sid.inpe.br/mtc-m21b/2015/09.08.18.24-TDI

A METHODOLOGY TO APPLY FORMAL VERIFICATION TO UML-BASED SOFTWARE

Luciana Brasil Rebelo dos Santos

Doctorate Thesis of the Graduate Course in Applied Computing, guided by Drs. Valdivino Alexandre de Santiago Júnior, e Nandamudi Lankalapalli Vijaykumar, approved in Octuber 02, 2015.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W34P/3K7T2BB>

INPE
São José dos Campos
2015

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-m21b/2015/09.08.18.24-TDI

A METHODOLOGY TO APPLY FORMAL VERIFICATION TO UML-BASED SOFTWARE

Luciana Brasil Rebelo dos Santos

Doctorate Thesis of the Graduate Course in Applied Computing, guided by Drs. Valdivino Alexandre de Santiago Júnior, e Nandamudi Lankalapalli Vijaykumar, approved in Octuber 02, 2015.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W34P/3K7T2BB>

INPE
São José dos Campos
2015

Cataloging in Publication Data

Santos, Luciana Brasil Rebelo dos.

Sa59m A Methodology to apply formal verification to UML-based software / Luciana Brasil Rebelo dos Santos. – São José dos Campos : INPE, 2015.

xxviii + 168 p. ; (sid.inpe.br/mtc-m21b/2015/09.08.18.24-TDI)

Thesis (Doctorate in Applied Computing) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2015.

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

1. UML. 2. Formal verification. 3. Model checking. 4. SOLIMVA. 5. Formal methods. I.Title.

CDU 004.439

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

Computação Aplicada

Dr.	Solon Venâncio de Carvalho	[illegible] ***Presidente / INPE / SJCampos - SP***
Dr.	Valdivino Alexandre de Santiago Júnior	[illegible] ***Orientador(a) / INPE / São José dos Campos - SP***
Dr.	Nandamudi Lankalapalli Vijaykumar	[illegible] ***Orientador(a) / INPE / SJCampos - SP***
Dr.	Fábio Fagundes Silveira	[illegible] ***Convidado(a) / UNIFESP / São José dos Campos - SP***
Dr.	Edgar Toshiro Yano	[illegible] ***Convidado(a) / IEC/ITA / São José dos Campos - SP***

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

Título: " A METHODOLOGY TO APPLY FORMAL VERIFICATION TO UML-BASED SOFTWARE".

Aluno (a): ***Luciana Brasil Rebelo dos Santos***

São José dos Campos, 02 de outubro de 2015

"Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma - which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary."

Steve Jobs

To my beloved family: my son Pedro, my husband Oiram, and my parents Raymundo and Lucimar, who made me a better person and are the real reasons for my happiness.

ACKNOWLEDGEMENTS

It is so hard on just one page to be grateful for all who contributed to this job. It was not only five years. It was the effort of a lifetime. All my experiences and learnings were required and applied here. In advance, I apologize if I forgot to quote someone.

First, there could not be different. I want to thank my parents Raymundo and Lucimar. They both really wanted to bring me into this world. Until today, they are my examples of force, work, fight, and determination, because they were the first to show me, with their own example, that if we work hard and steady, we achieve success. They inspire me in times of trouble! I also thank my brothers, Fátima, Nenê, Jackson, Jefferson, Dinha, and Lene, who always supported me, each one in their own way. I can feel the pride and respect that you have for me. I love you all, thank you! I would also like to remind my brother-in-law Edwaldo, that, in my earliest memories, was the first of the family to encourage me to attend the best universities in the country. Only a person with the soul of a poet like him to have that view before all.

I would like to thank my advisors: Dr. Valdivino and Dr. Vijaykumar. Each one, in their own way, were essential to the progress of this work. Thank you so much for your friendship, confidence, seriousness, and support whenever it was necessary. You both improved my qualities as a researcher and professor. I hope this partnership will continue in the course of our lives.

