and/or tools to automate the execution of test cases (SANTIAGO et al., 2008a).

It is not in the scope of this work to conduct a study on cost and effectiveness of test suites generated by Model-Based Testing approaches. Particularly, the effectiveness of model-based test suites generated by Statecharts and FSMs is heavily dependent on two aspects. First, the model must be suitable enough in order to derive the generation of coherent Executable Test Suites. In the SWPDC case study, the models created by the SOLIMVA tool were even superior in some respects than the models manually developed by an expert. The second point is the choice of the test criteria. This is an open issue in the academic community. Comparisons of fault detection effectiveness of some FSM (SIDHU; LEUNG, 1989; SOUZA, 2010) and Statechart (ANTONIOL et al., 2002; BRIAND et al., 2004) test criteria have been published but it seems that there is no definite answer with respect to this issue. The SOLIMVA methodology does not address this second point.

Note that in Section 4.1.2, the term "**characteristics**" of Executable Test Suites has been carefully chosen over efficiency since the latter has its own meaning in the context of Software Testing, that is, the ability of a test suite in finding defects in source code. However, it is possible to infer some observations on the test criteria for Statecharts mentioned in this work. Due to the models generated by the SOLIMVA tool and the definition of the all-simple-paths test criterion, this criterion demonstrated to be not as good as the all-transitions test criterion. One crucial situation was not in the all-simple-paths Executable Test Suite, i.e. occurence of severe problems during initialization of PDC. Assuming that there may be redundancies in some equipments of subsystems in a satellite, if an equipment exhibits a serious problem that would endanger its functioning it can be replaced with any copy of itself on board of the satellite or such equipment can be operated in a degraded mode. But in no hypothesis this problem in the equipment can propagate and cause significant problems in other subsystems so that the mission can be compromised. So do not test the behavior of the PDC/SWPDC in situations of serious problems during the startup of PDC, where in this case the PDC must continue in the Initiation Operation Mode, is a serious flaw.

In addition, in SOLIMVA's normal scenario 17 to address the reset process, the all-simple-paths test suite is not suitable at all to meet the test objective. Two Executable Test Cases compose the Executable Test Suite but within them there is no behavior (command) to reset PDC itself. So it is not possible to perceive the behavior of SWPDC during a reset process because PDC is not led to a reset. This situation is even worse and alert to the fact that it is necessary to evaluate not only the cost and efficiency of sets of test cases generated according to test criteria but even the achievement of test objectives in applying the test cases in real and complex projects. Of course, one can not conclude that the all-simple-paths test criterion is worse than the all-transitions criterion but these observations give rise to further analysis. In fact, theoretically, all-transitions and all-simple-paths are incomparable (SOUZA, 2000).

It is interesting to notice that different test criteria may suggest different interpretations when running the set of test cases. For instance, consider SOLIMVA's normal scenario 71 and the Executable Test Suites shown in Table 4.11. The all-paths-k-C0-configuration suite is composed of a single Executable Test Case. This test case is a kind of merge of test cases 1 and 2 of the all-transitions Executable Test Suite. When running these Executable Test Suites, the tester can interpret in a different way *Action6*. For the all-transitions Executable Test Suite and according to test case 1, he/she will switch PDC on (*Action1*) and a severe and permanent problem during initialization of PDC will be simulated (*Action6*) so that PDC is expected to remain in the Initiation Operation Mode. Then, PDC is turned off and on again and the second test case considers the normal behavior of processing where no problem during initialization of PDC occurred. For the all-paths-k-C0-configuration Executable Test Suite, PDC is switched on (*Action1*) only once and a severe but transient problem during initialization of PDC will be simulated (*Action6*). This means that the problem occurred but was transient. PDC has remained in the Initiation Operation Mode but since the problem no longer occurs, then the normal flow of processing could be performed (*Action2, VER-OP-MODE/Timeout, ...*).

A question that might be raised is why in both the expert's and SOLIMVA's models the general behavior is: switch the computer (PDC) on, do something, switch the computer off. This is unfeasible in practice because in many Executable Test Cases of a Executable Test Suite it is necessary to turn the computer on and off and, if all the Executable Test Suites derived for all scenarios are considered, this will almost certainly make unfeasible the execution of test cases in a real and complex project. The explanation for this fact is to make each scenario self-contained and thus, the Executable Test Suites are also self-contained. In other words, any scenario can be selected as being the first to have its test cases executed since all models assume that the computer will be turned on. Thus, the Verification and Validation team has the flexibility to begin running test cases from any scenario: not necessarily the Executable Test Suite derived for scenario number 1 must be the first to be executed.

Suppose that for a very trivial application only 9 normal scenarios were derived (neither simple nor unfolded scenarios were created). If the execution order of scenarios (Executable Test Suites) is 5, 6, 3, 9, 1, 2, 4, 7 and 8, then the initial test steps (which involve turning the system on, perform initialization processing) of test cases of the test suite of normal scenario 5 which are also common to normal scenario 6 are considered as already carried out (pre-conditions) when running the test suite of normal scenario 6. Therefore it is not necessary to switch the system off, turn it back on again, run the initial test steps so that the Executable Test Suite of normal scenario 6 can be executed: the process continues from the steps that are not common to the initial test steps of test cases of the test suite of normal scenario 5. The same analysis applies to scenarios 6 $\rightarrow$ 3, 3 $\rightarrow$ 9 and so on. Naturally, some test suites of certain scenarios require that the computer is turned on/off more than once given the characteristics of the test suites themselves (see explanation above of the all-transitions Executable Test Suite of SWPDC's normal scenario 71). But this need not be repeated for all scenarios.

The algorithm for automated detection of BSAO tuples (Figure 3.10) is not general enough to work out with all NL sentences. However, it is important to stress that there are guidelines to develop NL requirements in space application product development. For instance, ECSS has a standard to write Technical Requirements Specifications (ECSS, 2009). Among other features, this standard provides "recommendations for the wording of requirements" stating how to write requirements, how modal verbs should be used, and terms to be avoided within the NL requirements. Directives like this and experience in space application projects led the design and implementation of the algorithm for generating BSAO tuples.

However, even if the algorithm for generating BSAO tuples fails, the automated *refinement based on domain information* (Section 3.3.2.1) helps to eliminate incorrect tuples. Consider again NL requirement POCP021:

> POCP021 - The PDC may not receive a command sent in its entirety. After identifying the beginning of a command frame, the PDC shall wait two times MAX-TRANSM-DELAY for the rest of the command. If this stipulated time expires, a timeout shall occur, the PDC shall abort the communication, the command shall be discarded, an event report shall be generated, and the PDC shall wait for a new OBDH's command.

In normal scenario 143, one of the tuples created due to this requirement is:

- Behavior: ;
- Subject: command;
- Action: 111-be 112-discard;
- Object: event report.

A problem exists because "event report" is not Object but Subject of the phrase "an event report shall be generated" (recall the observations about Subject, Action, and Object of BSAO tuples made in Sections 3.3.1 and 3.3.1.1). However, the *refinement based on domain information* gets rid of the influence of this tuple so that no state/transition due to such a tuple appears in the final Statechart models. For the SWPDC case study, the cases in which there were incorrect BSAO tuples were satisfactorily treated by the aforementioned refinement.

The SOLIMVA methodology and the SOLIMVA tool are limited by possible defects within the incorporated (version 3.0 of the *Stanford POS Tagger* (TOUTANOVA et al., 2003), version 11.01 of the *Java WordNet::Similarity* (UNIVERSITY OF SUSSEX, 2010)) and external (version 2.1 of *TConfig* (UNIVERSITY OF OTTAWA, 2008)) tool-s/packages that were used. For instance, as any other POS tagger tool, the *Stanford POS Tagger* is not 100% free of failures. Its output is the basis for identifying the BSAO tuples. Consider the following requirement:

The SWPDC shall format scientific data from each EPP Hx (x = 1 or 2).

Version 3.0 of the *Stanford POS Tagger* recognizes "format" as a *common noun, singular or mass* (POS tag "NN") instead of a *verb, base form* (POS tag "VB"). This is a problem that the SOLIMVA methodology/SOLIMVA tool must not try to solve. It is not in the scope of this work to develop a new POS tagging algorithm and tool to identify lexical categories of words: there are many of them available, and the idea was to use a publicly available POS tagger (Stanford) to help with the primary goal which is to generate the Statechart models from NL aiming at system and acceptance model-based test case generation.

Similarly, undiscovered defects in version 11.01 of the *Java WordNet::Similarity* package will impact on the WSD refinement (Section 3.3.2.2) as it may generate incorrect values of weights on the edges of the sense dependency graph. Problems in version 2.1 of the *TConfig* tool will also affect the identification of scenarios (Section 3.2).

The following advantages should be emphasized if a test designer decides to apply version 1.0 of the SOLIMVA methodology and the SOLIMVA tool rather than relying on a completely manual ad hoc strategy like the expert's approach discussed in Section 4.1:

a) Compared to other formal methods, FSMs and Statecharts are relatively easy to understand. The SOLIMVA methodology aims to avoid that a test designer, who is not even experienced with these simple modeling techniques, needs to develop the models from scratch. It should be noted that professionals from aerospace application domain and graduate students find it difficult in translating NL software requirements specifications into Statecharts or FSMs in order to address system and acceptance testing. The SOLIMVA methodology requires the definition of the application domain, by means of a Dictionary, and scenarios from the user. However, the methodology and the SOLIMVA tool provide a "first" model so that the test designer can start working. If the test designer does not feel comfortable with the model, he/she can try to improve it (*manual refinement*);

b) As pointed out previously, the translation from the abstract level into the executable level is somehow manually accomplished by the test designer before or after using a Model-Based Testing tool. The SOLIMVA methodology proposes to make such a translation after using the tool (GTSC) in order to use NL requirements as closely as possible in their original form due to the fact that this is a more realistic approach. In addition, verifying a set of *input/output* pairs within the Abstract Test Cases rather than directly looking at NL requirements documents is a more feasible way to perform the translation from the Abstract Test Suite into the Executable Test Suite. Moreover, the Abstract Test Cases provide a concise notation and emphasize the most relevant NL sentences, allowing the test designer to generate more suitable Executable Test Suites;

c) The SOLIMVA methodology and the SOLIMVA tool allow to automatically start reading NL requirements. This fact added to the translation from the Abstract Test Suite into the Executable Test Suite make the methodology an interesting solution to minimize problems related to the incomplete/inconsistent creation of models for Model-Based Testing;

d) SOLIMVA suggests a precise, systematic and mathematical-based solution to identify scenarios for system and acceptance test case generation by means of combinatorial designs. In the end, the methodology attempts to answer a very important question in a Verification and Validation process: until when should the system be tested? Since the SOLIMVA methodology allows to identify the total and exact number of scenarios (simple, normal, unfolded), and consequently Executable Test Suites for system and acceptance testing, then this would be the upper limit;

e) The separation of test objectives proposed by SOLIMVA results in a set of scenarios with goals more closely related and therefore a better strategy is achieved.

On the other hand, if version 1.0 of the SOLIMVA methodology and the SOLIMVA tool are compared to some approaches presented in Chapter 2, the following advantages can be listed:

a) Ease of use. As mentioned in Section 3.2, the SOLIMVA methodology does not require any knowledge in formal methods and their respective notations to define the application domain from the user, like CIRCE (AMBRIOLA; GERVASI, 2006) and CARL (GERVASI; ZOWGHI, 2005) do. Moreover, the definition of the application domain as proposed in SOLIMVA seems to be far more simpler than the one presented in *Text Analyzer* (SNEED, 2007);

b) Writing of NL requirements. The SOLIMVA tool provides a certain freedom to the user to write NL requirements, but its approach is not unrestricted NL, like NL-OOPS (MICH, 1996), because it is necessary to define the application domain by means of a Dictionary. Thus, the SOLIMVA tool tends not to have the problems of limited applicability that are associated with unrestricted NL approaches. In addition, the design of the SOLIMVA tool is such that it is not necessary that the requirements are written strictly complying with predefined standards of writing as with very restricted controlled NL such as ACE (FUCHS et al., 1999; FUCHS et al., 2000);

c) Identification of scenarios. The SOLIMVA methodology proposes a formal manner, by means of combinatorial designs, to identify scenarios for system and acceptance test case generation. Other methodologies, like CoFI (AMBRÓSIO et al., 2007), do not adopt a mathematical-based solution in order to achieve such a purpose, and this fact might limit the strength of such proposals;

d) Semantics. From the perspective of model generation based on NL sentences, in some tools the user is required to manually provide explicit definitions for concepts. For instance, in CIRCE, the user provides such explicit definitions by means of *definitions*, an element of the *requirements document model* of the tool. In this case, it is possible to assert that the semantics of the model is somehow manually provided by the user. A similar observation can be made with respect to *Text Analyzer* when the user must identify keywords in the text. In the SOLIMVA tool, a proposal to automate the identification of the semantics related to the generated model was implemented by adapting a WSD algorithm in order to identify self transitions in the resulting Statechart model;

e) Automation. Although many publications presented in Chapter 2 support the automated translation from NL requirements into another notation, others do not have such a characteristic, for instance Kim and Sheldon (2004) and Liang and Palmer (1994) proposals. This is another important feature of the SOLIMVA tool;

f) Starting from NL requirements. Some publications, like the one of Fraser et al. (1991), aim to bridge the gap between informal and formal requirements specification languages. However, they do not begin to approach directly from NL requirements as the SOLIMVA methodology does;

g) Mathematical formalism. In order to generate the test cases, the SOLIMVA methodology and the SOLIMVA tool translate the NL requirements into a formal method, the Statecharts language. The MOR Editor (LU et al., 2008) lacks mathematical formalism.

As mentioned in Section 3.1, four deliverables were consulted to apply version 1.0 of the SOLIMVA methodology to the SWPDC case study: Requirements Baseline, Software Requirements Specification, PDC-OBDH Communication Protocol Specification, and PDC-EPPs Communication Protocol Specification. However, all documents developed within the scope of the QSEE project, including these four deliverables, were made in the Portuguese language. Therefore, these four selected documents were translated into the English language due to wide acceptance of English worldwide and thus being one of the languages most commonly used to prepare documents in software projects. In addition, INPE has international cooperation for the development of satellites and this results in that basically all documents associated with the development of software projects for satellites are also prepared in English. The mapping between the documents' titles in English and Portuguese is shown in Table 4.23.

Table 4.23 - Mapping between the documents' titles in English and Portuguese

Title in English	**Title in Portuguese**
Requirements Baseline	Requisitos de Base (INPE.CEA, 2006d)
Software Requirements Specification	Especificação Técnica do Software (INPE.CEA, 2006a)
PDC-OBDH Communication Protocol Specification	Protocolo de Comunicação PDC-OBDH (INPE.CEA, 2006c)
PDC-EPPs Communication Protocol Specification	Protocolo de Comunicação PDC-EPPs (INPE.CEA, 2006b)

A total of 97 requirements, collected from the four documents, were used in order to generate the Statechart models and the Executable Test Cases to cover the 20 scenarios of the expert. Of course, it is possible to find more than 97 requirements in these four documents. However, only those requirements necessary to make it possible to cover the expert's scenarios were considered. Requirements were collected as follows: 12 from the Requirements Baseline, 64 from the Software Requirements Specification, 20 from the PDC-OBDH Communication Protocol Specification, and 1 from the PDC-EPPs Communication Protocol Specification.

Logically, some of these requirements have been somehow created in NL. But there are also UML diagrams such as sequence diagrams and activity diagrams in these deliverables. Thus, some of these 97 requirements were extracted from these diagrams and converted into NL requirements. However, "new" NL requirements had to be created by the test designer in order to generate coherent test cases. These requirements were not present in any of the documents, including the Software Requirements Specification, consulted to apply SOLIMVA. So, for instance, when it states that 64 requirements were "collected" from the Software Requirements Specification this means that some of these NL requirements really existed and were used. However, some other requirements did not exist and they were therefore created. If the Software Requirements Specification was more complete, it would not be necessary to create such NL requirements as they would already be in the document. Appendix A presents all 97 NL requirements as well as some additional considerations are made on this matter.

The lack of information (requirements) in software specifications is a well known problem: incompleteness. Next chapter proposes an approach to address this severe defect present in software specifications.

5 AN APPROACH TO DETECT INCOMPLETENESS IN SOFTWARE SPECIFICATIONS

As stated at the end of the previous chapter, the test designer created some NL requirements in order to generate test cases considering the SWPDC case study. Consider the following two requirements of the SWPDC's Software Requirements Specification:

- Housekeeping Data should be generated automatically and continuously by SWPDC every 600 seconds (default value).
- The time for generating Housekeeping Data may be changed by means of command sent by the OBDH. The minimum value of time for generating Housekeeping Data is 60 seconds while the maximum value is 1,000 seconds, with resolution in second.

When reading these requirements, the test designer might wonder when, in fact, Housekeeping Data can be received for the first time by the OBDH. So, questions might be:

> *Is that right after one minute from when the PDC has been energized which is the time that must elapse for the OBDH can effectively communicate with the PDC (NL requirement POCP001)? Therefore, would it be possible to receive Housekeeping Data right after this time by means of commands?*

The answer to both questions is no. Only after at least 600 seconds have been elapsed, which is the default value for the period (time interval) in which Housekeeping Data are generated, the PDC can send Housekeeping Data to the OBDH. This explanation is not clearly described in the deliverables consulted to apply version 1.0 of the SOLIMVA methodology. Thus, the test designer literally created a new NL requirement to deal with this situation:

> SRS004 - The OBDH should wait 600 seconds before asking for a Housekeeping Data frame.

In fact, the ideal behavior for the OBDH to get Housekeeping Data is: wait the current time (60, 80, 200, 600 seconds, or whatever other admissible value) to generate

Housekeeping Data and after that, send the commands related to transmission of such data (PREPARE HOUSEKEEPING DATA, one or more TRANSMIT HOUSEKEEPING DATA, etc.). This is not clear in the documentation too. Of course, after PDC is operating for a long period, Housekeeping Data will be continuously generated and eventually it may not be necessary to wait any time for receiving Housekeeping Data. However, it is always advisable that the OBDH wait, at least, the current time to generate Housekeeping Data, not only in the first but also in other attempts, before requesting the data themselves to provide time for data being generated.

When discussing expert's scenario 12 (Section 4.1.1), it was stated that the command *P_DMP-DataP0-7FFFH-BFFFH* (PREPARE DUMP DATA FROM PAGE 0 OF DATA MEMORY, WITH INITIAL ADDRESS = 7FFFH AND FINAL ADDRESS = BFFFH) must not be processed by SWPDC because the initial memory address is lower than the minimum memory address allowed for all pages of the Data Memory. However, this was a customer's (INPE) requirement, but that was not described in any documents. There is no clear indication about this requirement and therefore one can not know what is the behavior that SWPDC must have (if the command must or must not be processed) just by looking at the documentation generated for the SWPDC software product. This information is missing in both documents generated early within the software development lifecycle, such as the SWPDC's Software Requirements Specification, and documents generated in later stages of the lifecycle, such as the SWPDC's User Manual.

These are just some examples of a very serious problem that is present in software specifications[1]: **incompleteness**. As explained, since software specifications are created early within the software development lifecycle, their defects affect the next software artifacts, including source code, to be developed. For instance, test designers take into account software requirements specifications to develop system and acceptance test cases. Developers consider software requirements specifications and related artifacts to create software design documents and, ultimately, develop the source code. Thus, the existence of defects such as incompleteness almost certainly will generate source code that does not meet the undisclosed goals of the customer, and may result in the generation of incoherent system and acceptance test cases.

[1]In this chapter, the term "software specifications" is used to encompass not only software requirements specifications but also other relevant artifacts, such as communication protocol specifications, consulted for the analysis of incompleteness. In addition, there is often an initial behavioral modeling of the software in software requirements specifications.

This chapter presents an extension of version 1.0 of the SOLIMVA methodology in order to address the secondary goal of this PhD thesis: detecting incompleteness in software specifications. Model Checking (BAIER; KATOEN, 2008) combined with k-permutations of n values of variables and specification patterns (DWYER et al., 1999) were used to tackle this problem. Thus, this extension generated version 2.0 of the SOLIMVA methodology. The new and most important activity of version 2.0 of SOLIMVA was applied to the SWPDC case study. In this chapter when used without any explicit indication of the version number, the terms "SOLIMVA methodology" and "SOLIMVA" refer to version 2.0 of the SOLIMVA methodology.

5.1 Description of version 2.0 of the SOLIMVA methodology

Version 2.0 of the SOLIMVA methodology with the new activities created to address the problem of incompleteness in software specifications is shown in Figure 5.1. The new activities are *Analyze Incompleteness* and *Improve Specifications*. It is important to note that the same activities, with the same features, present in version 1.0 of the SOLIMVA methodology (Figure 3.3) are also present in version 2.0 of the methodology. Moreover, with regard to the activities that already exist in version 1.0, the workflow is essentially the same in version 2.0 of the SOLIMVA methodology. The only difference is that the execution of these new activities (*Analyze Incompleteness* and *Improve Specifications*) should proceed in parallel with the *Define and Input Dictionary* activity where the Dictionary starts to be defined.

The most important activity to deal with the problem of incompleteness is *Analyze Incompleteness* which will be described in Section 5.1.1. It is by means of this activity that incompleteness defects are truly detected. Once incompleteness defects are detected, the quality of the assessed software specifications can be improved by completing the documents when necessary. This is the *Improve Specifications* activity shown in Figure 5.1. Therefore the *Improve Specifications* activity does not relate effectively to the second objective of this PhD thesis, which is only the detection of incompleteness defects in software specifications. However, in order to make the methodology most appropriate to certain processes for improving the quality of documents, such activity was incorporated into version 2.0 of the SOLIMVA methodology (more on this topic in Section 5.3).