I would also like to thank all my Graduate colleagues, in particular: Michelle, Sherfis, Sabrina, Érica Souza, Érica Golvêa, Juliana Anochi, Juliana Balera, Diego, Marlon, and Alessandro. Thank you so much for all moments of relaxation and joy. Special thanks to my first pupil, Eduardo, who was essential in the development of this work. I hope you decide to continue in Computing.

I want to thank Instituto Nacional de Pesquisas Espaciais (INPE) and CAPES, for the financial support in this research.

And finally, thank you Oiram, my most precious friend, for being a great partner and a perfect father for our son. I will never thank you enough for your patience and dedication. You supplied all the needs of our family while I dedicated to my PhD. I do love you... and thank you Pedro, now you will have your mom back just for you.

ABSTRACT

Software development organizations aim to add quality to the created products, especially those dealing with critical systems, which require high quality software. Formal Methods offer a large potential to provide more effective verification techniques. Besides, Formal Verification methods, such as Model Checking, are best applied in early stages of system design, when costs are low and benefits can be high, increasing the quality of systems. Unified Modeling Language (UML) is widely used for modeling (object-oriented) software, and its use is increasing in the aerospace industry. Verification and Validation of complex software developed according to UML is not trivial due to complexity of the software itself, and the several different UML models/diagrams that can be used to model behavior and structure of the software. This PhD thesis presents an extension of a methodology called SOLIMVA, initially developed to generate model-based system and acceptance test cases considering Natural Language requirements artifacts (SOLIMVA 1.0), and to detect incompleteness in software specifications by means of Model Checking (SOLIMVA 2.0). Such an extension generated SOLIMVA 3.0 which transforms up to three different UML behavioral diagrams (sequence, behavioral state machine, and activity) into a single Transition System to support Model Checking of software developed in accordance with UML. In SOLIMVA 3.0, properties are formalized based on use case models or requirements expressed in pure textual notation. The translation into the Transition System is done for the NuSMV model checker, but there is a possibility in using other model checkers, such as SPIN. A tool, XML Metadata Interchange to Transition System (XMITS), was developed to automate some steps of SOLIMVA 3.0 methodology. The approach was applied to two real case studies (embedded software) related to project under development at Instituto Nacional de Pesquisas Espaciais (INPE). Defects were detected within the design of these software systems showing the feasibility of the methodology. The main contribution of this PhD thesis is the transformation of a non-formal language (UML) to a formal language (language of the NuSMV model checker) towards a greater adoption in practice of Formal Methods in software development.

Keywords: UML. Formal Verification. Model Checking. SOLIMVA. Formal Methods.