Before proceeding with the description of the activity related to detection of incompleteness it is important to define what is meant by an incomplete software specification. The definition below is in accordance with the *IEEE Guide to Software Requirements Specifications* (IEEE, 1984). However, only the characteristics defined

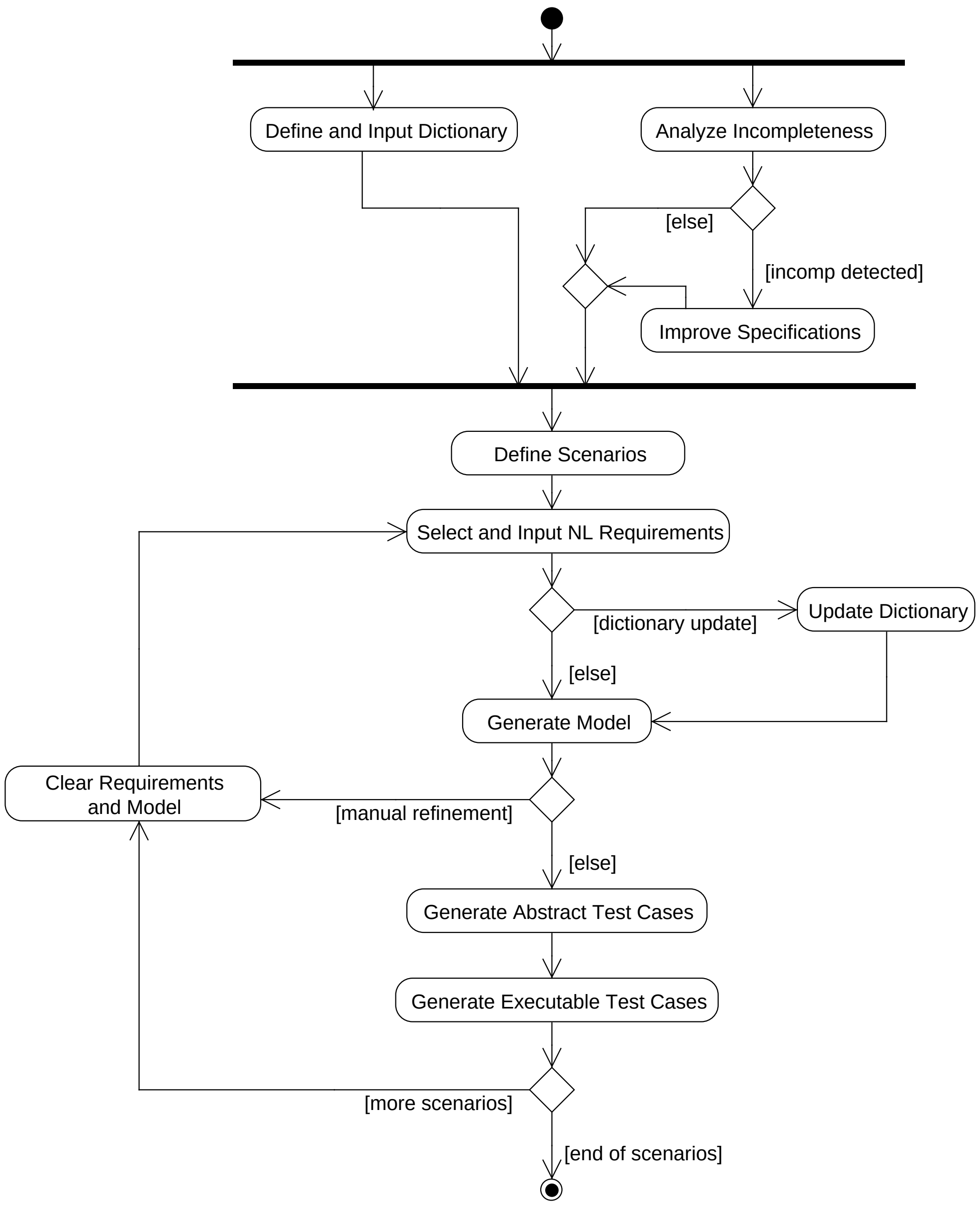


Figure 5.1 - Version 2.0 of the SOLIMVA methodology. Caption: incomp = incompleteness defect

by this IEEE standard that are most related to this work are mentioned in this definition.

Definition 5.1. A software specification is **complete** if it possesses the following qualities:

- Inclusion of all significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces;
- Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to valid and invalid input values.

If a software specification does not satisfy any aspect of the these two qualities then this specification is considered incomplete. By the definition given above, it is evident that it is extremely difficult to have a complete software specification especially if complex software products such as software embedded in satellite computers are taken into account. The combination of requirements, valid and invalid input data, conditions results in a high number of possibilities to be considered so it is very difficult to predict all of these situations.

There are some definitions of defects' severity in the literature. However, in the context of this PhD thesis, a new proposal for defects' severity will be defined and used as follows.

Definition 5.2. Severity of an incompleteness defect found in software specifications is:

- Low: development team and/or Verification and Validation team can easily deal with the lack of information;
- Medium: development team and/or Verification and Validation team produce solutions that are not the best but that may be acceptable in the "desirable" features of the software product;
- High: development team and/or Verification and Validation team produce solutions that are far outside of the "desirable" characteristics of the software product, and this may result in incorrect source code and/or test cases generated inconsistently.

Observe that this definition of severity is in fact an estimate of what may occur in the event that the incompleteness defect is not detected and eliminated from the software specification. For example, in the case of High severity, most likely source code will be developed with some defects because such an implementation will not be, at the end, in accordance with what the customer actually wants (because it was not clear in the documentation). Moreover, it is likely that test cases are poorly designed with, for example, incorrect expected results.

5.1.1 The *Analyze Incompleteness* activity

The proposal to identify incompleteness in software specifications is summarized in Figure 5.2. The process presented in Figure 5.2 is very much like the traditional process to apply Model Checking (BAIER; KATOEN, 2008). However, there are important changes of philosophy which differ the way Model Checking is applied within the *Analyze Incompleteness* activity. As stated in Section 2.1.2.1, in the traditional approach, properties are generated based on requirements documents and they are formalized using some sort of temporal logic. In the context of software development, software specifications and their requirements are the basis to generate properties. On the other hand, in the traditional process, the finite-state model (transition system) describes the behavior of the system. However, the goal of this approach is to detect defects in software specifications and then there is no sense to use software specifications and their requirements as being the truth to be followed. Indeed, they are the objects of analysis to see whether there are problems of incompleteness. Therefore, software specifications and their requirements are used not to develop properties but rather to develop the model of the system. This explain why the analyst takes a look at software specifications (*Study Software Specifications* sub-activity) to develop the model of the system (*Model the System* sub-activity) in Figure 5.2.

However, it is necessary to generate the properties preferably in a way independent of the requirements in the software specifications. This is achieved by combining specification patterns (DWYER et al., 1999) with 2-permutations of n values of characteristics (*Use Specification Patterns + 2-Permutations of* n *Analysis* sub-activity). Similarly, specification patterns are not used in the traditional manner where, based on a requirement, a pattern and the scope within the pattern that mostly characterize such requirement are identified and properties are then generated in LTL, CTL. A different interpretation of pattern/pattern scope is provided aiming at detecting incompleteness defects (this will be discussed later). Of course, properties are also formalized (*Formalize Properties* sub-activity) in this proposal.

The model of the system is simulated prior to Model Checking (*Simulate the Model*

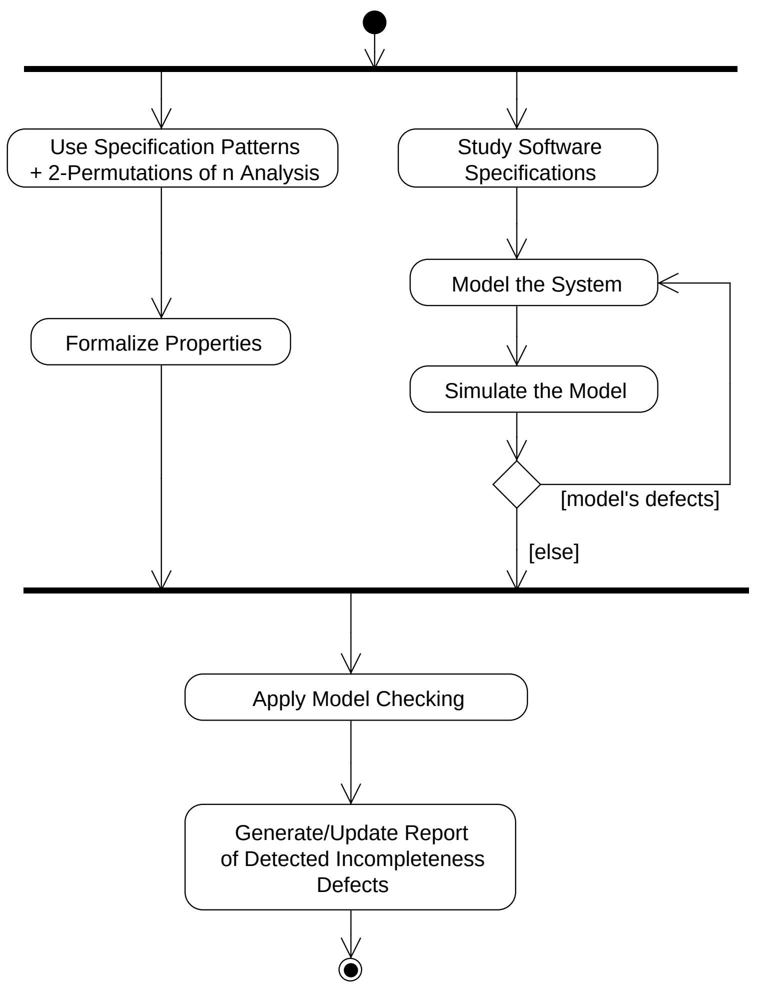


Figure 5.2 - Sub-activities of the *Analyze Incompleteness* activity

sub-activity) in order to get rid of modeling defects. Eliminating simple modeling defects before any form of thorough checking occurs may reduce the time-consuming verification effort (BAIER; KATOEN, 2008). When there is no more remaining defect in the model and all properties are created, Model Checking is applied (*Apply Model Checking* sub-activity). Detected incompleteness defects are then reported in a certain type of document (*Generate/Update Report of Detected Incompleteness Defects* sub-activity).

The process shown in Figure 5.2 is a simplified view of the *Analyze Incompleteness* activity. Moreover, this process assumes that the formalization of properties and the generation/simulation of the model are done in parallel which implies that there must be more than one professional in the organization that has expertise in Model Checking, which is not always the reality. Figure 5.3 shows, in form of an activity diagram, a more detailed and sequential view of the *Analyze Incompleteness* activity while Figure 5.4 shows this same perspective in form of a procedure. In the sequence, the explanations about the *Analyze Incompleteness* activity are given taking into account the form of procedure that describes the activity (Figure 5.4).

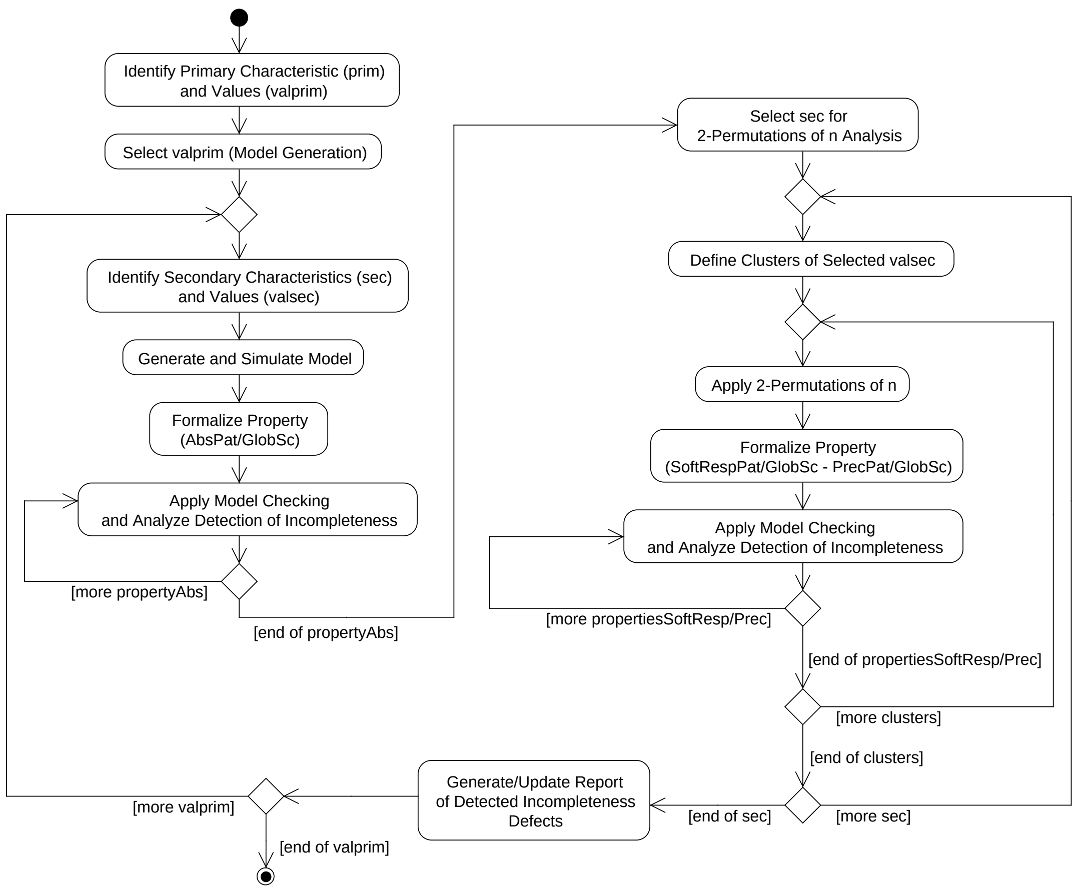

Identify Primary Characteristic (prim) and Values (valprim)
Select valprim (Model Generation)
Identify Secondary Characteristics (sec) and Values (valsec)
Generate and Simulate Model
Formalize Property (AbsPat/GlobSc)
Apply Model Checking and Analyze Detection of Incompleteness
[more propertyAbs]
[end of propertyAbs]
Select sec for 2-Permutations of n Analysis
Define Clusters of Selected valsec
Apply 2-Permutations of n
Formalize Property (SoftRespPat/GlobSc - PrecPat/GlobSc)
Apply Model Checking and Analyze Detection of Incompleteness
[more propertiesSoftResp/Prec]
[end of propertiesSoftResp/Prec]
[more clusters]
[end of clusters]
[end of sec]
[more sec]
Generate/Update Report of Detected Incompleteness Defects
[more valprim]
[end of valprim]

Figure 5.3 - More detailed and sequential view of the *Analyze Incompleteness* activity: activity diagram

First, it must be chosen a primary characteristic ($prim$) and its values ($valprim_i$; line 1). A primary characteristic is an attribute obtained from software specifications that identifies the main states of the software product. For instance, in space application software, the *operation mode* is a good candidate to be a primary characteristic. The analyst must then select some $valprim_i$ in order to generate finite-state models (line 2). For each selected $valprim_i$, secondary characteristics (sec_j), which are other attributes of software specifications, and their values ($valsec_{jk}$) must be obtained (line 4). Then, a finite-state model for each selected $valprim_i$ must be generated and

```
input  : software specifications
output: report of detected incompleteness defects
Identify a primary characteristic ($prim$) and its values ($valprim_i, 1 \leq i \leq l$);
Select some $valprim_i$ for model generation;
foreach $selected\ valprim_i$ do
    Identify secondary characteristics ($sec_j, 1 \leq j \leq m$) and their values ($valsec_{jk}, 1 \leq k \leq n$);
    Generate and simulate a model for $valprim_i$;
    foreach $valsec_{jk}$ do
        Formalize property according to AbsPat/GlobSc: $propertyAbs_{jk}$;
    end
    forall the $propertyAbs_{jk}$ do
        Apply Model Checking;
        if $(propertyAbs_{jk} = false \land expected = false) \lor (propertyAbs_{jk} = true \land expected = true)$ then
            No incompleteness defect detected;
        else
            if $(propertyAbs_{jk} = false \land expected = true) \lor (propertyAbs_{jk} = true \land expected = false)$
            then
                Incompleteness defect detected;
            end
        end
    end
    Select some $sec_j$ for 2-permutations of n analysis;
    foreach $selected\ sec_j$ do
        Define clusters ($cluster_p, 1 \leq p \leq q$) of selected $valsec_{jk}$;
        foreach $cluster_p$ do
            Apply 2-permutations of n;
            foreach $permutation\ t\ not\ addressed\ by\ previous\ clusters$ do
                Formalize property according to SoftRespPat/GlobSc: $propertySoftResp_t$;
                Formalize property according to PrecPat/GlobSc: $propertyPrec_t$;
            end
            forall the $propertySoftResp_t/propertyPrec_t\ pair$ do
                Apply Model Checking;
                if $(propertySoftResp_t = false) \lor (propertySoftResp_t = true \land propertyPrec_t = true)$
                then
                    Incompleteness defect detected;
                else
                    if $(propertySoftResp_t = true \land propertyPrec_t = false)$ then
                        No incompleteness defect detected;
                    end
                end
            end
        end
    end
    Generate/Update report of detected incompleteness defects;
end
```

Figure 5.4 - More detailed and sequential view of the *Analyze Incompleteness* activity: procedure

also simulated to determine and correct possible defects which arise by translating the software specifications into the model (line 5). Note that a model generated for a certain $valprim_i$ may not contain all the other $valprim_i$. This depends on the $valprim_i$ that was selected. For example, if $l = 5$, i.e. $prim$ has 5 values, and if $valprim_3$ was selected so that a model will be generated, then it is possible that only $valprim_3$, $valprim_1$, and $valprim_4$ can be considered in the model for $valprim_3$. Similarly, the secondary characteristics (sec_j) and their values ($valsec_{jk}$) may vary depending on $valprim_i$.

Specification patterns come into picture to formalize properties in CTL for each $valsec_{jk}$ and according to the **Absence Pattern and Globally Scope** ($propertyAbs_{jk}$; line 7). Each property is generated as follows:

$$\forall\Box\neg(prim = valprim_i \wedge sec_j = valsec_{jk}).$$

Thus, if the model does not satisfy this property ($false$) this means that $valsec_{jk}$ is present in the software specifications regarding $valprim_i$; otherwise, if the property is satisfied ($true$) then it is not present. However, this interpretation of being or not present in the software specifications will be better explained in Section 5.2. But, the analyst must predict in advance whether the property must or must not be satisfied (*expected*; lines 11 and 14). In other words, if $valsec_{jk}$ must appear in the software specifications then the expected result is "the property must not be satisfied" ($false$). On the other hand, if $valsec_{jk}$ must not appear, the expected result is "the property must be satisfied" ($true$). The following definition determines when an incompleteness defect is detected due to the Absence Pattern and Globally Scope.

Definition 5.3. Detection of incompleteness defect due to the Absence Pattern and Globally Scope: an incompleteness defect is detected if there is a discrepancy between the satisfaction or non-satisfaction of the property by the model and the expected result. That is, if the property is satisfied and it was not expected to be satisfied or vice versa.

After that, some sec_j must be selected for 2-permutations of n analysis (line 19) and, for each chosen sec_j, clusters ($cluster_p$) of selected $valsec_{jk}$ are determined (line 21). Application of 2-permutations of n is then accomplished for each $cluster_p$ (line 23), and each permutation $t = \{t1, t2\}$ will derive two types of CTL properties. The first property is based on the **Soft Response Pattern and Globally Scope** ($propertySoftResp_t$; line 25). This new pattern/pattern scope is a variation of the

Response Pattern and Globally Scope (Section 2.1.2.1) proposed by Dwyer et al. (1999). The definition of the Soft Response Pattern and Globally Scope with the corresponding CTL formula is as follows.

Definition 5.4. Soft Response Pattern and Globally Scope: a state/event p **should** be followed in some path by a state/event q within the entire program/model execution. CTL formula: $\forall\Box(p \to \exists\Diamond q)$.

Soft Response Pattern and Globally Scope is not as demanding in terms of satisfaction compared to the original Response Pattern and Globally Scope. In other words, it is easier for a model to satisfy Soft Response Pattern than the original Response Pattern. This is because the Soft Response Pattern and Globally Scope requires that if p occurs then there is some path ($\exists$ path quantifier) in which q occurs following p. So it does not require that to happen for all the paths ($\forall$ path quantifier) as the original Response Pattern demands, and thus Soft Response Pattern is not as demanding as the Response Pattern. Each property is then generated as follows:

$$\forall\Box((prim = valprim_i \wedge sec_j = valsec_{t1}) \to \\ \exists\Diamond(prim = valprim_i \wedge sec_j = valsec_{t2})).$$

Observe that $valsec_{t1}$ means that this is the $valsec_{jk}$ which is the first element of permutation t as well as $valsec_{t2}$ is the $valsec_{jk}$ which is the second element of t. The second property is according to the **Precedence Pattern and Globally Scope** ($propertyPrec_t$; line 26). The property is:

$$\neg\exists[\neg(prim = valprim_i \wedge sec_j = valsec_{t1}) \cup ((prim = valprim_i \wedge \\ sec_j = valsec_{t2}) \wedge \neg(prim = valprim_i \wedge sec_j = valsec_{t1}))].$$

After all properties are formalized, Model Checking is applied (line 29). If $propertySoftResp_t$ is not satisfied then an incompleteness defected is detected. If $valsec_{t1}$ is present in a model that represents the software specifications then the Soft Response Pattern and Globally Scope requires that there is some path in which $valsec_{t2}$ is also present in such a model following $valsec_{t1}$. The values of secondary characteristics ($valsec_{jk}$) that form a certain $cluster_p$ must have some kind of behavioral relationship and, therefore, it is expected that if $valsec_{t1}$ of a permutation t is present in a model that represents the software specifications then in some path $valsec_{t2}$ should also be present. Therefore, the interpretation is that if this property does not hold then the software specifications were not complete enough to predict such behavior, i.e. $valsec_{t2}$ occurs as a "response" to $valsec_{t1}$. Hence an incompleteness defect is detected.

Being $propertySoftResp_t$ satisfied then the result depends on $propertyPrec_t$. If $propertyPrec_t$ is also satisfied then an incompleteness defect is also detected. The reasoning behind this is that if $propertyPrec_t$ is satisfied it is because, in a model that represents the software specifications, $valsec_{t1}$ always occurs before (precede) $valsec_{t2}$ and this shows that the software specifications were not complete enough to predict the opposite, namely, $valsec_{t2}$ occurs before $valsec_{t1}$. However, the non-satisfaction of $propertyPrec_t$ indicates that no incompleteness defect exists. The definition below summarizes when an incompleteness defect is detected due to the combination of these two pattern/pattern scopes.

Definition 5.5. Detection of incompleteness defect due to the Soft Response Pattern and Globally Scope/Precedence Pattern and Globally Scope: an incompleteness defect is detected if $propertySoftResp_t$ is not satisfied or if both $propertySoftResp_t$ and $propertyPrec_t$ are satisfied.

As shown in Figure 5.4, the main actions of the *Analyze Incompleteness* activity (lines 3 to 41) must be accomplished for each generated model (for each selected $valprim_i$).

5.2 Case study: SWPDC software product

The new and most important activity (*Analyze Incompleteness*) of version 2.0 of the SOLIMVA methodology was applied to the SWPDC case study (SANTIAGO et al., 2007). The following deliverables were evaluated to detect incompleteness: SWPDC's Software Requirements Specification and PDC-OBDH Communication Protocol Specification. The primary characteristic (*prim*) is the PDC's operation mode (*opmode*) which has four possible values: $valprim = \{initiation, safety, nominal, diagnosis\}$.

The analysis was performed considering only the main operation mode of PDC, the Nominal Operation Mode ($valprim_3 = nominal$). Then, four secondary characteristics were selected. Thus, $sec = \{service, cmd, resp, action\}$ where:

- *service* represents services supported by SWPDC (5 values);
- *cmd* are commands defined in the PDC-OBDH Communication Protocol Specification (46 values; some of these values are related to aspects of

robustness of SWPDC);

- *resp* are responses defined in the PDC-OBDH Communication Protocol Specification (16 values; some of these values are related to aspects of robustness of SWPDC);
- *action* represents some actions that SWPDC shall perform (3 values).

Table 5.1 shows the characteristics and their values for developing the finite-state model for $valprim_3 = nominal$. As noted earlier, not all $valprim_i$ were considered to develop the model. Based on the software specifications, the goal was to model the behavior considering only the Nominal Operation Mode which is an operation mode that PDC can be long after the Initiation Operation Mode (*initiation*). Thus, it is not very relevant to include the Initiation Operation Mode in this model.

Table 5.1 - Characteristics and values for developing the finite-state model for $valprim_3 = nominal$

Characteristics	**Values**
$prim = opmode$	$valprim = \{$*safety, nominal, diagnosis*$\}$
$sec_1 = service$	$valsec_1 = \{$*sci, hk, dmp, load, noserv*$\}$
$sec_2 = cmd$	$valsec_2 = \{$*chOM, stDA, reDA, txSci, txHk, txDmp, txDg, txTst, retAnsw, prepHk, prepDmp, prepDg, prepTst, ldDat, exPrg, chPHkTm, chPIniPtr, chPSmpTm, txSciEBuf, txHkEBuf, txDmpEBuf, ldDatInvAddr, ldDatInvCSC, exPrgInvCKS, chPHkTmInvdat, chPSmpTmInvdat, chPIniPtrInvdat, prepDmpInvMemSelect, prepDmpInvIniAddr, prepDmpInvFinAddr, prepDmpInvAddrInterv, txDmpInvtype, txDmpInvlen, txDmpInvcks, prepDmpInvtype, prepDmpInvlen, prepDmpInvcks, ldDatInvtype, ldDatInvlen, ldDatInvcks, chPHkTmInvtplncks, chPIniPtrInvtplncks, chPSmpTmInvtplncks, txSciinc, txHkinc, txDmpinc*$\}$
$sec_3 = resp$	$valsec_3 = \{$*cmdRec, noData, sciData, hkData, dmpData, dgData, tstData, ldStatOk, ldStatErrCKS, ldStatErrCSC, ldStatErrAddr, timeout, sciDatainc, hkDatainc, dmpDatainc, noresp*$\}$
$sec_4 = action$	$valsec_4 = \{$*inidtacq, stopdtacq, noaction*$\}$

Note that in the secondary characteristic *cmd*, the values are abbreviations for commands defined in the PDC-OBDH Communication Protocol Specification. Thus, $chOM$ is an abbreviation for the command CHANGE OPERATION MODE. But, there are several values that are related to aspects of robustness of SWPDC. For instance, in the service for loading new programs on the fly into PDC's Data Memory,

$exPrgInvCKS$ refers to the cases where the checksum recalculated of the new executable code and the received checksum are unequal and then the new program should not be executed (Section 4.1.1). On the other hand, $chPHkTmInvdat$ is a command to CHANGE SOFTWARE PARAMETERS - TIME INTERVAL TO GENERATE HOUSEKEEPING DATA. But, the new time interval is an invalid value, i.e. it is out of range of possible values for this parameter. There are also values that cover situations where a command is not entirely received by SWPDC, e.g. *txSciinc* which means that, for some reason, not all the bytes that make up the command TRANSMIT SCIENTIFIC DATA came to PDC.

Likewise, the values of the secondary characteristic *resp* are abbreviations for the responses defined in the PDC-OBDH Communication Protocol Specification such as *cmdRec* which is the COMMAND CORRECTLY RECEIVED response. Robustness is also in order: $ldStatErrCKS$ means STATUS OF THE LOADING PROCESS - CHECKSUM ERROR which is the expected response for $exPrgInvCKS$. In addition, some values also address the opposite situation, i.e. PDC received a command but it was the response that PDC sent that was not completely received by the OBDH. For example, *sciDatainc* is the incomplete reception of a SCIENTIFIC DATA response.

The model generated for $valprim_3 = nominal$ has 71 reachable states. Figure 5.5 shows a small part of this model. The NuSMV (FONDAZIONE BRUNO KESSLER / CARNEGIE MELLON UNIVERSITY / UNIVERSITY OF GENOVA / UNIVERSITY OF TRENTO, 2011) was the Model Checker selected for use. In each state, there are the values that the characteristics (variables) take. For instance, in the upper leftmost state: $opmode = nominal$, $service = sci$, $cmd = txSci$, $resp = sciDatainc$, and $action = inidtacq$.

The atomic propositions in the transition system (finte-state model) depend on the properties under consideration (BAIER; KATOEN, 2008). However, a very simple choice is to let each value of each characteristic acts as an atomic proposition. Hence, *nominal* is considered an atomic proposition, it can be *true* or $false$, as well as all other values of the characteristics. In this sense, the set of atomic propositions, AP, is:

$$\begin{gathered} AP = \{safety, nominal, diagnosis, sci, hk, \ldots, chOM, stDA, \ldots, \\ cmdRec, noData, \ldots, inidtacq, stopdtacq, noaction\}. \end{gathered}$$

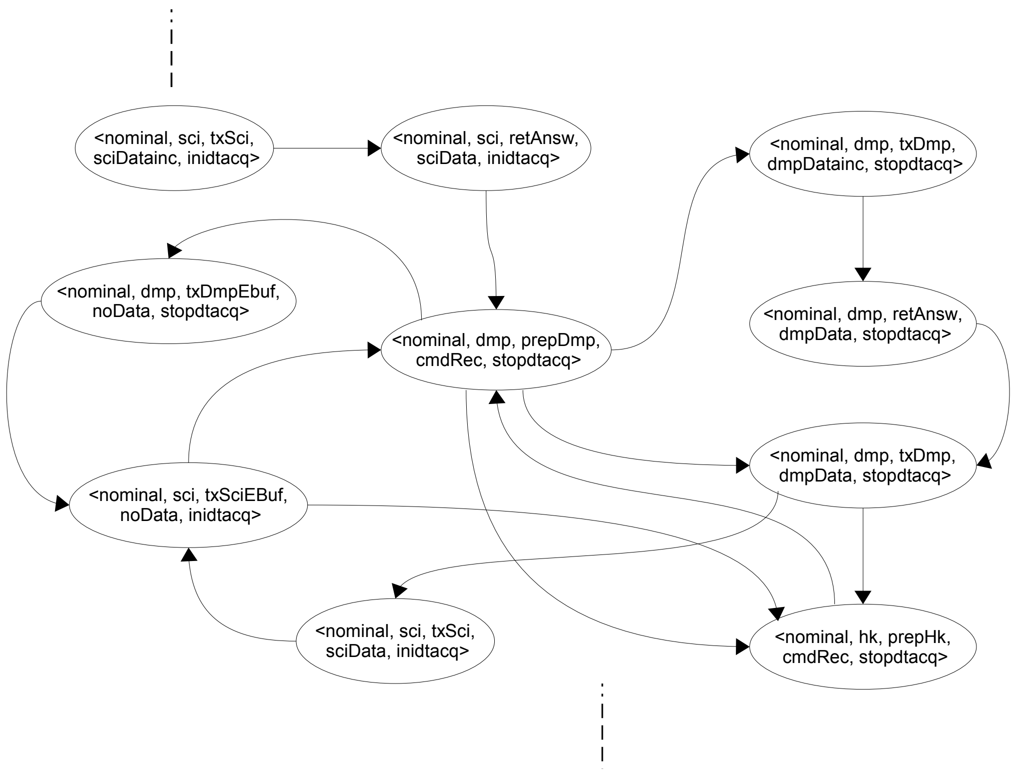


Figure 5.5 - A small part of the finite-state model for $valprim_3 = nominal$

The labeling function, L, for each state s is obtained directly. $L(s)$ intuitively stands for exactly those atomic propositions $a \in AP$ which are satisfied by state s (BAIER; KATOEN, 2008). Therefore,

$$L(< nominal, sci, txSci, sciDatainc, inidtacq >) = \{nominal, sci, txSci, \\ sciDatainc, inidtacq\}.$$

After generating and simulating the model, 70 CTL properties were generated according to the Absence Pattern and Globally Scope. For instance, in order to verify whether the command TRANSMIT SCIENTIFIC DATA ($txSci$) and the response HOUSEKEEPING DATA ($hkData$) are present in the software specifications, these CTL properties were created:

$$\forall\Box\neg(opmode = nominal \wedge cmd = txSci),$$
$$\forall\Box\neg(opmode = nominal \wedge resp = hkData).$$

This template of CTL property was repeated for each $valsec_{jk}$ where $1 \leq j \leq 4$. By applying Model Checking, 5 incompleteness defects were detected due to $propertyAbs_{jk}$ as shown in Table 5.2. The impact's attributes of the Orthogonal Defect Classification (ODC) (CHILLAREGE et al., 1992) are Installability and Documentation for these 5 incompleteness defects, and defects' severity range from Low to High.

Table 5.2 - Incompleteness defects due to the Absence Pattern and Globally Scope. Caption: Exp = Expected; Install = Installability; Docum = Documentation; R = set of reachable states

Property	**Exp**	**Impact**	**Severity**	**Unknown Behavior**
$Abs_{jk} = true$	$false$	Install	Medium	$ldDatInvAddr \notin R \Rightarrow$ situations where the address within a $ldDat$ command is invalid
$Abs_{jk} = true$	$false$	Install	High	$exPrgInvCKS \notin R \Rightarrow$ situations where the Checksum of the entire new executable code within an $exPrg$ command sent to SWPDC does not match the Checksum recalcutated by SWPDC after loading the entire new executable code
$Abs_{jk} = true$	$false$	Docum	Low	$prepDmpInvIniAddr \notin R \Rightarrow$ situations where the initial memory address within a $prepDmp$ command is invalid
$Abs_{jk} = true$	$false$	Docum	Low	$prepDmpInvFinAddr \notin R \Rightarrow$ situations where the final memory address within a $prepDmp$ command is invalid
$Abs_{jk} = false$	$true$	Docum	Low	$noresp \in R \Rightarrow$ responses for $prepTst$ and $prepDg$ commands in the Nominal Operation Mode are not completely known

Observe that the first four incompleteness defects occurred due to the fact that it was expected that $propertyAbs_{jk}$ was violated. However, such a property was satisfied. It is very important to understand the reason why the incompleteness defect was detected. In these cases, any of the values ($ldDatInvAddr$, $exPrgInvCKS$, ...) were not present in any state of the set of reachable states. However, this does not necessarily mean that there is no information about such values in the software specifications. For instance, it is clear in the PDC-OBDH Communication Protocol Specification that if the $exPrgInvCKS$ command is sent to PDC then the expected response is $ldStatErrCKS$. But, what it is not clear it is how exactly SWPDC behaves in case there is a discrepancy between the received and recalculated checksums when trying to load a new program (executable code) into PDC's Data

Memory.

The last incompleteness defect in Table 5.2 was due to the otherwise, i.e. $propertyAbs_{jk}$ should be satisfied and it was not the case. Transmission of Test and Diagnosis Data is not allowed in the Nominal Operation Mode. However, it is not clear what is the behavior of SWPDC if it receives commands to PREPARE TEST DATA ($prepTst$) and PREPARE DIAGNOSIS DATA ($prepDg$). Thus, NO CLEAR RESPONSE ($noresp$) appears in reachable states where $prepTst$ and $prepDg$ are also satisfied.

In order to generate CTL properties in accordance with the Soft Response and Precedence Patterns some secondary characteristics must be chosen. The most relevant secondary characteristic was selected: the commands that the OBDH can send to PDC ($sec_2 = cmd$). Five clusters of selected $valsec_{2k}$ were then defined and 2-permutations of n was applied considering each of these 5 clusters. Table 5.3 shows the details of the clustering process.

Table 5.3 - Details of the clusters of selected $valsec_{2k}$ for $sec_2 = cmd$. Caption: #Cl = Cluster number; Det = Determinant factor; TtPerm = Total of Permutations; Sci = Scientific; Load = Loading; Hk = Housekeeping; Dmp = Dump

#Cl	**Det**	**Cluster**	**TtPerm**
1	Sci	{*txSci, retAnsw, txSciEBuf, txSciinc*}	12
2	Load	{*stDA, reDA, ldDat, exPrg, ldDatInvCSC, ldDatInvcks*}	30
3	Hk	{*txSci, txHk, retAnsw, prepHk, txHkEBuf, txHkinc*}	30
4	Dmp	{*txSci, txDmp, retAnsw, prepDmp, txDmpEBuf, prepDmpInvMem-Select, prepDmpInvAddrInterv, txDmpInvlen, prepDmpInvcks*}	72
5	Other	{*chPHkTm, chPSmpTmInvdat, chPIniPtrInvtplncks*}	6

The total number of permutations produced by 2-permutations of n values of characteristics for each $cluster_p$ is shown in the column *TtPerm*. Note that there is a determinant factor (column *Det*) that drives the creation of each cluster. In the first four clusters, these determinant factors are services supported by SWPDC: Scientific (Sci), Loading new programs (Load), Housekeeping (Hk), and Dump (Dmp). The fifth cluster ($cluster_5$) was generated due to other general features. Therefore, in the first cluster ($cluster_1$) the selected $valsec_{2k}$ are commands/situations related to acquisition, formatting, and transmission of Scientific Data: TRANSMIT SCIENTIFIC DATA ($txSci$), RETRANSMIT THE LAST DATA RESPONSE ($retAnsw$), TRANSMIT SCIENTIFIC DATA WHEN THE SCIENTIFIC DATA BUFFER IS

EMPTY OR IT DOES NOT HAVE ENOUGH DATA TO FORM A RESPONSE ($txSciEBuf$), and INCOMPLETE RECEPTION OF TRANSMIT SCIENTIFIC DATA COMMAND ($txSciinc$).

Based on each permutation $t = \{t1, t2\}$ a pair of CTL properties was generated: $propertySoftResp_t/propertyPrec_t$. For instance, for $cluster_1$, 2-permutations of n produced the following 12 permutations:

$$\begin{gathered}\{txSci, retAnsw\}, \{txSci, txSciEBuf\}, \{txSci, txSciinc\},\\ \{retAnsw, txSci\}, \{retAnsw, txSciEBuf\}, \{retAnsw, txSciinc\},\\ \{txSciEBuf, txSci\}, \{txSciEBuf, retAnsw\}, \{txSciEBuf, txSciinc\},\\ \{txSciinc, txSci\}, \{txSciinc, retAnsw\}, \{txSciinc, txSciEBuf\}.\end{gathered}$$

Thus, the first permutation $t = \{txSci, retAnsw\}$ generated the following pair of CTL properties:

$$\begin{aligned}&\forall\Box((opmode = nominal \wedge cmd = txSci) \rightarrow\\ &\quad \exists\Diamond(opmode = nominal \wedge cmd = retAnsw)),\\ &\neg\exists[\neg(opmode = nominal \wedge cmd = txSci) \cup ((opmode = nominal \wedge\\ &\quad cmd = retAnsw) \wedge \neg(opmode = nominal \wedge cmd = txSci))].\end{aligned}$$

On the other hand, the twelfth permutation $t = \{txSciinc, txSciEBuf\}$ generated:

$$\begin{aligned}&\forall\Box((opmode = nominal \wedge cmd = txSciinc) \rightarrow\\ &\quad \exists\Diamond(opmode = nominal \wedge cmd = txSciEBuf)),\\ &\neg\exists[\neg(opmode = nominal \wedge cmd = txSciinc) \cup ((opmode = nominal \wedge\\ &\quad cmd = txSciEBuf) \wedge \neg(opmode = nominal \wedge cmd = txSciinc))].\end{aligned}$$

This process was then repeated based on all other permutations not only in $cluster_1$ but also in the remaining clusters. In total, 146 CTL properties according to the Soft Response Pattern and Globally Scope, and 146 CTL properties in accordance with the Precedence Pattern and Globally Scope were formalized. Due to the fact that some pairs of $valsec_{2k}$ are present in various clusters, some permutations appear in more than one cluster. For example, $txSci$ and $retAnsw$ are in $cluster_1$, $cluster_3$, and $cluster_4$. In these clusters, the same permutations $t = \{txSci, retAnsw\}$ and $t = \{retAnsw, txSci\}$ are derived. Of course, it is necessary to create pairs of $propertySoftResp_t/propertyPrec_t$ only once: in the case of a permutation has already generated properties in a previous cluster, it is not necessary to generate them in a subsequent cluster.

By applying Model Checking, 16 incompleteness defects due to the properties formalized according to the Soft Response Pattern and Globally Scope and Precedence Pattern and Globally Scope were detected, as shown in Tables 5.4 (first 8 incompleteness defects) and 5.5 (last 8 incompleteness defects). Usability and Installability are the impact's attributes of ODC (CHILLAREGE et al., 1992) related to all these incompleteness defects and defects' severity are Low and High.

Table 5.4 - First 8 incompleteness defects due to the Soft Response Pattern and Globally Scope/Precedence Pattern and Globally Scope. Caption: Usab = Usability; Install = Installability; cmd = command

Property1	**Property2**	**Impact**	**Severity**	**Unknown Behavior**
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$txSci \prec txSciinc \Rightarrow$ a cmd to transmit Sci Data is incompletely received ($txSciinc$) by SWPDC, and this cmd is sent before a cmd to transmit Sci Data that is completely received ($txSci$)
$SoftR_t = true$	$Pre_t = true$	Install	Low	$stDA \prec ldDat \Rightarrow$ a cmd to load part of a new executable code into PDC's Data Memory ($ldDat$) is sent before a cmd to stop Sci Data acquisition ($stDA$)
$SoftR_t = true$	$Pre_t = true$	Install	Low	$stDA \prec exPrg \Rightarrow$ a cmd to execute a new executable code which was supposedly uploaded into PDC's Data Memory ($exPrg$) is sent before a cmd to stop Sci Data acquisition ($stDA$)
$SoftR_t = true$	$Pre_t = true$	Install	Low	$stDA \prec ldDatInvCSC \Rightarrow$ a cmd to load part of a new executable code into PDC's Data Memory and which has an invalid Command Sequence Control ($ldDatInvCSC$) is sent before a command to stop Sci Data acquisition ($stDA$)

(Continues)

Table 5.4 - Conclusion

Property1	**Property2**	**Impact**	**Severity**	**Unknown Behavior**
$SoftR_t = true$	$Pre_t = true$	Install	Low	*stDA* $\prec$ *ldDatInvcks* $\Rightarrow$ a cmd to load part of a new executable code into PDC's Data Memory and which has an invalid Checksum (*ldDatInvcks*) is sent before a cmd to stop Sci Data acquisition (*stDA*)
$SoftR_t = true$	$Pre_t = true$	Install	High	*ldDat* $\prec$ *exPrg* $\Rightarrow$ a cmd to execute a new executable code which was supposedly uploaded into PDC's Data Memory (*exPrg*) is sent before a cmd to load part of a new executable code (*ldDat*)
$SoftR_t = true$	$Pre_t = true$	Install	High	*ldDat* $\prec$ *ldDatInvCSC* $\Rightarrow$ a cmd to load part of a new executable code into PDC's Data Memory and which has an invalid Command Sequence Control (*ldDatInvCSC*) is sent before a cmd to load part of a new executable code (*ldDat*)
$SoftR_t = true$	$Pre_t = true$	Install	High	*ldDat* $\prec$ *ldDatinvcks* $\Rightarrow$ a cmd to load part of a new executable code into PDC's Data Memory and which has an invalid Checksum (*ldDatInvcks*) is sent before a cmd to load part of a new executable code (*ldDat*)

In column *Unknown Behavior*, $\prec$ means precedence. For instance, *stDA* $\prec$ *ldDat* means the *stDA* command always precedes the *ldDat* command and then the detailed behavior of SWPDC if the opposite situation occurs, i.e. *ldDat* precedes *stDA* is not known. As pointed out in Section 5.1.1, properties formalized in accordance with the Soft Response Pattern and Globally Scope ($SoftR_t$) are easier to be satisfied if compared to the original Response Pattern and Globally Scope. Thus, all incompleteness defects shown in Tables 5.4 and 5.5 were, in fact, detected due to the satisfaction of properties formalized according to the Precedence Pattern and Globally Scope (Pre_t).