UMA METODOLOGIA PARA APLICAR VERIFICAÇÃO FORMAL A SOFTWARE DESENVOLVIDO DE ACORDO COM UML

RESUMO

Organizações que desenvolvem software objetivam produzir produtos de software de qualidade, especialmente aquelas que lidam com sistemas críticos, que demandam software de alta qualidade. Métodos Formais oferecem grande potencial para prover técnicas de verificação mais efetivas. Além disso, métodos de Verificação Formal, como *Model Checking*, são aplicados de maneira mais eficiente nos estágios iniciais do projeto de software, quando os custos ainda são baixos e os benefícios podem ser altos, aumentando a qualidade dos sistemas de software. A Linguagem de Modelagem Unificada (UML) é consideravelmente utilizada para modelar software (orientado a objetos), e seu uso tem crescido na indústria aeroespacial. Verificação e Validação de sistemas complexos desenvolvidos de acordo com UML não são tarefas triviais, devido à complexidade do software em si, e a diversos diagramas/modelos UML diferentes que podem ser usados para modelar o comportamento e a estrutura do sistema. Esta tese de doutorado apresenta uma extensão de uma metodologia chamada SOLIMVA, desenvolvida inicialmente para gerar casos de teste de sistema e de aceitação baseados em modelos, considerando requisitos em Linguagem Natural (SOLIMVA 1.0), e para detectar não completude em especificações de software utilizando *Model Checking* (SOLIMVA 2.0). Tal extensão gerou a SOLIMVA 3.0, a qual transforma até três diferentes diagramas comportamentais da UML (sequência, atividades e máquina de estado) em um único Sistema de Transição de Estados para possibilitar a aplicação de *Model Checking* em software desenvolvido de acordo com a UML. Na SOLIMVA 3.0, as propriedades são formalizadas baseando-se nos modelos de casos de uso ou em requisitos expressos em notação textual pura. A tradução para o Sistema de Transição de Estados é feita para a ferramenta de *Model Checking* NuSMV, mas existe a possibilidade de se utilizar outras ferramentas, como por exemplo, SPIN. Uma ferramenta, *XML Metadata Interchange to Transition System* (XMITS), foi desenvolvida para automatizar algumas atividades da metodologia SOLIMVA 3.0. A abordagem foi aplicada em dois estudos de caso reais (software embarcado) relacionados a um projeto em desenvolvimento no Instituto Nacional de Pesquisas Espaciais (INPE). Foram encontrados defeitos nos projetos desses sistemas de software, mostrando a viabilidade da metodologia. A principal contribuição desta tese de doutorado é a transformação de uma linguagem não formal (UML) para uma linguagem formal (linguagem de entrada da ferramenta de *Model Checking* NuSMV), tendo como objetivo uma maior utilização, na prática, de Métodos Formais no processo de desenvolvimento de software.

Palavras-chave: UML. Verificação Formal. Model Checking. SOLIMVA. Métodos Formais.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

SA	–	Software Assurance
NASA	–	National Aeronautics and Space Administration
V&V	–	Verification and Validation
UML	–	Unified Modeling Language
TS	–	Transition System
SWPDC	–	Software for the Payload Data Handling Computer
SWPDCpM	–	Software for the Payload Data Handling Computer protoMIRAX
IEEE	–	Institute of Electrical and Electronics Engineers
XML	–	eXtensible Markup Language
XMITS	–	XML Metadata Interchange to Transition System
FSM	–	Finite State Machine
XMI	–	XML Metadata Interchange
LTL	–	Linear Temporal Logic
CTL	–	Computation Tree Logic
NL	–	Natural Language
GTSC	–	Geração Automática de Casos de Teste Baseada em Statecharts
IUT	–	Implementation Under Test
SD	–	Sequence Diagram
AD	–	Activity Diagram
SMD	–	State Machine Diagram
INPE	–	Instituto Nacional de Pesquisas Espaciais
Std	–	Standard
UC	–	Use Case
alt	–	alternatives
opt	–	option
par	–	parallel
gvs	–	guard value structure
txt	–	extension for text file
DC	–	Do not care
ATM	–	Automated Teller Machine
PIN	–	Personal Identification Number
OMG	–	Object Management Group
BNF	–	Backus-Naur Form
IDE	–	Integrated Development Environment
TUTS	–	The Unified Transition System
SAX	–	Simple API for XML
QSEE	–	Qualidade do Software Embarcado em Aplicações Espaciais
CIFASIS	–	Centro Internacional Franco Argentino de Ciencias de la Información
	–	y de Sistemas
OBDH	–	On-Board Data Handling

PDC	–	Payload Data Handling Computer
EPP	–	Event Pre-Processors
SRS	–	Software Requirements Specification
RPQ	–	Relatório de Pesquisa
PCD	–	Power Conditioning Unit
POST	–	Power-On Self Test
SRAM	–	Static Random Access Memory
SDRAM	–	Synchronous Dynamic Random Access Memory
CRX	–	Subsistema Câmera de Raios X
ACS	–	Attitude Control and Pointing Subsystem
FCTS	–	Flight Control and Telecommunications Subsystem
TM&TC	–	Telemetry and Command Subsystem
PSS	–	Power Supply Subsystem
GPS	–	Global Position System
SGB	–	Board Management subsystem
PC	–	Personal Computer
AMD	–	Advanced Micro Devices
RTEMS	–	Real-Time Executive for Multiprocessor Systems
CTL	–	Control
AP	–	Application Process
ESS	–	Estação de Solo
HK	–	Housekeeping
SCA	–	Subsistema de Controle de Atitude
TC	–	Telecommand
TM	–	Telemetry
VC	–	Verificação de Comando