Table 5.5 - Last 8 incompleteness defects due to the Soft Response Pattern and Globally Scope/Precedence Pattern and Globally Scope. Caption: Usab = Usability; Install = Installability; cmd = command

Property1	**Property2**	**Impact**	**Severity**	**Unknown Behavior**
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$txHk \prec txHkinc \Rightarrow$ a cmd to transmit Hk Data is incompletely received ($txHkinc$) by SWPDC, and this cmd is sent before a cmd to transmit Hk Data that is completely received ($txHk$)
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$prepHk \prec txHk \Rightarrow$ a cmd to transmit Hk Data ($txHk$) is sent before a cmd to prepare Hk Data to be transmitted ($prepHk$)
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$prepHk \prec txHkEBuf \Rightarrow$ a cmd to transmit Hk Data is sent but the Hk Data buffer is empty ($txHkEBuf$) before a cmd to prepare Hk Data to be transmitted ($prepHk$)
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$prepHk \prec txHkinc \Rightarrow$ a cmd to transmit Hk Data is incompletely received ($txHkinc$) by SWPDC, and this cmd is sent before a cmd to prepare Hk Data to be transmitted ($prepHk$)
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$txDmp \prec txDmpInvlen \Rightarrow$ a cmd to transmit Dmp Data and which has an invalid length ($txDmpInvlen$) is sent before a cmd to transmit Dmp Data ($txDmp$)
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$prepDmp \prec txDmp \Rightarrow$ a cmd to transmit Dmp Data ($txDmp$) is sent before a cmd to prepare Dmp Data to be transmitted ($prepDmp$)

(Continues)

Table 5.5 - Conclusion

Property1	**Property2**	**Impact**	**Severity**	**Unknown Behavior**
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$prepDmp \prec txDmpEBuf \Rightarrow$ a cmd to transmit Dmp Data is sent but the Dmp Data buffer is empty ($txDmpEBuf$) before a cmd to prepare Dmp Data to be transmitted ($prepDmp$)
$SoftR_t = true$	$Pre_t = true$	Usab	Low	$prepDmp \prec txDmpInvlen \Rightarrow$ a cmd to transmit Dmp Data and which has an invalid length ($txDmpInvlen$) is sent before a cmd to prepare Dmp Data to be transmitted ($prepDmp$)

All detected defects based on the three patterns/pattern scopes which have been classified as being of High severity are related to the service Loading new program (executable code) into PDC's Data Memory on the fly. This service was really poorly documented in the software specifications. In addition, all defects found can also be classified according to the impact's attribute Documentation. However, in some cases impact's attributes Installability and Usability best classified the defects found. In total, 362 CTL properties were formalized and 21 incompleteness defects were detected. Therefore, 5.8% of the formalized properties detected incompleteness defects. Table 5.6 summarizes the application of the SOLIMVA methodology to address incompleteness in software specifications considering the SWPDC case study.

5.3 Final remarks about this chapter

This chapter presented an extension of version 1.0 of the SOLIMVA methodology to address the secondary objective of this PhD thesis. This extension generated version 2.0 of the SOLIMVA methodology. The new and most important activity of version 2.0 of SOLIMVA was applied to the SWPDC case study (SANTIAGO et al., 2007) in order to detect incompleteness defects in software specifications. This section provides additional discussion about some approaches related to the problem of incompleteness which were presented in Chapter 2. The differences and the contribution of the approach proposed by the SOLIMVA methodology are also highlighted.

CIRCE (AMBRIOLA; GERVASI, 2006; AMBRIOLA; GERVASI, 1997) detects incomplete-

Table 5.6 - Summary of the application of the SOLIMVA methodology to address incompleteness in software specifications: SWPDC case study

Feature	**Quantity**
Reachable States in the Model	71
Clusters for $sec_2 = cmd$	5
Properties: Absence Pattern and Globally Scope	70
Properties: Soft Response Pattern and Globally Scope	146
Properties: Precedence Pattern and Globally Scope	146
Properties: Total	362
Impact's Attribute of Defect: Installability	9
Impact's Attribute of Defect: Documentation	3
Impact's Attribute of Defect: Usability	9
Defect's Severity: Low	16
Defect's Severity: Medium	1
Defect's Severity: High	4
Defects: Total	21

ness defects by searching for unsued data (variables that are never modified) and data coming from nowhere in Data Flow Diagrams. Kim and Sheldon (2004) proposed to search for absorbing states/activities in Statecharts/Activity Charts. Park et al. (2000) claimed that their tool allows an analyst to improve completeness of a single NL document but what it really supports is the detection of inconsistency defects between NL requirements and duplication of requirements. Between documents, their tool supports traceability but not detection of incompleteness defects.

The SCR formal method and its respective toolset have been used to detect a few missing cases in requirements specifications (EASTERBROOK; CALLAHAN, 1998; HEITMEYER et al., 1998). The formal language RSML and related tools were also able to identify incompleteness defects in software requirements specifications but a significant number of spurious defects were reported (HEIMDAHL; LEVESON, 1996). Yu et al. (2008) proposed an approach to detect incompleteness based on a condition guard tree but their approach does not seem to be scalable as the case study presented was very simple.

The combination of the UPPAAL Model Checker and the CoFI testing methodology (AMBRÓSIO et al., 2007; PONTES et al., 2009b) is another attempt to address the issue of incompleteness but such a combination resembles techniques already proposed in the literature: Software Reading Techniques such as PBR (BASILI et al., 1996). Although their studies try, Model Checking is not really applied to detect incompleteness defects because there is no effective verification of the model against the properties. Moreover, not only in previous studies (PONTES et al., 2009b; PONTES et al., 2009a) but also in a more recent work (PONTES et al., 2010), the authors did not provide guidelines to formalize the properties.

Regarding the detection of incompleteness defects in software specifications, version 2.0 of the SOLIMVA methodology has four important characteristics:

a) Software specifications and their requirements are used not to develop properties but rather to develop the model of the system. This is explained because the purpose is to identify defects in software specifications;

b) Precise guidelines are provided to formalize properties by combining specification patterns with 2-permutations of n values of characteristics. Moreover, the properties are generated in a manner independent of the requirements in the software specifications;

c) The philosophy of Model Checking is respected because the properties and the model are generated from different sources;

d) Specification patterns are not also used in the traditional way. A different interpretation for pattern/pattern scope is provided aiming at detecting incompleteness defects.

The activities of version 1.0 of the SOLIMVA methodology shown in Figure 3.3, which are also present in version 2.0 of the methodology, are typically performed by professionals of the Verification and Validation team such as a test designer. However, the *Analyze Incompleteness* and *Improve Specifications* activities can be accomplished by several other professionals of other teams. In a Waterfall software development lifecycle model as adopted in QSEE project (Figure 3.2), a process for improving the quality of deliverables (documents) in general can be as described below:

a) The deliverables which are related to a particular formal technical review (SRR, PDR, ...) starts to be developed. For example, for the PDR of QSEE project, the PDC-OBDH Communication Protocol Specification was developed by the customer and suppliers developed Software Requirements Specifications;

b) The secretary of the formal technical review communicates to all participants the beginning of the review by providing general guidelines for the process including dates of submission of documents to the secretary, and dates of the formal technical review meetings;

c) The deliverables are finished and sent to the secretary of the formal technical review. The secretary then delivers all documents to all participants of the review;

d) Each participant of the formal technical review evaluates the deliverables in order to detect problems such as ambiguity, inconsistency, and incompleteness. With respect to incompleteness, if the SOLIMVA methodology was applied to assess software specifications, ideally each professional would perform the *Analyze Incompleteness* activity as described in this chapter by using Model Checking, specification patterns and 2-permutations of n values of characteristics. Thus, the *Analyze Incompleteness* activity would be performed by professionals of various other teams and not only of the Verification and Validation team;

e) A Review Item Discrepancy (RID) is created for each detected problem. RID is a document which informs, among others, the identification of the document, in which local (page, section) of the document the problem was found, who is the author of the RID, what is the status of the RID (accepted, rejected, etc.), the description of the problem. In this context, the *report of detected incompleteness defects* mentioned in Figure 5.4 is indeed a set of RIDs created due to incompleteness in software specifications;

f) All participants of the review send their respective RIDs to the secretary before the formal technical review meetings;

g) During the formal technical review meetings which are held in the presence of all those involved in the process, RIDs are discussed. A RID whose status is considered as accepted will mean that those responsible for the creation of the deliverable must implement, after the formal technical review meetings, the suggestions for improving the document which were proposed during the meetings and updated in the RID. For software specifications, this improvement would be carried out by the *Improve Specifications* activity of the SOLIMVA methodology.

Version 2.0 of the SOLIMVA methodology demonstrated its efficiency by the detection of 21 incompleteness defects in the SWPDC case study. But just as in the case of applying version 1.0 of the methodology (Chapter 4), threats to external validity (BASILI et al., 1996) also exist with respect to version 2.0 of SOLIMVA. The methodology has been applied to only one case study (SWPDC) which is not enough and by a single professional. In addition, the effort to apply Model Checking is relatively high because it demands for qualified professional and time for applying the method. The SOLIMVA methodology proposes a precise manner to formalize properties, but the amount of properties can be quite high if several models were developed and many secondary characteristics and their values were considered to generate properties. The time required to simulate each model in order to eliminate defects in modeling is also usually high in real and complex projects.

In any case, the contribution made by the SOLIMVA methodology is relevant because it uses state of the art techniques to address a problem still very common in developing software systems and, as stated earlier, the approach proved to be efficient.

One of the weaknesses of Model Checking is that there is no guarantee of completeness, i.e. only *stated requirements* are checked (BAIER; KATOEN, 2008). In a sense, in the SOLIMVA methodology, Model Checking helps to fight this weakness of Model Checking itself as it (Model Checking) is a crucial part of SOLIMVA for the detection of incompleteness in software specifications.

Due to the fact that all activities of version 1.0 of the SOLIMVA methodology, with the same features, are also present in version 2.0, logically that all the advantages, limitations of version 1.0 of the methodology also apply to version 2.0 of the SOLIMVA methodology, with respect to the generation of model-based system and acceptance test cases considering NL requirements deliverables.

Next chapter presents the conclusions and future directions for this PhD thesis.

6 CONCLUSIONS

Because of the importance of software in modern society, the academic community and industry has been dedicating efforts so that software products of high quality are developed. Defects in artifacts (source code, documentation, etc.) developed for a certain software product may cause minor annoyances, such as the user does not have access to his/her e-mail account for a period of one day, but may have much more serious consequences like causing damage to the environment or causing loss of human lives (LEVESON; TURNER, 1993). Although some authors argue that it is an exaggeration to talk about software crisis (EMAM; KORU, 2008), some studies have found that the cost due to failures in software is around 50 to 80 billion dollars a year (GALORATH INCORPORATED, 2008).

As mentioned in the Introduction (Chapter 1), two objectives, one primary and one secondary, should have been achieved by the solutions proposed in this PhD thesis. Discussion of the solutions presented by this PhD thesis to achieve these goals are presented below. In this chapter, the terms "SOLIMVA methodology" and "SOLIMVA" refer to version 2.0 of the SOLIMVA methodology as this version has all proposed activities to achieve the two objectives of this PhD thesis.

6.1 Solution to achieve the primary objective

The primary objective was to generate model-based system and acceptance test cases considering NL requirements deliverables. Recall that in system and acceptance testing, the entire software product is considered in order to generate test cases. Some researchers state that UML is currently the de facto standard for modeling (object-oriented) software, and its use is increasing in the development of critical systems such as aerospace applications (ZOUGHBI et al., 2011). However and despite all the problems related to NL, its simplicity still makes it attractive for stakeholders to contribute in the preparation of documents (or parts of documents) of software products such as software requirements specifications. In addition, NL is strongly related to UML use case models since the narrative part of the model is made using NL. Therefore, one way or other, in greater or lesser extent, NL is still widely used to develop software requirements specifications or other artifacts created for documenting requirements (MICH et al., 2004).

This primary goal was achieved by the SOLIMVA methodology (Chapter 3). In order to support the SOLIMVA methodology, a tool, also called SOLIMVA, was designed and implemented, and such a tool makes it possible to automatically translate

NL requirements into Statechart models (abstract models). After generating these models, the GTSC environment (SANTIAGO et al., 2008b) is used to generate Abstract Test Cases which are later translated into Executable Test Cases. The SOLIMVA methodology relies on state of the art techniques such as Model-Based Testing (Statechart-based testing) and combinatorial designs (MATHUR, 2008), as well as incorporated into the SOLIMVA tool are other state of the art techniques such as Part Of Speech Tagging (TOUTANOVA et al., 2003) and Word Sense Disambiguation (NAVIGLI, 2009). In summary, the solution proposed by SOLIMVA to achieve the primary objective of this PhD thesis is an attempt to bridge the gap between the state of the art and the state of the practice. This is an important approach towards a wider use of the theory proposed by the academic community in real projects in the industry, and in institutes of research and development.

The SOLIMVA methodology and the SOLIMVA tool were applied (Chapter 4) to SWPDC, a space application software product whose characteristics are representative of an important class of complex software in the Space Segment. SOLIMVA was compared with a previous manual approach developed by an expert under two aspects: coverage of test objectives and characteristics of Executable Test Cases. About coverage of test objectives, the SOLIMVA methodology not only covered the test objectives associated with the expert's scenarios but also proposed a better strategy with test objectives clearly separated according to the directives of combinatorial designs. Executable Test Cases derived in accordance with the SOLIMVA methodology not only possessed similar characteristics with the expert's Executable Test Cases but also predicted behaviors that did not exist in the expert's strategy. Then, the models automatically created by the SOLIMVA methodology and by the SOLIMVA tool are suitable for generating test cases.

Among other advantages of the SOLIMVA methodology and the SOLIMVA tool over the approach of the specialist are:

a) The user must first perform some tasks such as creating a Dictionary and the identification of scenarios. However, the SOLIMVA tool generates the model (abstract) automatically. This functionality is very important because it implies that the user does not have to worry about translating NL requirements into a formal language (Statecharts) and does not need to have expertise in modeling software;

b) By allowing NL requirements to be read automatically along with the fact of proposing the translation from the Abstract Test Suite into the

Executable Test Suite, the SOLIMVA methodology and the SOLIMVA tool have the potential to minimize problems related to the incomplete/inconsistent creation of models for Model-Based Testing;

c) SOLIMVA suggests a precise, systematic and mathematical-based solution to identify scenarios for system and acceptance test case generation by means of combinatorial designs. The result of this approach is that SOLIMVA suggests a response to a very important issue in a Verification and Validation process that is: how many different ways should the IUT be stimulated considering all test suites? In other words, until when should the system be tested? From a practical standpoint, this information is really relevant because complex systems allow users an enormous amount of possibilities of interaction (scenarios). In determining the maximum amount of scenarios, the methodology provides a clear directive to answer this question.

Summarizing what was mentioned at the end of Chapter 4, the SOLIMVA methodology and the SOLIMVA tool have some benefits over other research such as:

a) The SOLIMVA methodology provides a formal manner, by means of combinatorial designs, to identify scenarios for model-based system and acceptance test case generation;

b) SOLIMVA is supported by a formal method (Statecharts language) but it does not require skills related to formal notations in order to define the application domain from the user;

c) Despite the limitations of the algorithm for generating BSAO tuples (Figure 3.10), the SOLIMVA tool provides a certain freedom to the user to write NL requirements, but its approach is not unrestricted NL because it is necessary to define the application domain by means of a Dictionary. Thus, the SOLIMVA tool tends not to have the problems of limited applicability that are associated with unrestricted NL approaches. In addition, the design of the SOLIMVA tool is such that it is not necessary that the requirements are written strictly complying with predefined standards of writing as with very restricted controlled NL approaches;

d) The SOLIMVA tool has a solution to automate the identification of self transitions in the resulting Statechart model by adapting a Word Sense Disambiguation algorithm;

e) SOLIMVA begins to generate the models and hence the Executable Test Suites directly from NL requirements.

Still on the solution proposed by the SOLIMVA methodology to achieve the primary objective, guidelines to apply SOLIMVA to a second case study, CMD/SATCS (CARDOSO et al., 2008), of the space domain related to the Ground Segment were also presented. The feasibility of applying the methodology to this second complex case study was demonstrated, where generating test cases would only be a repetitive process.

6.2 Solution to achieve the secondary objective

As emphasized in Chapter 1, quality of requirements plays an important role in the successful creation of other artifacts of the software development lifecycle. Defects in software requirements specifications like incompleteness are propagated to other artifacts and, therefore, undermine the development process and the final quality of the software product.

The secondary objective of this work relates to a severe problem in software specifications: incompleteness. This secondary goal was also achieved by the SOLIMVA methodology (Chapter 5). State of the art techniques, such as Model Checking (CLARKE; EMERSON, 2008; QUEILLE; SIFAKIS, 2008; BAIER; KATOEN, 2008) and specification patterns (DWYER et al., 1999), combined with k-permutations of n values of variables were used to address this secondary goal. The SOLIMVA methodology was applied to the SWPDC software product (SANTIAGO et al., 2007) in order to detect incompleteness defects in software specifications.

Two patterns/pattern scopes proposed by Dwyer et al. (1999) (Absence Pattern and Globally Scope, Precedence Pattern and Globally Scope) and a third new pattern/pattern scope (Soft Response Pattern and Globally Scope), being a variation of other pattern/pattern scope defined by Dwyer et al. (1999) (Response Pattern and Globally Scope), were used to detect incompleteness defects. A finite-state model was developed for the main operation mode of PDC, the Nominal Operation Mode, and such a model had 71 reachable states. Using CTL, 70 properties were formalized according to the Absence Pattern and Globally Scope, 146 properties according to the Soft Response Pattern and Globally Scope, and 146 properties in accordance with the Precedence Pattern and Globally Scope.

In total, 362 CTL properties were formalized and 21 incompleteness defects were detected in accordance with the following distribution:

a) Impact's attributes of ODC: Installability = 9, Documentation = 3, Usability = 9;

b) Defect's severity: Low = 16, Medium = 1, High = 4.

The efficiency of the SOLIMVA methodology was thus demonstrated by the incompleteness defects detected. Regarding the detection of incompleteness defects in software specifications, some important features of the SOLIMVA methodology are:

a) Since the aim is to detect defects in software specifications, software specifications and their requirements are used not to develop properties but rather to develop the model of the system;

b) SOLIMVA provides clear guidelines to formalize properties by combining specification patterns with 2-permutations of n values of characteristics. In addition, the properties are generated in a manner independent of the requirements in the software specifications;

c) Regarding specification patterns, they are not also used in the traditional way. A different interpretation for pattern/pattern scope is proposed to identify incompleteness defects.

All positive aspects presented in this and in the previous section show that the SOLIMVA methodology and the SOLIMVA tool can make an important contribution to both Software Verification and Validation process and Software Systems Requirements Engineering.