LIST OF SYMBOLS

s	–	Segundos
ms	–	Milissegundos
MHz	–	Megahertz
MB	–	Megabyte

CONTENTS

Page

1 INTRODUCTION

Almost 30 years ago, Parnas and Clements (PARNAS; CLEMENTS, 1986) argued that "... the picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic. No system has ever been developed in that way, and probably none ever will." After such a long time, professionals still face the problem of producing high quality software systems. Yet, this is not a privilege of the software industry; quality is a desirable property related to every single product, whatever is its scope. As a result, quality is an important concept in the context of Software Engineering. Godbole (GODBOLE, 2006) presents several definitions of quality, to name a few, defect level, defect origins, product complexity, conformance to requirements, user satisfaction, and robustness. Regardless of definition, developers agree that high-quality software is an important goal and achieving it requires huge effort from organizations involved in developing software.

Critical systems demand high reliable software, and it is essential to ensure that the software has the fewest number of defects when it is released for use. Software Assurance (SA), according to the National Aeronautics and Space Administration (NASA) (NASA, 2009), includes several disciplines, to name a few: Software Quality (comprised of the functions of Software Quality Engineering, Software Quality Assurance and Software Quality Control); Software Safety; Software Reliability; Software Verification and Validation; and Software Independent Verification and Validation. Hence, Verification and Validation (V&V) plays a key role of getting quality and has been gaining importance in academia as well as in private sector.

V&V activities are usually divided into static and dynamic. The static ones do not require the execution or even the existence of a program or executable model to be performed. The dynamic ones are based on the execution of a program or model (DELAMARO et al., 2007). V&V activities aim to ensure that:

a) the software is being correctly developed,

b) the software that is being developed is correct.

V&V encompasses a large range of activities and techniques, of which one can mention **testing** (MARTHUR, 2008)(DELAMARO et al., 2007), **inspection** (IEEE, 1990)(GILB et al., 1993), and **Formal Verification** (BAIER; KATOEN, 2008) (CLARKE et al., 1999), (SANTIAGO JÚNIOR, 2011). To determine whether there are any defects in human thoughts, actions, and the products generated, the process of testing is applied. The primary goal of testing is to determine if the thoughts, actions, and products are as desired, that is, they conform to the requirements (MARTHUR, 2008). Another technique is inspection. It is a technique that relies on visual examination of developed products to detect defects, violation of development standards, and other problems (IEEE, 1990). Software requirements specifications, design documents, source code, and UML diagrams are examples of deliverables which can be examined within inspection. On the other hand, 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).

V&V activities are usually time-consuming, specially if critical/intricate systems are considered. Techniques are developed to facilitate and make the efforts easier with these tasks.

1.1 Motivation

A recent paper (PETRE, 2013) reports interviews with 50 professional software engineers in 50 companies about the use of the Unified Modeling Language (UML) (OMG, 2015) in practice. Although considered as "de facto" standard by some authors, the majority of professionals interviewed simply do not use UML, and those who do use it tend to do selectively and often informally. Lack of context, overhead of understanding the notations, and issues of synchronization/consistency are some problems mentioned by the practitioners.

On the other hand, a survey (UBM TECH, 2013) of the embedded systems market worldwide presented in the same year (2013) of the study above shows that 19% of the professionals report on using UML for system level design. This survey was undertaken with 2,098 professionals which amounts to 400 practitioners claiming their organizations adopt UML. This is an extensive survey including organizations around the world and addressing various aspects of the development of embedded systems, such as design environment, embedded design process, operating systems, microprocessors used, system level design (where UML is cited), among others. Despite the criticisms presented in Petre (PETRE, 2013), this embedded market survey shows that UML is indeed used in practice, even though its use might be specific to a particular part (e.g. design) of the project in many cases. Modeling systems for object oriented and/or embedded software development is an approach that has been employed by researchers and practitioners, specially by means of the several UML behavioral diagrams.

In UML, dynamic aspects of system behavior can be specified by interactions (i.e. sequence diagrams). UML behavioral state machine (variant of Harel's Statecharts (HAREL, 1987)) and activity diagrams give a view of the system that is associated with instances of classes. These types of diagrams represent complementary views of the system, but, at the same time, hide redundant descriptions of the same aspects of the system. This gives the opportunity for V&V techniques to ensure the consistency of these descriptions (KNAPP; MERZ, 2002). Nevertheless, V&V of complex software developed according to UML is not trivial due to complexity of the software itself, and the several different UML models/diagrams that can be used to model behavior and structure of the software.

A major challenge in software and systems development process is to advance defect detection at early stages of their life-cycles. Formal Methods offer a large potential to obtain an early integration of verification in the design process, and to provide more effective verification techniques (BAIER; KATOEN, 2008). Besides, Formal Verification methods, Model Checking and Theorem Proving, are best applied in early stages of system design, when costs are low and benefits can be high, increasing the quality of systems.

Model Checking (BAIER; KATOEN, 2008),(CLARKE; EMERSON, 2008),(QUEILLE; SIFAKIS, 1982) is a Formal Verification method which has been receiving much attention from academic community due to its mathematical foundations. However, Model Checking is not widely used in practice due to aspects such as high learning curve and cost, and the lack of commercially supported tools. Therefore, the level of automation for Formal Verification methods should be increased so that they can be used as easily as using a compiler. In this line, approaches that translate industry non-formal standards such as UML to model checkers notation are a great step towards a wide acceptance of Formal Methods in every day software development (SANTIAGO JÚNIOR, 2011). As stated by Schäfer, "...adoption of Formal Methods will be easier when they can be applied within standard development process and when they are based on standard notation" (SCHÄFER et al., 2001).

Transition System, also called finite-state model, is a standard class of models to represent hardware and software systems (BAIER; KATOEN, 2008). They are often used as models to describe the behavior of systems. Basically, they are directed graphs where nodes represents states, and edges model transitions, i.e, state changes. Such a system evolves through its state space assuming different configurations, where a configuration can be understood as the set of states to which the system abides at any particular moment (DEBBABI et al., 2010). Model Checking is a formal automatic verification technique for finite state systems that checks temporal logic specifications on a given model. In the context of verifying design models expressed as UML behavioral diagrams, several works explore the idea of Transition Systems, but using single diagrams (MIKK et al., 1998),(LATELLA et al., 1999),(KONRAD; CHENG, 2006),(LAM, 2007),(ESHUIS, 2006),(ANDERSON et al., 1996),(DUBROVIN; JUNTTILA, 2008),(UCHITEL; KRAMER, 2001). However, Transition System concept has a general nature and a broad range of behavioral diagrams, such as activity, sequence, and behavioral state machine can be conveniently adapted to use this concept (DEBBABI et al., 2010).

1.2 Objective

Considering all that has been exposed so far, **the objective of this PhD thesis is to transform a non-formal language (UML) to a formal language (language of a model checker) in order to detect defects within the design of the software product**. An effort to achieve this goal, where the mathematical complexity of Model Checking is partially hidden from the professional and using UML as the input notation for modeling, can generate a scientific/technological solution with great potential to be used for improving the quality of real and complex software products.