6.3 Future work

Future developments related to this research are divided into two classes. Regarding the primary objective of this PhD thesis and the solution proposed by the SOLIMVA methodology, some directions to follow are:

a) Automated definition of the application domain. Although it was mentioned that the lack of information limits the applicability of unrestricted NL approaches, it is interesting to try to generate the domain information automatically. In the context of the SOLIMVA methodology, this means generating the Dictionary automatically. As previously proposed in other studies (RAYSON et al., 1999; RAYSON et al., 2000), statistical NLP techniques may be a solution to this problem. But further investigations are required;

b) Implementation of a combinatorial designs algorithm. Rather than using an external tool (*TConfig* (UNIVERSITY OF OTTAWA, 2008)) for generating Mixed-Level Covering Arrays, a combinatorial designs algorithm, such as IPOG (LEI et al., 2007), should be implemented within the SOLIMVA tool. In the end, this will facilitate the definition of scenarios because by using the SOLIMVA tool itself it will be possible to generate the factor combinations;

c) Automated identification of scenarios for system and acceptance testing. In a much more ambitious solution than the proposal of the previous item, the identification of scenarios could be done automatically. The user would only provide the NL artifacts and then the scenarios would be automatically produced and reported, also in NL, to the user;

d) Automated selection of a set of NL requirements which together characterize each scenario. This is also another challenge. Once a scenario has been identified then the SOLIMVA tool would automatically select the set of NL requirements which characterize such a scenario and put them in the correct order. Only the NL deliverables would be provided as input to the tool;

e) Improvement of the algorithm for generating BSAO tuples. As mentioned in Chapter 4, the algorithm for automated detection of BSAO tuples (Figure 3.10) not always produced correct results. Although it was successful in the case study considered, it is important to improve this algorithm in order to apply the SOLIMVA methodology and the SOLIMVA tool to other application domains beyond the space domain. One way towards this improvement is by using typed dependencies (MARNEFFE; MANNING, 2008);

f) Automated translation from Abstract Test Suites into Executable Test Suites. As discussed in Chapter 4, the idea behind such a translation was just to substitute one or more *input/output* pairs of the Abstract Test Suite for the corresponding *test input data/expected result* or *actions* of the Executable Test Suite. If an automated solution is designed and implemented so that it is possible to group *input/output* pairs to be translated into *test input data/expected result* or *actions*, then this problem can be solved. Eventually, data mining methods, such as clustering (WITTEN et al., 2011), may help in this regard;

g) Application to other case studies. It is very important to investigate the feasibility of the SOLIMVA methodology in other case studies not only in the space application domain but also in other domains. For example, the thorough application of SOLIMVA to the CMD/SATCS (CARDOSO et al., 2008) software is a natural choice. Besides, it is relevant to conduct empirical studies with other professionals and analyze the impact of the introduction of SOLIMVA in other settings;

h) Assessment of scalability. Despite the explanations given in Chapter 4, scalability seems to be an issue. The SOLIMVA methodology seems to be appropriate up to complex but medium sized software products. Thus, the methodology needs to be improved in order to deal with very large software systems. One way towards this improvement is to optimize (reduce) the number of generated scenarios by using techniques to merge scenarios or even to discard some of them;

i) Improvement of the usability of the SOLIMVA tool. The SOLIMVA tool was developed according to the Object-Oriented Programming paradigm using the Java programming language (Appendix B has more details on this topic). The priority in developing the SOLIMVA tool was to create and implement the algorithms that are, in fact, its main features. Its Graphical User Interface is relatively simple, as it should be, but it is necessary to improve usability features such as exception handling. In addition, the Statechart model is generated in textual notation. Therefore, it is interesting to show it in graphical form using software such as *Graphviz* (GRAPHVIZ.ORG, 2011);

j) Handling documents in the Portuguese language. As mentioned in Chapter 4, all documents developed within the scope of the QSEE project were made in the Portuguese language. Therefore, the selected documents were translated into the English language due to wide acceptance of English worldwide and also due to INPE's international cooperation. However, it is important that the SOLIMVA methodology and the SOLIMVA tool can handle artifacts (documents) drawn up in Portuguese.

Regarding the secondary objective of this PhD thesis and the solution proposed by the SOLIMVA methodology, efforts should be devoted to the following tasks:

a) Automated generation of finite-state models based on software specifications. Generating a model automatically from software specifications, especially if such specifications were prepared by making extensive use of NL, is very complex. So this is another interesting problem to be investigated considering that the model created is suitable for Model Checking. Regarding the SOLIMVA methodology, this implies that the primary characteristic, the secondary characteristics and all their values should automatically be identified as well as the behavior of the system. From this perspective, it is assumed that only the artifacts (documents) themselves are provided for this new version of the SOLIMVA tool;

b) Automated formalization of properties. In the context of SOLIMVA, this implies to automatically formalize the properties according to the three patterns/pattern scopes shown in Chapter 5. The definition of clusters for generating properties due to Soft Response Pattern and Globally Scope/Precedence Pattern and Globally Scope should also occur automatically as well as the derivation of permutations;

c) Automated application of Model Checking, detection of incompleteness defects, and generation/updating of reports. Since models and properties are automatically generated then it is logical to apply Model Checking also automatically. For the case of the SOLIMVA methodology, it would suffice to incorporate the NuSMV Model Checker into the SOLIMVA tool and make calls to NuSMV from the SOLIMVA tool. The result of running NuSMV, such as counterexamples, should also be handled by the SOLIMVA tool. The incompleteness defects should also be detected automatically, according to the *Analyze Incompleteness* activity and the definitions provided in Chapter 5. It would also be remarkable if reports of detected incompleteness defects were automatically generated/updated in NL for easy comprehension by professionals;

d) Use of other combinations of patterns/pattern scopes proposed by Dwyer et al. (1999). Three combinations of patterns/pattern scopes were adopted by SOLIMVA where one is a new combination. As Dwyer et al. (1999) proposed 8 patterns and 5 pattern scopes, other combinations should be investigated because they can improve the quality of the approach to detect incompleteness defects;

e) Application to other case studies. It is very important to apply the solution proposed by SOLIMVA to detect incompleteness defects in software specifications to other case studies as well as to conduct empirical studies to analyze the impact of this proposal in other settings;

f) Assessment of scalability. The effort to apply Model Checking should not be neglected and therefore it should be assessed the feasibility of applying SOLIMVA to very large software systems;

g) Addressing inconsistency in software specifications. Another defect in software specifications is inconsistency like logical contradiction. It should be investigated the possibility of effective detection of inconsistency defects by applying Model Checking.

6.4 Final remarks about this PhD thesis

This section presents concluding remarks about this PhD thesis. The conceptions of the SOLIMVA methodology and the SOLIMVA tool were based on a fact: in greater or lesser extent, NL is still widely used to develop software requirements documents. Naturally, formal methods are a much more appropriate option to not only prepare requirements documents but also to develop other artifacts of the software development lifecycle. But, as mentioned in Chapter 1, such methods do not yet have wide application for the development of systems and software projects in general.

Woodcock et al. (2009) point out several ways to a greater use in practice of formal methods for systems development which includes also software development. One of the directions mentioned refers to increase the level of automation. They mention that some authors predicted in the past that Formal Verification should be used by developers as easily as using a compiler. Other authors mentioned that tools need to be integral parts of development environments, and that the gap between the tools and the standard production process needs to disappear. This approach means moving Formal Verification technology from innovators to first adopters. They also assert that there is an increasing integration between graphical notations and the mathematical representations required for formal analysis. For example, UML-B is an integration between UML and Event-B which is a formal method for system-level modeling and analysis.

Basically, all these previous observations can be summarized in the following sentence: formal methods may be widely used for systems development since its adoption does not require considerable efforts on the part of professionals in industry and in institutes of research and development. In other words, "mere mortals" should feel that formal methods are relatively easy to use. In this sense, the solution proposed by the SOLIMVA methodology to generate model-based system and acceptance test cases considering NL requirements deliverables attempted to follow this line of reasoning. That is, SOLIMVA enables customers, developers continue to develop the artifacts related to the software product in the manner they feel more comfortable, in that case in Natural Language. Then, the SOLIMVA methodology and the SOLIMVA tool transform this user-friendly language into a formal method (Statecharts language) to support processes of the software development lifecycle (Verification and Validation, Software Testing). This is another way that has the potential to spread the adoption of formal methods in practice because its use actually becomes transparent to the user.

Perhaps there is some exaggeration when the word "automation" is used both by academia and industry in the context of software development. For example, as mentioned in Chapter 1, Software Testing automation is, in practice, a semi-automated process involving significant human intervention particularly when dealing with complex software. For considering a Software Testing process as truly automated, taking into account the main activities of the Software Testing process and assuming that the generation of system and acceptance test cases is based on models as considered in the SOLIMVA methodology, the user would only provide for a given tool a certain set of artifacts (software requirements specification, etc.) developed in a user-friendly notation (UML, NL, or a combination thereof). So, having these documents, the tool would:

a) automatically identify all scenarios;

b) automatically generate all models suitable for testing;

c) automatically generate all Executable Test Cases. This step means that test cases would be automatically translated from a possible abstract level into the executable level, that all Executable Test Cases due to all models would be generated automatically including the expected results, that an analysis would be automatically made to eliminate redundant test cases, and that the order of execution of the Executable Test Suites derived according to all scenarios would be accomplished automatically too;

d) automatically run all Executable Test Cases according to the order proposed in the previous step;

e) automatically evaluate the results of the execution of test cases (automated oracle). In this step, the test cases would be automatically associated with verdicts such as “passed” or “failed”;

f) automatically decide which procedure to take should any verdicts of test cases are “failed”. The tool could then automatically generate partial test reports and communicate such test reports for the development team so that the corrections of the defects found in the source code could be made. If it is possible to continue the execution of test cases even with in the presence of some detected defects, then the tool could continue running or, alternatively, wait for the corrected source code to resume execution. This decision should also be automatic. For each new corrected version of the source code, the tool would automatically select a set of test cases for regression testing;

g) automatically generate all the documentation (test case specifications, test reports, etc.) associated with the Software Testing process, after all the test cases were considered successful. Such documentation should be in language easily understood by the user.

This is one possible interpretation for automation of the entire Software Testing process. Naturally, this level of automation, sophistication, intelligence, and expertise is not yet available in tools that support the Software Testing process in any application domain. But it is certainly interesting that academia and industry can devote more efforts in order to achieve, if not quite but almost entirely, this level of automation. Similar efforts towards an increased automation must be dedicated in the development of formal methods in general.

REFERENCES

ABRIAL, J.-R. Formal methods in industry: achievements, problems, future. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 28., 2006, Shanghai, China. **Proceedings...** New York, NY, USA: ACM, 2006. p. 761–768. 7

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. 36, 41, 182

_____. On the systematic analysis of natural language requirements with CIRCE. **Automated Software Engineering**, v. 13, n. 1, p. 107–167, 2006. 36, 37, 38, 39, 41, 150, 158, 182

AMBRÓSIO, A. M.; MATTIELLO-FRANCISCO, F.; SANTIAGO JÚNIOR, V. A.; SILVA, W. P.; MARTINS, E. Designing fault injection experiments using state-based model to test a space software. In: LATIN-AMERICAN SYMPOSIUM ON DEPENDABLE COMPUTING (LADC), 3., 2007, Morelia, México. **Proceedings...** Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2007. p. 170–178. (INPE-15078-PRE/9987). Available from: <`http://urlib.net/sid.inpe.br/mtc-m17@80/2007/11.30.16.22`>. Access in: Nov. 30, 2011. 30, 44, 158, 183

AMMANN, P. E.; BLACK, P. E.; MAJURSKI, W. Using model checking to generate tests from specifications. In: IEEE INTERNATIONAL CONFERENCE ON FORMAL ENGINEERING METHODS (ICFEM), 2., 1998, Brisbane, Australia. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1998. p. 46–54. 46

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. 3

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. 30, 152

BAIER, C.; KATOEN, J.-P. **Principles of model checking**. Cambridge, MA, USA: The MIT Press, 2008. 975 p. 6, 11, 23, 31, 38, 40, 42, 44, 45, 163, 166, 167, 174, 175, 186, 190

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. 29, 31

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. 8

BASILI, V. R.; GREEN, S.; LAITENBERGER, O.; LANUBILE, F.; SHULL, F.; SØRUMGÅRD, S.; ZELKOWITZ, M. V. The empirical investigation of Perspective-Based Reading. **Empirical Software Engineering Journal**, v. 1, n. 2, p. 133–164, 1996. 4, 45, 143, 183, 186

BEHRMANN, G.; DAVID, A.; LARSEN, K. G. A tutorial on UPPAAL. In: BERNARDO, M.; CORRADINI, F. (Ed.). **Formal methods for the design of real-time systems**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2004. v. 3185, p. 200–236. Lecture Notes in Computer Science (LNCS). 25, 44

BEN-ARI, M. **Principles of the Spin model checker**. London, UK: Springer-Verlag, 2008. 216 p. 24

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. 8

BERTOLINO, A.; GNESI, S. Use case-based testing of product lines. **ACM SIGSOFT Software Engineering Notes**, v. 28, n. 5, p. 355–358, 2003. 30

BHARADWAJ, R.; HEITMEYER, C. L. Model checking complete requirements specifications using abstraction. **Automated Software Engineering**, v. 6, n. 1, p. 37–68, 1999. 42, 43, 44, 46

BINDER, R. V. **Testing object-oriented systems**: models, patterns, and tools. USA: Addison-Wesley Professional, 1999. 1248 p. 15, 30

BOWEN, J. P.; HINCHEY, M. G. Seven more myths of formal methods. **IEEE Software**, v. 12, n. 4, p. 34–41, 1995. 6

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. 6

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. 28

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. 30, 152

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, v. 35, n. 8, p. 667–691, 1986. 24

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. 35

CALLAGHAN, P. **An evaluation of LOLITA and related natural language processing systems**. 1998. 212 p. Thesis (PhD in Computer Science) — University of Durham, Durham, UK, 1998. 37

CARDOSO, P. E.; BARRETO, J. P.; CARDOSO, L. S.; HOFFMANN, L. T. Using design patterns, components and metadata to design the command and monitoring frameworks of the INPE's satellite control system. In: INTERNATIONAL CONFERENCE ON SPACE OPERATIONS (SPACEOPS), 9., 2008, Heidelberg, Germany. **Proceedings...** Reston, VA, USA: American Institute of Aeronautics and Astronautics, 2008. v. AIAA-2008-3591. 11, 93, 144, 147, 190, 193

CHAN, W.; ANDERSON, R. J.; BEAME, P.; BURNS, S.; MODUGNO, F.; NOTKIN, D.; REESE, J. D. Model checking large software specifications. **IEEE Transactions on Software Engineering**, v. 24, n. 7, p. 498–520, 1998. 44

CHILLAREGE, R.; BHANDARI, I. S.; CHAAR, J. K.; HALLIDAY, M. J.; MOEBUS, D. S.; RAY, B. K.; WONG, M.-Y. Orthogonal defect classification - a concept for in-process measurements. **IEEE Transactions on Software Engineering**, v. 18, n. 11, p. 943–956, 1992. 176, 179

CHOW, T. S. Testing software design modeled by finite-state machines. **IEEE Transactions on Software Engineering**, SE-4, n. 3, p. 178–187, 1978. 29

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In: GRUMBERG, O.; VEITH, H. (Ed.). **25 years of model checking**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2008. v. 5000, p. 196–215. Lecture Notes in Computer Science (LNCS). 6, 11, 23, 190

CLARKE, E. M.; LERDA, F. Model checking: software and beyond. **Journal of Universal Computer Science**, v. 13, n. 5, p. 639–649, 2007. 23, 24, 38

CRISTIÁ, M.; MONETTI, P. Implementing and applying the Stocks-Carrington framework for model-based testing. In: BREITMAN, K.; CAVALCANTI, A. (Ed.). **Formal methods and software engineering**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2009. v. 5885, p. 167–185. Lecture Notes in Computer Science (LNCS). 32

CRISTIÁ, M.; SANTIAGO, V.; VIJAYKUMAR, N. L. On comparing and complementing two MBT approaches. In: LATIN-AMERICAN TEST WORKSHOP (LATW), 11., 2010, Punta del Este, Uruguay. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2010. p. 1–6. 17, 32

DIAS NETO, A. C. **Seleção de técnicas de teste baseado em modelos**. 2009. 220 p. Thesis (PhD in Computing and Systems Engineering) — Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, RJ, Brazil, 2009. 2, 18, 34

DIAS NETO, A. C.; SUBRAMANYAN, R.; VIEIRA, M.; TRAVASSOS, G. H. **Characterization of model-based software testing approaches**. Rio de Janeiro, RJ, Brazil, 2007. 114p. Technical report. Available from: <http://www.cos.ufrj.br/uploadfiles/1188491168.pdf>. Access in: Oct. 11, 2011. 2, 3, 18

DICK, J.; FAIVRE, A. Automating the generation and sequencing of test cases from model-based specifications. In: WOODCOCK, J.; LARSEN, P. (Ed.). **FME '93: industrial-strength formal methods**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1993. v. 670, p. 268–284. Lecture Notes in Computer Science (LNCS). 30

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. 11, 23, 24, 40, 44, 163, 166, 171, 190, 194

EASTERBROOK, S. **The difference between verification and validation**. 2010. Available from: <http://www.easterbrook.ca/steve/?p=2030>. Access in: Oct. 24, 2011. 13

EASTERBROOK, S.; CALLAHAN, J. Formal methods for verification and validation of partial specifications: a case study. **Journal of Systems and Software**, v. 40, n. 3, p. 199–210, 1998. 43, 183

EL-FAR, I. K.; WHITTAKER, J. A. Model-based software testing. In: MARCINIAK, J. J. (Ed.). **Encyclopedia of software engineering**. USA: Wiley, 2001. 2, 17, 91

EMAM, K. E.; KORU, A. G. A replicated survey of it software project failures. **IEEE Software**, v. 25, p. 84–90, 2008. 4, 187

ENGELS, A.; FEIJS, L.; MAUW, S. Test generation for intelligent networks using model checking. In: BRINKSMA, E. (Ed.). **Tools and algorithms for the construction and analysis of systems**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1997. v. 1217, p. 384–398. Lecture Notes in Computer Science (LNCS). 46

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. 3

EUROPEAN COOPERATION FOR SPACE STANDARDIZATION (ECSS). **ECSS-E-70-41A**: ECSS space engineering - ground systems and operations - telemetry and telecommand packet utilization. Noordwijk, The Netherlands, 2003. 228 p. 45

_____. **ECSS-S-ST-00C**: ECSS system - description, implementation and general requirements. Noordwijk, The Netherlands, 2008. 34 p. 47, 143, 147

_____. **ECSS-E-ST-10-06C**: ECSS space engineering - technical requirements specification. Noordwijk, The Netherlands, 2009. 31 p. 155

EUROPEAN SPACE AGENCY (ESA). **Ariane-5**: Learning from flight 501 and preparing for 502. Paris, France, 1997. ESA Bulletin Nr. 89. Available from: <`http://www.esa.int/esapub/bulletin/bullet89/dalma89.htm`>. Access in: Oct. 8, 2011. 1

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. 35

FANTECHI, A.; GNESI, S.; LAMI, G.; MACCARI, A. Applications of linguistic techniques for use case analysis. **Requirements Engineering**, v. 8, n. 3, p. 161–170, 2003. 8, 35, 40

FANTECHI, A.; GNESI, S.; RISTORI, G.; CARENINI, M.; VANOCCHI, M.; MORESCHINI, P. Assisting requirement formalization by means of natural language translation. **Formal Methods in System Design**, v. 4, n. 3, p. 243–263, 1994. 41, 42

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. 39, 64, 65

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. 4, 35

FONDAZIONE BRUNO KESSLER / CARNEGIE MELLON UNIVERSITY / UNIVERSITY OF GENOVA / UNIVERSITY OF TRENTO. **NuSMV Home Page**. 2011. Available from: <`http://nusmv.fbk.eu/`>. Access in: June 10, 2011. 25, 174

FRASER, M. D.; KUMAR, K.; VAISHNAVI, V. K. Informal and formal requirements specification languages: bridging the gap. **IEEE Transactions on Software Engineering**, v. 17, n. 5, p. 454–466, 1991. 41, 159

FROHLICH, P.; LINK, J. Automated test case generation from dynamic models. In: BERTINO, E. (Ed.). **ECOOP 2000 - object-oriented programming**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2000. v. 1850, p. 472–491. Lecture Notes in Computer Science (LNCS). 30, 31

FUCHS, N. E.; SCHWERTEL, U.; SCHWITTER, R. Attempto Controlled English - not just another logic specification language. In: FLENER, P. (Ed.). **Logic-based program synthesis and transformation**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1999. v. 1559, p. 1–20. Lecture Notes in Computer Science (LNCS). 38, 41, 158

FUCHS, N. E.; SCHWERTEL, U.; TORGE, S. A natural language front-end to model generation. **Journal of Language and Computation**, v. 1, n. 2, p. 199–214, 2000. 38, 41, 158

GALORATH INCORPORATED. **Software project failure costs billions. Better estimation & planning can help**. 2008. Available from: <http://www.galorath.com/wp/software-project-failure-costs-billions-better-estimation-planning-can-help.php>. Access in: Oct. 19, 2011. 187

GANAI, M.; GUPTA, A. **SAT-Based scalable formal verification solutions**. New York, NY, USA: Springer Science+Business Media, 2007. 326 p. 23

GARGANTINI, A.; HEITMEYER, C. Using model checking to generate tests from requirements specifications. **ACM SIGSOFT Software Engineering Notes**, v. 24, n. 6, p. 146–162, 1999. 42, 46

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. 9, 37, 38, 41, 158

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. 8, 35, 37

GODBOLE, N. S. **Software quality assurance**: principles and practice. Oxford, UK: Alpha Science International, 2006. 419 p. 1

GRAPHVIZ.ORG. **Graphviz - graph visualization software**. 2011. Available from: <`http://www.graphviz.org/`>. Access in: Oct. 21, 2011. 193

HALL, A. Seven myths of formal methods. **IEEE Software**, v. 7, p. 11–19, 1990. 6

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, v. 8, p. 231–274, 1987. 10, 17, 33, 43

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. 17, 19, 32, 90, 99

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. 31

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. 28

HEIMDAHL, M. P. E.; LEVESON, N. G. Completeness and consistency in hierarchical state-based requirements. **IEEE Transactions on Software Engineering**, v. 22, n. 6, p. 363–377, 1996. 43, 183

HEITMEYER, C.; KIRBY JR., J.; LABAW, B.; ARCHER, M.; BHARADWAJ, R. Using abstraction and model checking to detect safety violations in requirements specifications. **IEEE Transactions on Software Engineering**, v. 24, n. 11, p. 927–948, 1998. 42, 183

HIERONS, R. M. Testing from a Z specification. **The Journal of Software Testing, Verification and Reliability**, v. 7, n. 1, p. 19–33, 1997. 17, 29

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LüTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H. Using formal specifications to support testing. **ACM Computing Surveys**, v. 41, n. 2, p. 1–76, 2009. 33

HOARE, C. A. R. **Communicating Sequential Processes**. Englewood Cliffs, NJ, USA: Prentice-Hall, 1985. 238 p. 33

HOLT, A.; KLEIN, E.; GROVER, C. Natural language for hardware verification: semantic interpretation and model checking. In: WORKSHOP ON INFERENCE IN COMPUTATIONAL SEMANTICS (ICOS-1), 1., 1999, Amsterdam, The Netherlands. **Proceedings...** [S.l.], 1999. p. 133–137. 41, 42

HOLZMANN, G. J. **The SPIN model checker**: primer and reference manual. USA: Addison-Wesley Professional, 2003. 608 p. 24, 40, 43, 46

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. 30

HOPCROFT, J. E.; ULLMAN, J. D. **Introduction to automata theory, languages, and computation**. Reading, MA, USA: Addison Wesley, 1979. 418 p. 19, 33

HOWDEN, W. E. Reliability of the path analysis testing strategy. **IEEE Transactions on Software Engineering**, SE-2, n. 3, p. 208–215, 1976. 30

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. 38

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS, CIÊNCIAS ESPACIAIS E ATMOSFÉRICAS (INPE.CEA). **Q00-ETS-v03**: Especificação Técnica do Software. São José dos Campos, SP, Brazil, 2006. 82 p. Internal publication. 159

_____. **Q00-PPDEP-v05**: Protocolo de Comunicação PDC-EPPs. São José dos Campos, SP, Brazil, 2006. 8 p. Internal publication. 159

_____. **Q00-PPDOB-v06**: Protocolo de Comunicação PDC-OBDH. São José dos Campos, SP, Brazil, 2006. 28 p. Internal publication. 159

_____. **Q00-RB-v08**: Requisitos de Base. São José dos Campos, SP, Brazil, 2006. 29 p. Internal publication. 159

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS, DIVISÃO DE DESENVOLVIMENTO DE SISTEMAS DE SOLO (INPE.DSS). **DSS-INT-SATCS-PR-005**: SATCS Design Description. São José dos Campos, SP, Brazil, 2010. 33 p. Internal publication. 145

JIANG, J. J.; CONRATH, D. W. Semantic similarity based on corpus statistics and lexical taxonomy. In: INTERNATIONAL CONFERENCE RESEARCH ON

COMPUTATIONAL LINGUISTICS (ROCLING), 10., 1997, Taipei, Taiwan. **Proceedings...** [S.l.], 1997. p. 19–33. 27

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. 25

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. 39, 41, 159, 183

KO, Y.; PARK, S.; SEO, J.; CHOI, S. Using classification techniques for informal requirements in the requirements analysis-supporting system. **Information and Software Technology**, v. 49, n. 11-12, p. 1128–1140, 2007. 5

KOH, K. Y.; SEONG, P. H. SMV model-based safety analysis of software requirements. **Reliability Engineering and System Safety**, v. 94, n. 2, p. 320–331, 2009. 41, 42

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. 40, 44

______. Automated analysis of natural language properties for UML models. In: BRUEL, J.-M. (Ed.). **Satellite events at the MoDELS 2005 conference**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2006. v. 3844, p. 48–57. Lecture Notes in Computer Science (LNCS). 40, 41

LAMI, G.; TRENTANNI, G. An automatic tool for improving the quality of software requirements. **ERCIM News**, n. 58, p. 18–19, 2004. 35

LAMSWEERDE, A. van. Requirements engineering in the year 00: a research perspective. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 22., 2000, Limerick, Ireland. **Proceedings...** New York, NY, USA: ACM, 2000. p. 5–19. 4

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. 13

LEACOCK, C.; CHODOROW, M. Combining local context and WordNet similarity for word sense identification. In: FELLBAUM, C. (Ed.). **WordNet: an electronic lexical database**. Cambridge, MA, USA: The MIT Press, 1998. chapter 11, p. 265–283. 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. 17, 29, 33

LEI, Y.; KACKER, R.; KUHN, D. R.; OKUN, V.; LAWRENCE, J. IPOG: A general strategy for t-way software testing. In: ANNUAL IEEE INTERNATIONAL CONFERENCE AND WORKSHOPS ON THE ENGINEERING OF COMPUTER-BASED SYSTEMS (ECBS), 14., 2007, Tucson, AZ, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p. 549–556. 54, 192

LEI, Y.; TAI, K.-C. In-Parameter-Order: A test generation strategy for pairwise testing. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH-ASSURANCE SYSTEMS ENGINEERING (HASE), 3., 1998, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1998. p. 254–261. 53

LESK, M. Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In: INTERNATIONAL CONFERENCE ON SYSTEMS DOCUMENTATION (SIGDOC), 5., 1986, Toronto, ON, Canada. **Proceedings...** New York, NY, USA: ACM, 1986. p. 24–26. 27

LEVESON, N. G.; TURNER, C. S. An investigation of the Therac-25 accidents. **Computer**, v. 26, n. 7, p. 18–41, 1993. 1, 187

LIANG, J.; PALMER, J. D. A pattern matching and clustering based approach for supporting requirements transformation. In: IEEE INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING (ICRE), 1., 1994, Colorado Springs, CO, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1994. p. 180–183. 41, 159

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. 3

LU, C.-W.; CHANG, C.-H.; CHU, W. C.; CHENG, Y.-W.; CHANG, H.-C. A requirement tool to support model-based Requirement Engineering. In: ANNUAL IEEE INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), 32., 2008, Turku, Finland. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p. 712–717. 39, 159

MARCUS, M. P.; MARCINKIEWICZ, M. A.; SANTORINI, B. Building a large annotated corpus of English: The Penn Treebank. **Computational Linguistics**, v. 19, n. 2, p. 313–330, 1993. 65

MARNEFFE, M.-C.; MANNING, C. D. The Stanford typed dependencies representation. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LINGUISTICS (COLING) - WORKSHOP ON CROSS-FRAMEWORK AND CROSS-DOMAIN PARSER EVALUATION, 22., 2008, Manchester, United Kingdom. **Proceedings...** Stroudsburg, PA, USA: Association for Computational Linguistics, 2008. p. 1–8. 72, 192

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. 19, 63

MATHUR, A. P. **Foundations of software testing**. Delhi, India: Dorling Kindersley (India), Pearson Education in South Asia, 2008. 689 p. 3, 10, 15, 17, 18, 21, 22, 53, 91, 188

McMILLAN, K. L. **Symbolic model checking**. New York, NY, USA: Springer-Verlag, 1993. 216 p. 42, 43, 44, 46

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. 36, 158

MICH, L.; FRANCH, M.; INVERARDI, P. L. N. Market research for requirements analysis using linguistic tools. **Requirements Engineering**, v. 9, n. 1, p. 40–56, 2004. 7, 8, 187

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). 37

MILLER, G. A. Nouns in WordNet. In: FELLBAUM, C. (Ed.). **WordNet: an electronic lexical database**. Cambridge, MA, USA: The MIT Press, 1998. chapter 1, p. 23–46. 26

MILLER, G. A.; LEACOCK, C.; TENGI, R.; BUNKER, R. T. A semantic concordance. In: WORKSHOP ON HUMAN LANGUAGE TECHNOLOGY (HLT), 1993, Princeton, NJ, USA. **Proceedings...** Stroudsburg, PA, USA: Association for Computational Linguistics, 1993. p. 303–308. 81

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. 36, 37

MYERS, G. J. **The art of software testing**. 2. ed. Hoboken, NJ, USA: John Wiley & Sons, 2004. 234 p. 14

NASA. **NASA Software Assurance**: Software Assurance Definitions. 2009. Available from: <`http://www.hq.nasa.gov/office/codeq/software/umbrella_defs.htm`>. Access in: Oct. 8, 2011. 1

NASA AMES RESEARCH CENTER. **Java Pathfinder Home Page**. 2011. Available from: <`http://babelfish.arc.nasa.gov/trac/jpf`>. Access in: Sept. 27, 2011. 25

NASA JET PROPULSION LABORATORY. **NASA Jet Propulsion Laboratory**: Mars Climate Orbiter Mission. 2011. Available from: <`http://www.jpl.nasa.gov/missions/missiondetails.cfm?mission=MCO`>. Access in: Oct. 8, 2011. 1

NATT OCH DAG, J.; REGNELL, B.; CARLSHAMRE, P.; ANDERSSON, M.; KARLSSON, J. A feasibility study of automated natural language requirements analysis in market-driven development. **Requirements Engineering**, v. 7, n. 1, p. 20–33, 2002. 40

NAVIGLI, R. Word sense disambiguation: A survey. **ACM Computing Surveys**, v. 41, n. 2, p. 1–69, 2009. 10, 25, 26, 91, 188

NELKEN, R.; FRANCEZ, N. Automatic translation of natural language system specifications into temporal logic. In: ALUR, R.; HENZINGER, T. (Ed.).

Computer aided verification. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1996. v. 1102, p. 360–371. Lecture Notes in Computer Science (LNCS). 41, 42

NETBEANS.ORG. **Welcome to NetBeans**. 2011. Available from: <`http://netbeans.org/`>. Access in: Dec. 28, 2011. 231

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. 5, 6, 8

OFFUTT, J.; ABDURAZIK, A. Generating tests from UML specifications. In: FRANCE, R.; RUMPE, B. (Ed.). **UML'99 - the unified modeling language**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 1999. v. 1723, p. 416–429. Lecture Notes in Computer Science (LNCS). 30

ORACLE. **Javadoc tool home page**. 2011. Available from: <`http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`>. Access in: Dec. 28, 2011. 233

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. 15, 28, 29, 31

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. 29

PARK, S.; KIM, H.; KO, Y.; SEO, J. Implementation of an efficient requirements-analysis supporting system using similarity measure techniques. **Information and Software Technology**, v. 42, n. 6, p. 429–438, 2000. 37, 183

PEDERSEN, T.; PATWARDHAN, S.; MICHELIZZI, J. WordNet::Similarity: measuring the relatedness of concepts. In: ANNUAL CONFERENCE OF THE NORTH AMERICAN CHAPTER OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS (NAACL), 5., 2004, Boston, MA, USA. **Proceedings...** Stroudsburg, PA, USA: Association for Computational Linguistics, 2004. p. 38–41. 81

PETRENKO, A.; YEVTUSHENKO, N. Testing from partial deterministic FSM specifications. **IEEE Transactions on Computers**, v. 54, n. 9, p. 1154–1165, 2005. 18, 29, 90

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. 29, 31

PONTES, R. P.; ESSADO, M.; VÉRAS, P. C.; AMBRÓSIO, A. M.; VILLANI, E. A comparative analysis of two verification techniques for DEDS: model checking versus model-based testing. In: INTERNATIONAL IFAC WORKSHOP ON DISCRETE-EVENT SYSTEM DESIGN (DESDes), 4., 2009, Valencia, Spain. **Proceedings...** Oxford, UK: Elsevier Ltd., 2009. 45, 184

_____. Model-based refinement of requirement specification: a comparison of two V&V approaches. In: INTERNATIONAL CONGRESS OF MECHANICAL ENGINEERING (COBEM), 20., 2009, Gramado, RS, Brazil. **Proceedings...** Rio de Janeiro, RJ, Brazil: ABCM, 2009. 44, 45, 183, 184

PONTES, R. P.; VILLANI, E.; AMBRÓSIO, A. M. Modelagem e verificação formal de software embarcado espacial segundo a norma PUS. In: CONGRESSO BRASILEIRO DE AUTOMÁTICA (CBA), 18., 2010, Bonito, MS, Brazil. **Proceedings...** Campinas, SP, Brazil: Sociedade Brasileira de Automática, 2010. p. 4779–4786. 45, 184

PORTER, A. A.; VOTTA JR., L. G.; BASILI, V. R. Comparing detection methods for software requirements inspections: a replicated experiment. **IEEE Transactions on Software Engineering**, v. 21, n. 6, p. 563–575, 1995. 4

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

QUEILLE, J.-P.; SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In: GRUMBERG, O.; VEITH, H. (Ed.). **25 years of model checking**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2008. v. 5000, p. 216–230. Lecture Notes in Computer Science (LNCS). 6, 11, 23, 190

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Transactions on Software Engineering**, SE-11, n. 4, p. 367–375, 1985. 20

RAYSON, P.; GARSIDE, R.; SAWYER, P. **Language engineering for the recovery of requirements from legacy documents**. Lancaster, UK: Lancaster University, 1999. 17 p. 192

_____. Assisting requirements engineering with semantic document analysis. In: INTERNATIONAL CONFERENCE ON COMPUTER-ASSISTED INFORMATION RETRIEVAL (RIAO: RECHERCHE D'INFORMATION ASSISTÉE PAR ORDINATEUR), 6., 2000, Paris, France. **Proceedings...** [S.l.], 2000. p. 1363–1371. 192

RUSSELL, S. J.; NORVIG, P. **Artificial intelligence**: a modern approach. Englewood Cliffs, NJ, USA: Prentice-Hall, 1995. 932 p. 25

SANTIAGO JÚNIOR, V. A.; CRISTIÁ, M.; VIJAYKUMAR, N. L. **Model-based test case generation using Statecharts and Z**: a comparison and a combined approach. São José dos Campos: INPE, 2010. 72 p. (INPE-16677-RPQ/850). Available from: <http://urlib.net/sid.inpe.br/mtc-m19@80/2010/02.26.14.05>. Access in: Apr. 14, 2010. 63, 108, 112, 117, 125, 126, 136

SANTIAGO JÚNIOR, V. A.; VIJAYKUMAR, N. L. Generating model-based test cases from natural language requirements for space application software. **Software Quality Journal**, v. 20, n. 1, p. 77–143, 2012. DOI: 10.1007/s11219-011-9155-6. 2, 47, 93

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. 31, 63

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. 10, 11, 47, 48, 49, 50, 91, 93, 147, 172, 182, 190

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. 2, 15, 152

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. 2, 10, 17, 31, 61, 188

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. 30

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. 17, 29, 62, 152

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. 29

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. 8, 29

SINHA, R.; MIHALCEA, R. Unsupervised graph-based word sense disambiguation using measures of word semantic similarity. In: IEEE INTERNATIONAL CONFERENCE ON SEMANTIC COMPUTING (ICSC), 1., 2007, Irvine, CA, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p. 363–369. 27, 28, 79, 81, 82

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. 39, 65

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. 34, 158

SOLEMON, B.; SAHIBUDDIN, S.; GHANI, A. A. A. Requirements engineering problems and practices in software companies: An industrial survey. In: SLEZAK, D.; KIM, T.-h.; KIUMI, A.; JIANG, T.; VERNER, J.; ABRAHAO, S. (Ed.). **Advances in software engineering**. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2009. v. 59, p. 70–77. Communications in Computer and Information Science. 5

SOUZA, É. F. **Geração de casos de teste para sistemas da área espacial usando critérios de teste para máquinas de estados finitos**. 2010. 133 p. (INPE-16682-TDI/1627). Dissertation (Master in Applied Computing) — Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil, 2010. 62, 152

SOUZA, S. R. S. **Validação de especificações de sistemas reativos: definição e análise de critérios de teste**. 2000. 264 p. Thesis (PhD in Applied Physics) — Universidade de São Paulo (USP), São Carlos, SP, Brazil, 2000. 19, 21, 62, 153

SPIVEY, J. M. **The Z notation**: A reference manual. 2. ed. Oxford, England: J. M. Spivey, 1998. 168 p. 33

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE). **ANSI/IEEE Std 830-1984**: IEEE guide to software requirements specifications. New York, NY, USA, 1984. 24 p. 37, 163

_____. **IEEE Std 610.12-1990**: IEEE standard glossary of software engineering terminology. New York, NY, USA, 1990. 83 p. 1, 3, 4, 13, 22

_____. **IEEE Std 829-1998**: IEEE standard for software test documentation. New York, NY, USA, 1998. 52 p. 14

THE OBJECT MANAGEMENT GROUP (OMG). **OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2**. Needham, MA, USA, 2007. 722 p. 3, 6, 17

TOUTANOVA, K.; KLEIN, D.; MANNING, C. D.; SINGER, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In: CONFERENCE OF THE NORTH AMERICAN CHAPTER OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS ON HUMAN LANGUAGE TECHNOLOGY, 2003, Edmonton, Canada. **Proceedings...** [S.l.], 2003. p. 173–180. 10, 65, 91, 156, 188

TRAVASSOS, G. H.; SHULL, F.; FREDERICKS, M.; BASILI, V. R. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA '99), 14., 1999, Denver, CO, USA. **Proceedings...** New York, NY, USA: ACM, 1999. p. 47–56. 4

UNIVERSITY OF OTTAWA. **Alan Williams' Page**. 2008. Available from: <`http://www.site.uottawa.ca/~awilliam/`>. Access in: Oct. 15, 2009. 53, 149, 156, 192, 232

UNIVERSITY OF SUSSEX. **David Hope's page**. 2010. Available from: <`http://www.cogs.susx.ac.uk/users/drh21/`>. Access in: July 22, 2010. 80, 156

UNIVERSITY OF ZURICH. **Attempto Project**. 2009. Available from: <`http://attempto.ifi.uzh.ch/site/tools/`>. Access in: Feb. 05, 2010. 38

UTTING, M.; LEGEARD, B. **Practical Model-Based Testing**: A tools approach. Waltham, MA, USA: Morgan Kaufmann Publishers, 2007. 456 p. 17, 18, 91

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. 31, 63

WEYUKER, E. J. On testing non-testable programs. **The Computer Journal**, v. 25, n. 4, p. 465–470, 1982. 15

WILSON, W. M.; ROSENBERG, L. H.; HYATT, L. E. Automated analysis of requirement specifications. In: INTERNATIONAL CONFERENCE ON

SOFTWARE ENGINEERING (ICSE), 19., 1997, Boston, MA, USA. **Proceedings...** New York, NY, USA: ACM, 1997. p. 161–171. 8, 35

WITTEN, I. H.; FRANK, E.; HALL, M. A. **Data Mining**: Practical machine learning tools and techniques. 3. ed. Waltham, MA, USA: Morgan Kaufmann Publishers, 2011. 664 p. 193

WOODCOCK, J.; LARSEN, P. G.; BICARREGUI, J.; FITZGERALD, J. Formal methods: Practice and experience. **ACM Computing Surveys**, v. 41, n. 4, p. 19:1–19:36, 2009. 6, 7, 195

YU, L.; SU, S.; LUO, S.; SU, Y. Completeness and consistency analysis on requirements of distributed event-driven systems. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON THEORETICAL ASPECTS OF SOFTWARE ENGINEERING (TASE), 2., 2008, Nanjing, China. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p. 241–244. 46, 183

ZOUGHBI, G.; BRIAND, L.; LABICHE, Y. Modeling safety and airworthiness (RTCA DO-178B) information: conceptual model and UML profile. **Software and Systems Modeling**, v. 10, n. 3, p. 337–367, 2011. 7, 187

APPENDIX A - SWPDC SOFTWARE PRODUCT: NATURAL LANGUAGE REQUIREMENTS

This appendix shows the 97 NL requirements collected from the four deliverables (documents) prepared for the SWPDC software product, and which were used by the SOLIMVA methodology and by the SOLIMVA tool to generate model-based system and acceptance test cases: Requirements Baseline, Software Requirements Specification, PDC-OBDH Communication Protocol Specification, and PDC-EPPs Communication Protocol Specification. As mentioned at the end of Chapter 4, all documents developed within the scope of the QSEE project were made in the Portuguese language. Therefore, the selected documents were translated from the Portuguese language into the English language due to wide acceptance of English worldwide and also due to INPE's international cooperation.

It was also said at the end of Chapter 4 that some of these requirements have been somehow created in NL. In this context, some requirements considered as already created in NL may even have minor modifications of the original requirement in Portuguese. But in these cases, the analysis was made taking into account that there was sufficient information in the documents, in NL, but distributed so that it was coherent to consider that such requirements have already been created in Natural Language. Moreover, some other requirements were extracted from UML diagrams (sequence, activity) and converted into NL requirements. However, "new" NL requirements had to be created by the test designer in order to generate coherent test cases. These requirements were not present in any of the documents consulted to apply the SOLIMVA methodology. So, for instance, requirements were "collected" from the Software Requirements Specification meaning that some of these NL requirements really existed and were used. However, some other requirements did not exist and they were therefore created.

Table A.1 shows the profile of the collected requirements used in the SOLIMVA methodology where the column *Source* refers to how requirements in Natural Language has been obtained: *NL* means that the requirements were already properly created in NL, *UML* means that the requirements were derived from UML diagrams and converted into NL, and *NEW* means that requirements were created because they did not exist.

In the next sections the requirements used when applying the SOLIMVA methodology will be presented. Identification of requirements is of the form *DC* where *C* is a counter starting at 001, and *D* identifies the deliverable (document) where the

Table A.1 - Profile of the collected requirements: SWPDC case study

Source	**Quantity**	Percentage
NL	34	35.05
UML	25	25.77
NEW	38	39.18
Total	97	100.00

requirement was collected. Therefore, D can assume the following values:

- RB = Requirements Baseline;
- SRS = Software Requirements Specification;
- POCP = PDC-OBDH Communication Protocol Specification;
- PECP = PDC-EPPs Communication Protocol Specification.