1.3 Proposal to Meet the Objective

In order to achieve the goal set for this PhD thesis, a methodology [1] called SOLIMVA (SANTIAGO JÚNIOR, 2011) was extended and thus this work developed SOLIMVA 3.0 (SANTOS et al., 2014a),(SANTOS et al., 2014b),(ERAS et al., 2015). The approach proposed in SOLIMVA 3.0 considers the properties generated from UML use case models or requirements expressed in pure textual notation (Natural Language), and the Transition System translated from up to three UML behavioral diagrams: sequence, activity, and behavioral state machine. Then, Model Checking can be used

[1]According to the Oxford Dictionary (PRESS, 2015), a methodology is “a system of methods used in a particular area of study or activity”. On the other hand, a method is “a particular procedure for accomplishing or approaching something, especially a systematic or established one”. Thus, both versions 1.0 and 2.0 of the SOLIMVA methodology as well as the version 3.0 of this methodology (contribution of this PhD thesis) would be more appropriately defined as “method” rather than “methodology”. However, in the context of Software Engineering, words “methodology” and “method” are largely used interchangeably, although many researchers believe it is important to differentiate between them. But there is no consensus as can be observed in some discussions (WIKI, 2015). Therefore, this PhD thesis will follow the traditional nomenclature adopted in Software Engineering and the term “SOLIMVA methodology” will be used instead of “SOLIMVA method”.

to ensure that the behavior of the system satisfies the requirements, that is, whether the properties are satisfied by the Transition System that represents the behavior of the application under evaluation. It is important to mention that such a model, automatically generated, will have a unified view of different perspectives of behavioral modeling of the system obtained by using these three UML diagrams.

The verification process established in SOLIMVA 3.0 essentially consists of sequence of scenarios to be checked. The analyst gathers requirements from software specifications. In practice, such requirements are generally expressed within UML use case models or simply in Natural Language. SOLIMVA 3.0 suggests using specification patterns (DWYER et al., 1999) to direct the formalization of properties in Computation Tree Logic (BAIER; KATOEN, 2008). The UML diagrams (sequence, behavioral state machine, and activity) are input to a tool developed to support SOLIMVA 3.0: XML Metadata Interchange to Transition System (XMITS) (SANTOS et al., 2014b),(ERAS et al., 2015). Hence, XMITS automatically generates a single, unified TS in the notation of the NuSMV model checker (KESSLER, 2015). By running NuSMV with the unified TS and the properties in CTL, it is possible to determine if there are defects with the design of the software product. In case the TS does not satisfy a certain property, a counterexample is presented by the model checker.

In order to facilitate the evaluation of the approach, case studies were conducted applying the methodology/tool to two real case studies (embedded software) of the space application: SWPDC - Software for the Payload Data Handling Computer (SANTIAGO et al., 2007) and SWPDCpM - Software for the Payload Data Handling Computer - protoMIRAX Experiment (BRAGA et al., 2015). These software systems are related to the balloon-borne high energy astrophysics experiment called protoMIRAX under development at Instituto Nacional de Pesquisas Espaciais (INPE - National Institute for Space Research).

As previously pointed out, SOLIMVA 3.0 extends SOLIMVA, a methodology initially developed to generate model-based system and acceptance test cases considering Natural Language requirements artifacts (SOLIMVA 1.0), and to detect incompleteness in software specifications by means of Model Checking (SOLIMVA 2.0) (SANTIAGO JÚNIOR, 2011).

By including Formal Verification in the SOLIMVA methodology, this enriches the V&V process, addressing not only testing and inspection but also Formal Verification. The reasons why the SOLIMVA methodology have been updated in the order of versions (1.0, 2.0, 3.0) and not of releases (e.g. 1.0, 1.1) is precisely because each new version deals with a different V&V technique (testing - 1.0, inspection - 2.0, Formal Verification - 3.0). As stated by Mathur (MARTHUR, 2008), Formal Verification can be viewed as a complementary technique to software testing. Thus, combining both software testing and Formal Verification can reduce the likelihood of failures.