A.1 Requirements collected from the Requirements Baseline

The requirements collected from the Requirements Baseline are listed as follows:

- RB001 - The OBDH shall send VER-OP-MODE to PDC.
- RB002 - The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH sends distinct commands to perform such actions.
- RB003 - The OBDH shall send CH-OP-MODE-Nominal to PDC.
- RB004 - The OBDH shall send CH-OP-MODE-Safety to PDC. After that, the PDC shall be in the Safety Operation Mode.
- RB005 - After switching both EPPHxs off via PDC, the OBDH shall switch the PDC off via the Power Conditioning Unit.
- RB006 - The OBDH should wait 10 seconds before asking for a Test Data frame.
- RB007 - The OBDH shall send AC-HW-RESET to PDC.
- RB008 - The OBDH shall send STOP-DT-ACQ to PDC.

- RB009 - The OBDH shall send LD-PRG-MIN-Ok to PDC.
- RB010 - The OBDH shall send several LD-PRG-INRNG-Ok to PDC.
- RB011 - The OBDH shall send EXEC-CKS-INV to PDC.
- RB012 - The OBDH shall send RSTART-DATA-ACQ to PDC.

A.2 Requirements collected from the Software Requirements Specification

The requirements collected from the Software Requirements Specification are listed as follows:

- SRS001 - The PDC shall be powered on by the Power Conditioning Unit.
- SRS002 - The PDC shall be in the Initiation Operation Mode after being powered on. The 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.
- SRS003 - If PDC does not present any irrecoverable problem, after the initiation process, the PDC shall automatically enter into the Safety Operation Mode.
- SRS004 - The OBDH should wait 600 seconds before asking for a Housekeeping Data frame.
- SRS005 - Housekeeping data transmission shall start with PREP-HK. After that, the OBDH can send several TX-DATA-HK to PDC. The transmission shall be ended with TX-DATA-SCI-End.
- SRS006 - The SWPDC shall obtain and handle scientific data from each EPP Hx. The SWPDC shall also accept scientific data transmission requests from OBDH.
- SRS007 - Test data transmission shall start with prep-tst. After that, the OBDH shall send two tx-data-tst to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS008 - The OBDH should wait MIN seconds before asking for a Housekeeping Data frame.

- SRS009 - The SWPDC shall distinguish between Power On/Reset processes.
- SRS010 - Requirements SRS002 and SRS003 also apply to the Reset process.
- SRS011 - The PDC shall store information in memory that defines the current configuration of the computing subsystem.
- SRS012 - The information that indicates which is the process currently occurring (Power On or Reset) is part of the configuration information, and this information shall be read and updated to a preset value in case of Power On. In case of Reset, this information shall only be read.
- SRS013 - The SWPDC shall always maintain temporarily stored the last data response sent to the OBDH because the OBDH can demand the retransmission of this last data response.
- SRS014 - The OBDH should wait MINDEF seconds before asking for a Housekeeping Data frame.
- SRS015 - The SWPDC shall be capable to load and execute new programs in the Nominal Operation Mode. First, the OBDH shall send STOP-DT-ACQ so that the SWPDC can interrupt the acquisition of scientific data. After this command, the PDC is expected to receive LD-PRG-MIN-Ok and, after that, the OBDH shall send several LD-PRG-INRNG-Ok. Then, the OBDH shall submit EXEC-CKS-OK to PDC so that the loaded program is executed. At the end of the process of loading and executing a new program, the OBDH shall send rstart-DATA-ACQ to PDC so that PDC can restart acquiring scientific data from EPP Hxs.
- SRS016 - Housekeeping data transmission shall start with prep-hk. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-hk to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS017 - Memory Dump data transmission shall start with prep-dmp-prg-inrng-inrng. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS018 - Memory Dump data transmission shall start with prep-dmp-prg-min-min. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS019 - Memory Dump data transmission shall start with prep-dmp-prg-max-max. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS020 - Memory Dump data transmission shall start with prep-dmp-prg-lessmin-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS021 - Memory Dump data transmission shall start with prep-dmp-prg-greatmax-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS022 - Memory Dump data transmission shall start with prep-dmp-dtp0-inrng-min. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS023 - Memory Dump data transmission shall start with prep-dmp-dtp0-min-inrng. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS024 - Memory Dump data transmission shall start with prep-dmp-dtp0-max-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS025 - Memory Dump data transmission shall start with prep-dmp-dtp0-lessmin-max. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS026 - Memory Dump data transmission shall start with prep-dmp-dtp0-greatmax-inrng. In this case, the SWPDC shall not stop scientific

data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS027 - Memory Dump data transmission shall start with prep-dmp-dtp0-min-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS028 - Memory Dump data transmission shall start with prep-dmp-dtp1-inrng-max. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS029 - Memory Dump data transmission shall start with prep-dmp-dtp1-min-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS030 - Memory Dump data transmission shall start with prep-dmp-dtp1-max-inrng. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS031 - Memory Dump data transmission shall start with prep-dmp-dtp1-lessmin-min. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS032 - Memory Dump data transmission shall start with prep-dmp-dtp1-greatmax-min. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS033 - Memory Dump data transmission shall start with prep-dmp-dtp1-min-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS034 - Memory Dump data transmission shall start with prep-dmp-dtp2-inrng-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS035 - Memory Dump data transmission shall start with prep-dmp-dtp2-min-max. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS036 - Memory Dump data transmission shall start with prep-dmp-dtp2-max-min. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS037 - Memory Dump data transmission shall start with prep-dmp-dtp2-lessmin-inrng. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS038 - Memory Dump data transmission shall start with prep-dmp-dtp2-greatmax-max. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS039 - Memory Dump data transmission shall start with prep-dmp-dtp2-min-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS040 - Memory Dump data transmission shall start with prep-dmp-dtp3-inrng-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS041 - Memory Dump data transmission shall start with prep-dmp-dtp3-min-inrng. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS042 - Memory Dump data transmission shall start with prep-dmp-dtp3-max-min. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS043 - Memory Dump data transmission shall start with prep-dmp-dtp3-lessmin-max. In this case, the SWPDC shall not stop scientific data

acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS044 - Memory Dump data transmission shall start with prep-dmp-dtp3-greatmax-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS045 - Memory Dump data transmission shall start with prep-dmp-dtp4-inrng-inrng. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS046 - Memory Dump data transmission shall start with prep-dmp-dtp4-min-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS047 - Memory Dump data transmission shall start with prep-dmp-dtp4-max-min. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS048 - Memory Dump data transmission shall start with prep-dmp-dtp4-lessmin-max. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS049 - Memory Dump data transmission shall start with prep-dmp-dtp4-greatmax-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS050 - Memory Dump data transmission shall start with prep-dmp-dtp5-inrng-inrng. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS051 - Memory Dump data transmission shall start with prep-dmp-dtp5-min-min. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS052 - Memory Dump data transmission shall start with prep-dmp-dtp5-max-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS053 - Memory Dump data transmission shall start with prep-dmp-dtp5-lessmin-max. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS054 - Memory Dump data transmission shall start with prep-dmp-dtp5-greatmax-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS055 - Memory Dump data transmission shall start with prep-dmp-dtp6-inrng-inrng. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS056 - Memory Dump data transmission shall start with prep-dmp-dtp6-min-min. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS057 - Memory Dump data transmission shall start with prep-dmp-dtp6-max-max. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS058 - Memory Dump data transmission shall start with prep-dmp-dtp6-lessmin-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS059 - Memory Dump data transmission shall start with prep-dmp-dtp6-greatmax-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

- SRS060 - Memory Dump data transmission shall start with prep-dmp-dtp7-inrng-inrng. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send several tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS061 - Memory Dump data transmission shall start with prep-dmp-dtp7-min-min. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS062 - Memory Dump data transmission shall start with prep-dmp-dtp7-max-max. In this case, the SWPDC shall stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS063 - Memory Dump data transmission shall start with prep-dmp-dtp7-lessmin-lessmin. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.
- SRS064 - Memory Dump data transmission shall start with prep-dmp-dtp7-greatmax-greatmax. In this case, the SWPDC shall not stop scientific data acquisition from EPP Hxs. After that, the OBDH can send one tx-data-dmp to PDC. The transmission shall be ended with tx-data-sci-end.

A.3 Requirements collected from the PDC-OBDH Communication Protocol Specification

The requirements collected from the PDC-OBDH Communication Protocol Specification are listed as follows:

- POCP001 - The PDC can only respond to requests (commands) from OBDH after the PDC has been energized for at least 1 minute. If OBDH sends commands within less than 1 minute, the OBDH shall not receive any response from PDC.
- POCP002 - The OBDH should wait 10 seconds before asking for a Scientific Data frame.
- POCP003 - The OBDH shall send CH-SW-PHK-MIN to PDC.

- POCP004 - The OBDH shall send CH-SW-PIP-INRNG to PDC.
- POCP005 - The OBDH shall send CH-SW-PST-MAX to PDC.
- POCP006 - The OBDH shall send CH-SW-PIP-MIN to PDC.
- POCP007 - The OBDH shall send CH-SW-PST-GREATMAX to PDC.
- POCP008 - The OBDH shall send CH-SW-PIP-MAX to PDC.
- POCP009 - The OBDH shall send CH-SW-PST-LESSMIN to PDC.
- POCP010 - The OBDH shall send CH-SW-PIP-LESSMIN to PDC.
- POCP011 - The OBDH shall send CH-SW-PST-MIN to PDC.
- POCP012 - The OBDH shall send CH-SW-PIP-GREATMAX to PDC.
- POCP013 - The OBDH shall send CH-SW-PST-MINDEF to PDC.
- POCP014 - The OBDH shall send CH-SW-PIP-INRNG to PDC.
- POCP015 - The OBDH shall send CH-SW-PST-DEF to PDC.
- POCP016 - The OBDH shall send CH-SW-PIP-LESSMIN to PDC.
- POCP017 - The OBDH shall send CH-SW-PST-DEFMAX to PDC.
- POCP018 - The OBDH shall send CH-SW-PHK-DEF to PDC.
- POCP019 - The PDC shall check all fields of the commands received. In case of inconsistency in the values received in any of the fields of an OBDH's command, the PDC shall abort the communication, the command shall be discarded, an event report shall be generated, and the PDC shall wait for a new OBDH's command. The PDC shall not send back any negative acknowledgment message to the OBDH.
- POCP020 - The PDC may not receive a command sent in its entirety. After identifying the beginning of a command frame, the PDC shall wait two times MAX-TRANSM-DELAY for the rest of the command. If this stipulated time expires, a timeout shall occur, the PDC shall abort the communication, the command shall be discarded, an event report shall be generated, and the PDC shall wait for a new OBDH's command.

A.4 Requirement collected from the PDC-EPPs Communication Protocol Specification

The single requirement collected from the PDC-EPPs Communication Protocol Specification is listed below:

- PECP001 - Each EPP Hx can only respond to requests (commands) from PDC after each EPP Hx has been energized for at least 30 seconds. If PDC sends commands within less than 30 seconds to a certain EPP Hx, the PDC shall not receive any response from this EPP Hx.

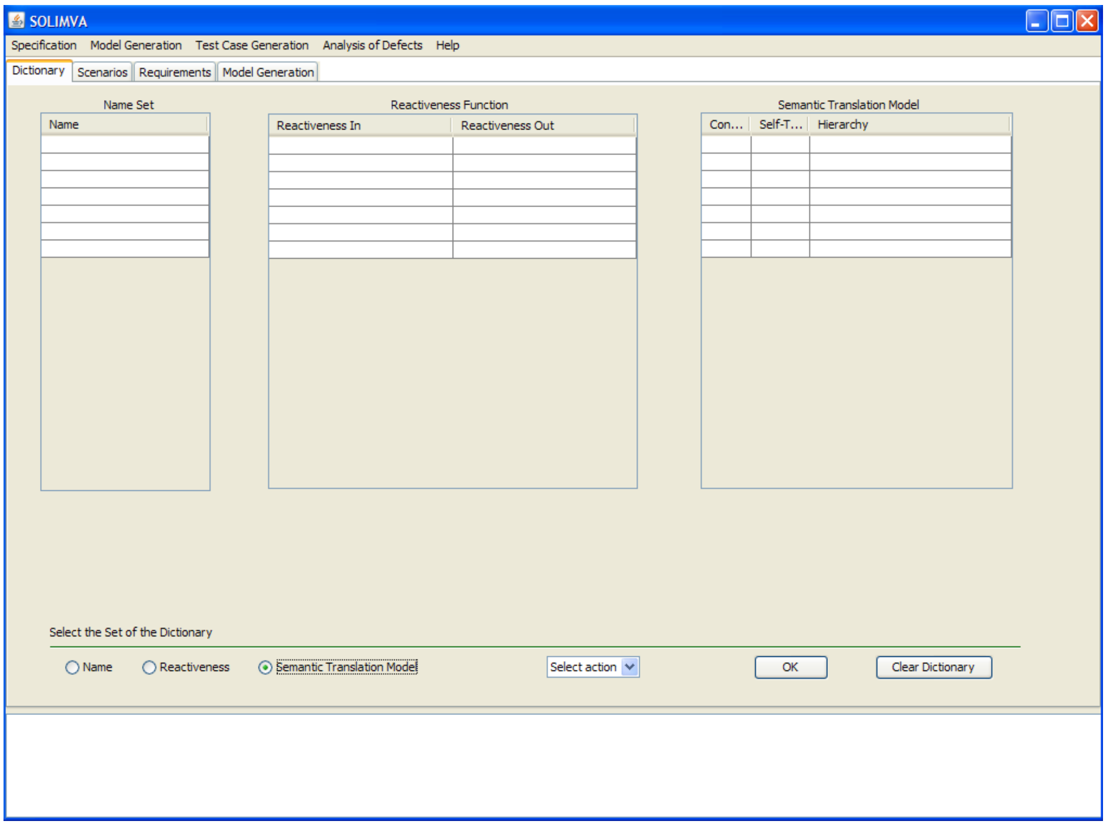
APPENDIX B - THE SOLIMVA TOOL

This appendix presents an overview of the most relevant tool that supports the SOLIMVA methodology, i.e. the SOLIMVA tool, in order to generate model-based system and acceptance test cases considering NL requirements deliverables. The Graphical User Interface of the SOLIMVA tool will be shown in sequence. The SOLIMVA tool was developed according to the Object-Oriented Programming paradigm using the Java programming language and version 6.5.1 of the *NetBeans* (NETBEANS.ORG, 2011) Integrated Development Environment. The current version of the SOLIMVA tool is 1.0.

Figure B.1 shows the Graphical User Interface of the SOLIMVA tool. Note that the interface has four tabs (recall the activities of the SOLIMVA methodology as shown in Figures 3.3 and 5.1):

- *Dictionary*. This tab allows the user to input the Dictionary;
- *Scenarios*. This tab is related to the definition of scenarios;
- *Requirements*. This tab allows the user to input the set of NL requirements that characterize a scenario;
- *Model Generation*. This tab allows the user to generate the Statechart model.

Note in Figure B.1 that the *Dictionary* tab allows the user to enter all information of the components of the Dictionary. At the far left of the tab, the names can be entered in NL. At the center, the Reactiveness (R) function can be defined. In the leftmost column (*Reactiveness In*), the user must enter the elements of the domain of R ($R.I_E$), while in the rightmost column (*Reactiveness Out*) must be the respective elements of the codomain of R ($R.O_E$). In the far right of the tab, the *Semantic Translation Model* of the Dictionary must be defined. As mentioned in Chapters 3 and 4, the Control ($S_{TM}.C$; column *Con...*) and Self Transition ($S_{TM}.F$; column *Self-T...*) sets are already predefined and the user does not need to change them. The Hierarchy ($S_{TM}.Y$) function is entered in the *Hierarchy* column in accordance with the notation *yip*/*yop* where *yip* is an element of the domain of $S_{TM}.Y$ ($Y.I_P$) while *yop* is an element of the codomain of $S_{TM}.Y$ ($Y.O_P$). Figure B.2 shows the *Dictionary* tab already filled with data on the application of the SOLIMVA methodology to the SWPDC case study.



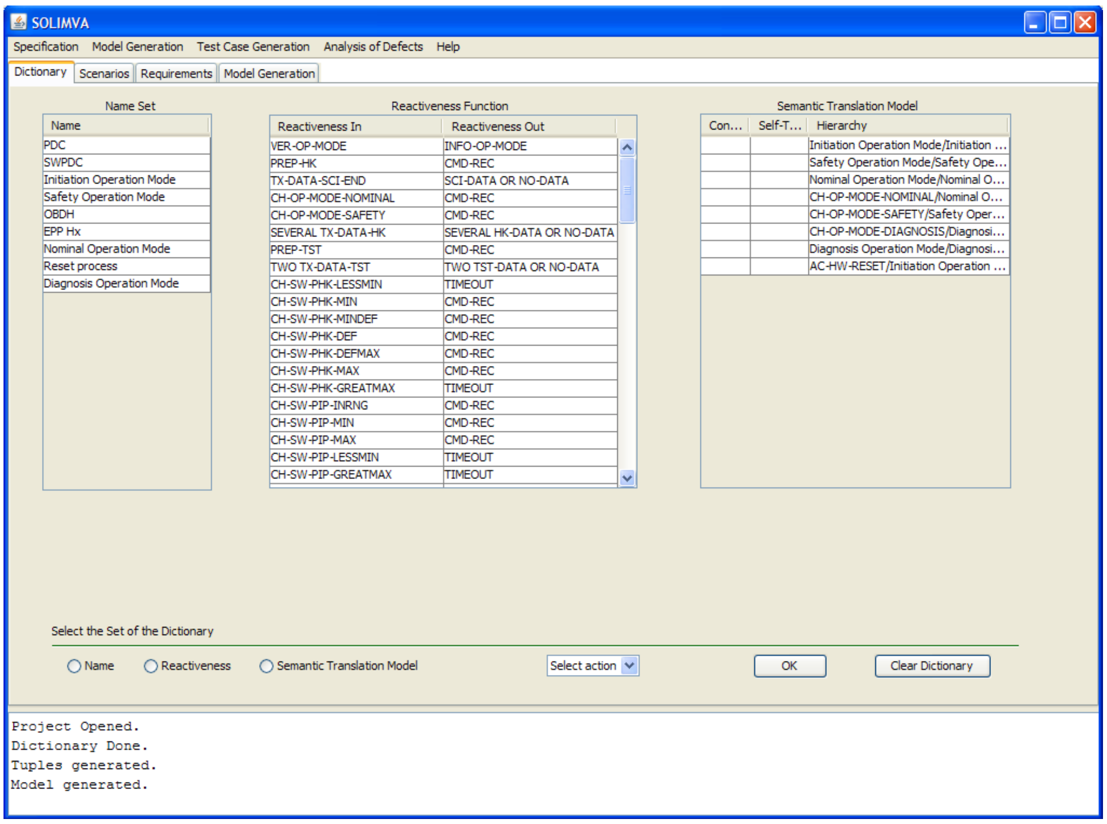
Select the Set of the Dictionary

Name Reactiveness [✓ Semantic Translation Model]

Figure B.1 - Graphical User Interface of the SOLIMVA tool

As said in Chapter 3, the definition of scenarios was accomplished by means of combinatorial designs and *TConfig* (UNIVERSITY OF OTTAWA, 2008), an external tool. Therefore, in the current version (1.0) of the SOLIMVA tool, there is no treatment of the data put in the *Scenarios* tab shown in Figure B.3. However, this tab has been already defined thinking of future implementation of a combinatorial designs algorithm as mentioned in Section 6.3. In the situation shown in Figure B.3, the algorithm implemented within the SOLIMVA tool would return the number of a factor combination in column *Run*, and the factor combination itself in the five columns (*FacA*, *FacB*, *FacC*, *FacD*, and *FacE* columns) where each column would have a level of a certain factor (In the example, it was assumed 5 factors).

Figure B.4 shows the *Requirements* tab where the user can input the set of NL requirements that characterize a scenario. In column *ReqId* should be entered the identification of the requirement and the column *Requirement* must contain the NL requirement itself. Figure B.5 shows the *Requirements* tab already filled with the set of NL requirements that characterize SOLIMVA's normal scenario 71 as shown



Name Set

Name
PDC
SWPDC
Initiation Operation Mode
Safety Operation Mode
OBDH
EPP Hx
Nominal Operation Mode
Reset process
Diagnosis Operation Mode

Reactiveness Function

Reactiveness In	Reactiveness Out
VER-OP-MODE	INFO-OP-MODE
PREP-HK	CMD-REC
TX-DATA-SCI-END	SCI-DATA OR NO-DATA
CH-OP-MODE-NOMINAL	CMD-REC
CH-OP-MODE-SAFETY	CMD-REC
SEVERAL TX-DATA-HK	SEVERAL HK-DATA OR NO-DATA
PREP-TST	CMD-REC
TWO TX-DATA-TST	TWO TST-DATA OR NO-DATA
CH-SW-PHK-LESSMIN	TIMEOUT
CH-SW-PHK-MIN	CMD-REC
CH-SW-PHK-MINDEF	CMD-REC
CH-SW-PHK-DEF	CMD-REC
CH-SW-PHK-DEFMAX	CMD-REC
CH-SW-PHK-MAX	CMD-REC
CH-SW-PHK-GREATMAX	TIMEOUT
CH-SW-PIP-INRNG	CMD-REC
CH-SW-PIP-MIN	CMD-REC
CH-SW-PIP-MAX	CMD-REC
CH-SW-PIP-LESSMIN	TIMEOUT
CH-SW-PIP-GREATMAX	TIMEOUT

Semantic Translation Model

Con...	Self-T...	Hierarchy
		Initiation Operation Mode/Initiation ...
		Safety Operation Mode/Safety Ope...
		Nominal Operation Mode/Nominal O...
		CH-OP-MODE-NOMINAL/Nominal O...
		CH-OP-MODE-SAFETY/Safety Oper...
		CH-OP-MODE-DIAGNOSIS/Diagnosi...
		Diagnosis Operation Mode/Diagnosi...
		AC-HW-RESET/Initiation Operation ...

```
Project Opened.
Dictionary Done.
Tuples generated.
Model generated.
```

Figure B.2 - SWPDC case study: the *Dictionary* tab

earlier in Table 4.7.

The *Generate Model* activity of the SOLIMVA methodology (Section 3.3) is accomplished by means of the *Model Generation* tab shown in Figure B.6. In order to generate the BSAO tuples, the user must select the *BSAO Generation* button. After this, the *BSAO to Model* button must be selected to generate the Statechart model. Observe that buttons *Clear Requirements* in Figures B.4, B.5, and *Clear BSAO Tuples and Model* in Figure B.6 must be chosen to carry out the *Clear Requirements and Model* activity of the SOLIMVA methodology.

In order to show the internal structure of the tool, Figure B.7 shows the class hierarchy of version 1.0 of the SOLIMVA tool. Such a hierarchy was created with *Javadoc* (ORACLE, 2011).

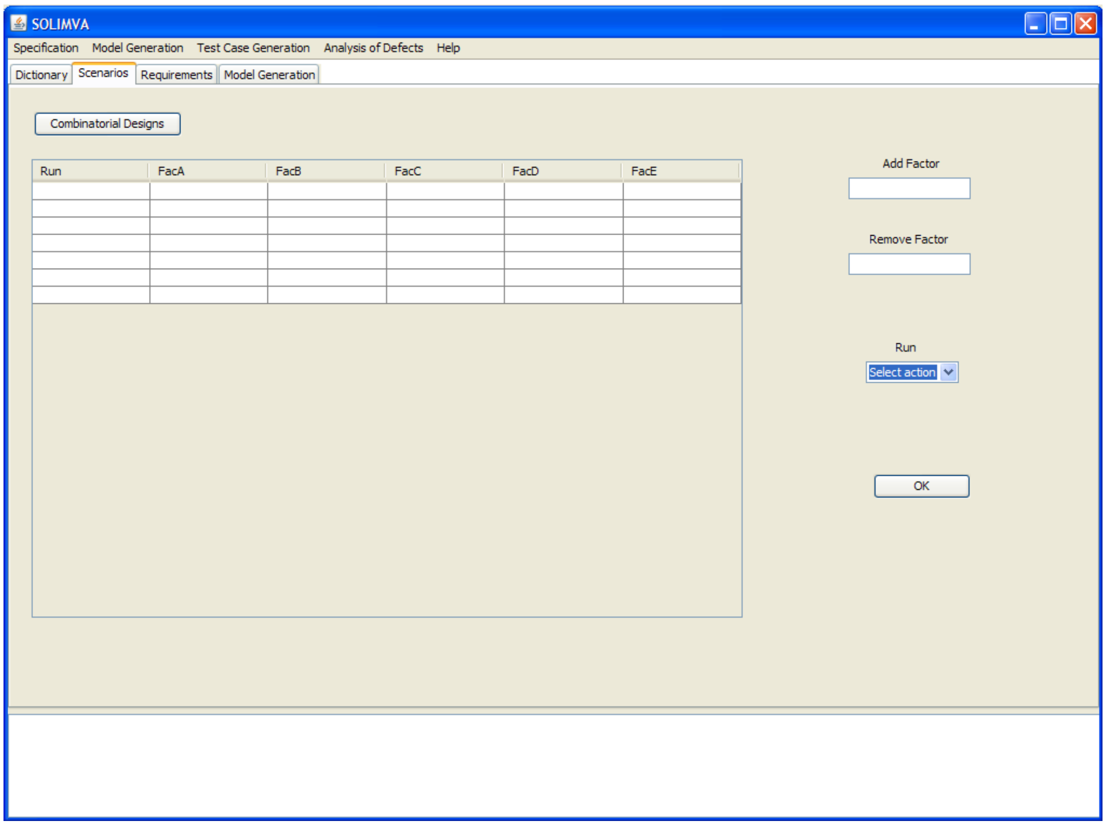


Figure B.3 - The *Scenarios* tab

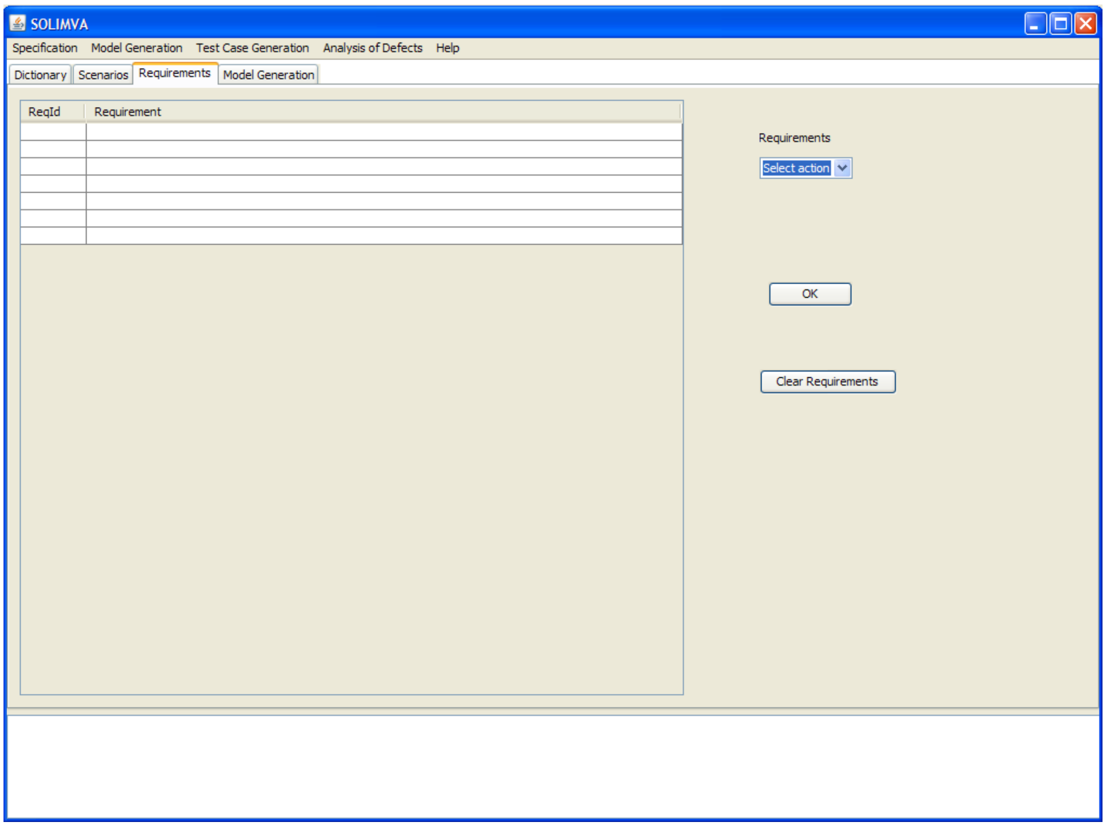
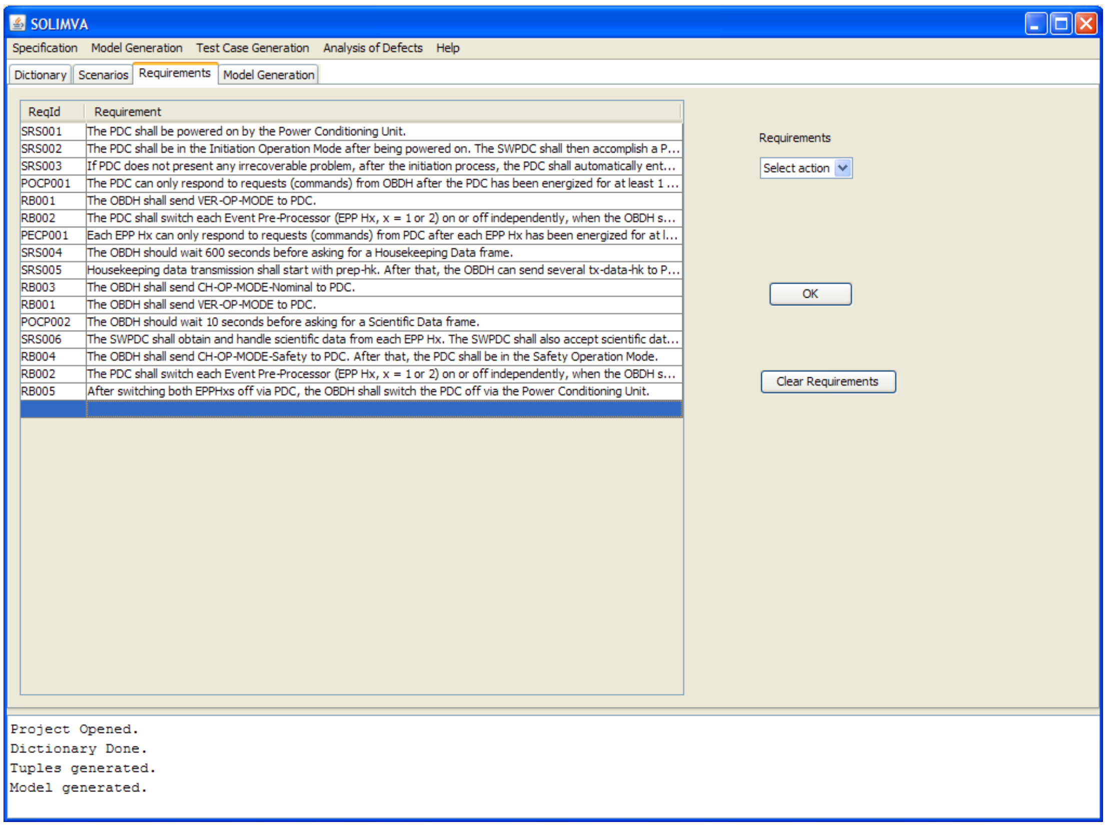


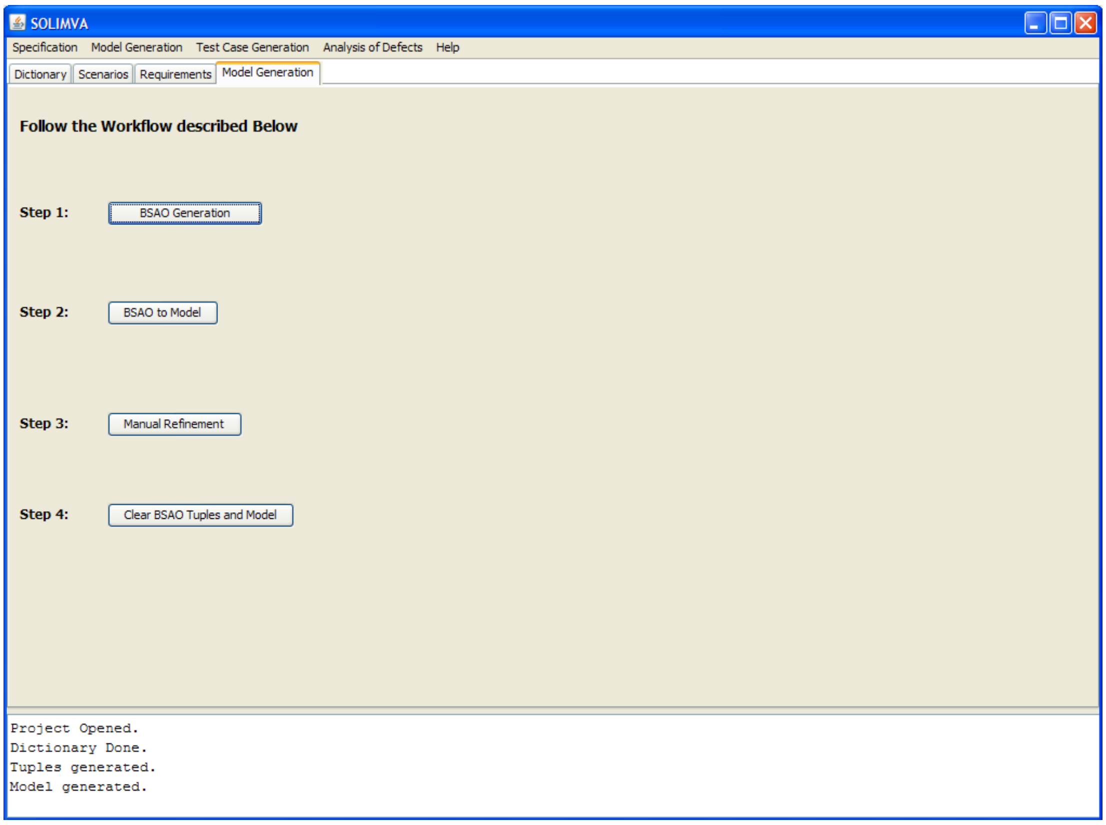
Figure B.4 - The *Requirements* tab



ReqId	Requirement
SRS001	The PDC shall be powered on by the Power Conditioning Unit.
SRS002	The PDC shall be in the Initiation Operation Mode after being powered on. The SWPDC shall then accomplish a P...
SRS003	If PDC does not present any irrecoverable problem, after the initiation process, the PDC shall automatically ent...
POCP001	The PDC can only respond to requests (commands) from OBDH after the PDC has been energized for at least 1 ...
RB001	The OBDH shall send VER-OP-MODE to PDC.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH s...
PECP001	Each EPP Hx can only respond to requests (commands) from PDC after each EPP Hx has been energized for at l...
SRS004	The OBDH should wait 600 seconds before asking for a Housekeeping Data frame.
SRS005	Housekeeping data transmission shall start with prep-hk. After that, the OBDH can send several tx-data-hk to P...
RB003	The OBDH shall send CH-OP-MODE-Nominal to PDC.
RB001	The OBDH shall send VER-OP-MODE to PDC.
POCP002	The OBDH should wait 10 seconds before asking for a Scientific Data frame.
SRS006	The SWPDC shall obtain and handle scientific data from each EPP Hx. The SWPDC shall also accept scientific dat...
RB004	The OBDH shall send CH-OP-MODE-Safety to PDC. After that, the PDC shall be in the Safety Operation Mode.
RB002	The PDC shall switch each Event Pre-Processor (EPP Hx, x = 1 or 2) on or off independently, when the OBDH s...
RB005	After switching both EPPHxs off via PDC, the OBDH shall switch the PDC off via the Power Conditioning Unit.

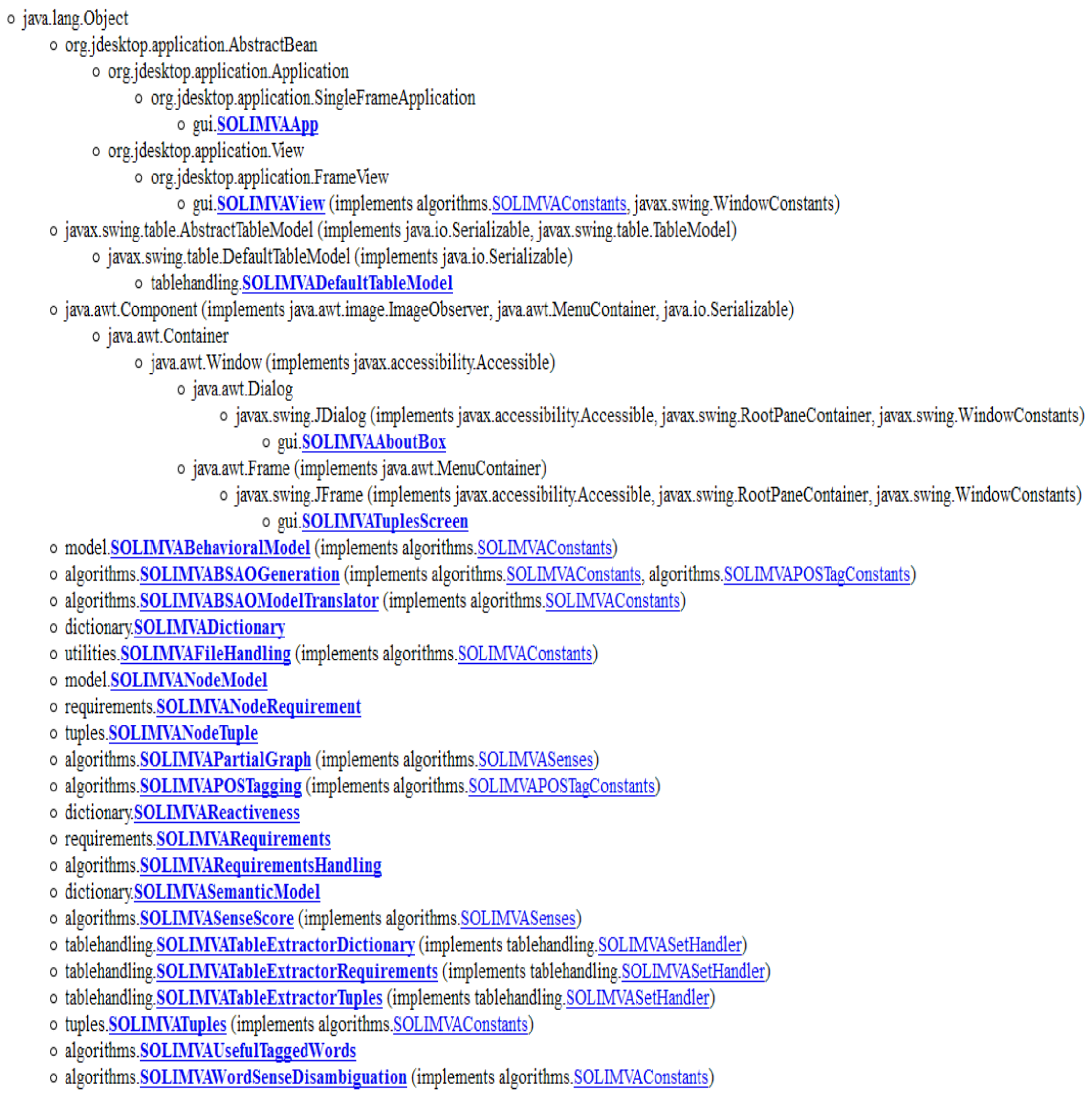
```
Project Opened.
Dictionary Done.
Tuples generated.
Model generated.
```

Figure B.5 - The *Requirements* tab already filled with the set of NL requirements that characterize SOLIMVA's normal scenario 71



```
Project Opened.
Dictionary Done.
Tuples generated.
Model generated.
```

Figure B.6 - The *Model Generation* tab



- java.lang.Object
  - org.jdesktop.application.AbstractBean
    - org.jdesktop.application.Application
      - org.jdesktop.application.SingleFrameApplication
        - gui.**SOLIMVAApp**
    - org.jdesktop.application.View
      - org.jdesktop.application.FrameView
        - gui.**SOLIMVAView** (implements algorithms.SOLIMVAConstants, javax.swing.WindowConstants)
  - javax.swing.table.AbstractTableModel (implements java.io.Serializable, javax.swing.table.TableModel)
    - javax.swing.table.DefaultTableModel (implements java.io.Serializable)
      - tablehandling.**SOLIMVADefaultTableModel**
  - java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - java.awt.Container
      - java.awt.Window (implements javax.accessibility.Accessible)
        - java.awt.Dialog
          - javax.swing.JDialog (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
            - gui.**SOLIMVAAboutBox**
        - java.awt.Frame (implements java.awt.MenuContainer)
          - javax.swing.JFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
            - gui.**SOLIMVATuplesScreen**
  - model.**SOLIMVABehavioralModel** (implements algorithms.SOLIMVAConstants)
  - algorithms.**SOLIMVABSAOGeneration** (implements algorithms.SOLIMVAConstants, algorithms.SOLIMVAPOSTagConstants)
  - algorithms.**SOLIMVABSAOModelTranslator** (implements algorithms.SOLIMVAConstants)
  - dictionary.**SOLIMVADictionary**
  - utilities.**SOLIMVAFileHandling** (implements algorithms.SOLIMVAConstants)
  - model.**SOLIMVANodeModel**
  - requirements.**SOLIMVANodeRequirement**
  - tuples.**SOLIMVANodeTuple**
  - algorithms.**SOLIMVAPartialGraph** (implements algorithms.SOLIMVASenses)
  - algorithms.**SOLIMVAPOSTagging** (implements algorithms.SOLIMVAPOSTagConstants)
  - dictionary.**SOLIMVAReactiveness**
  - requirements.**SOLIMVARequirements**
  - algorithms.**SOLIMVARequirementsHandling**
  - dictionary.**SOLIMVASemanticModel**
  - algorithms.**SOLIMVASenseScore** (implements algorithms.SOLIMVASenses)
  - tablehandling.**SOLIMVATableExtractorDictionary** (implements tablehandling.SOLIMVASetHandler)
  - tablehandling.**SOLIMVATableExtractorRequirements** (implements tablehandling.SOLIMVASetHandler)
  - tablehandling.**SOLIMVATableExtractorTuples** (implements tablehandling.SOLIMVASetHandler)
  - tuples.**SOLIMVATuples** (implements algorithms.SOLIMVAConstants)
  - algorithms.**SOLIMVAUsefulTaggedWords**
  - algorithms.**SOLIMVAWordSenseDisambiguation** (implements algorithms.SOLIMVAConstants)

Figure B.7 - Class Hierarchy of version 1.0 of the SOLIMVA tool