1.4 Contributions

The main contribution of this thesis is **the transformation of a non-formal language** (UML) **to a formal language** (language of the NuSMV model checker) **towards a greater adoption in practice of Formal Methods in software development**. The methodology can be applied throughout the lifecycle, even before software coding. This strategy overcomes some of the limitations of existing approaches (MIKK et al., 1998),(LATELLA et al., 1999),(KONRAD; CHENG, 2006),(LAM, 2007),(ESHUIS, 2006),(ANDERSON et al., 1996),(DUBROVIN; JUNTTILA, 2008),(UCHITEL; KRAMER, 2001),(BARESI et al., 2011),(MIYAZAWA et al., 2013),(BEATO et al., 2005),(CORTELLESSA; MIRANDOLA, 2002),(MERSEGUER et al., 2002), because it has the following features:

a) it uses different behavioral diagrams, when most of the studies use only one single diagram;

b) it detects design defects considering functional requirements of the software product, when some researches focus on specific types of requirements, such as performance;

c) it demands only behavioral diagrams, that are often present in software documentation, when other works require a very large amount of artifacts, including for example, structural and behavioral diagrams.

Additionally, the following secondary contributions can be asserted:

a) Implementation of a tool, XMITS, that allows the automated translation of the UML diagrams into the notation of a recognized model checker;

b) Application of SOLIMVA 3.0 methodology to two real space software products, SWPDC and SWPDCpM, showing the potential for a wide acceptance of Formal Verification for the development of complex software systems.

1.5 Document Organization

This chapter has presented the context, the motivations that led to the development of this PhD thesis, the objective, as well as the contributions of this PhD thesis. The organization of the remaining text is as follows:

a) Chapter 2 provides the theoretical basis for developing this PhD thesis, including basic concepts, UML and its diagrams, Formal Verification and Model Checking, an overview of the SOLIMVA methodology, and related work, emphasizing its differences with respect to this thesis.

b) Chapter 3 presents the proposal itself, that is, the solution to use Formal Verification for software developed in accordance with UML. Version 3.0 of SOLIMVA methodology is explained, through a running example, adressing Formal Verification.

c) In Chapter 4, the tool that was developed to support the proposed methodology is explained, as well as its architecture.

d) In Chapter 5, the results of the application of the methodology to both case studies, SWPDC and SWPDCpM, are presented.

e) Conclusions, contributions, final remarks, and future work are in Chapter 6.

f) Appendix A contains additional informations about SWPDC case study. Appendix B contains additional informations about SWPDCpM case study. Appendix C contains XMITS usability aspects and class diagrams.

2 THEORETICAL BASIS

This chapter presents the theoretical basis for developing this PhD thesis. The issues discussed are basic concepts related to V&V, UML, Formal Verification and Model Checking, and SOLIMVA 1.0 and 2.0 methodologies. Besides, related research literature is presented where some approaches that use Formal Verification and UML are emphasized. The main differences between these studies and this PhD thesis are also stressed.

2.1 Basic Concepts

As mentioned in Chapter 1, V&V is a discipline related to Software Assurance (SA). Verification refers to tasks that ensure the software correctly implements a specific task. Validation refers to other tasks which ensure that the software that has been built is according to customer requirements. Boehm states the same in a different way (BOEHM, 1981):

Verification: “Are we building the product right?”

Validation: “Are we building the right product?”

There is also a definition according to the IEEE (Institute of Electrical and Electronics Engineers) Standard Glossary of Software Engineering Terminology (IEEE, 1990) that states:

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”.

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”.

Based on the above definitions, Validation refers to check that the sofware is according to requirements. Verification helps to determine whether a high-quality software has been produced, but does not garantee that the software is indeed useful.

There is still another definition for Verification, acording to IEEE (IEEE, 1990): Verification is a formal proof of program correctness. This willl be the adopted definition when Formal Verification is introduced.

Other important concepts for Software Engineering are: fault, error, failure, and defect. In this thesis, it is utilized the definition given also by the IEEE. According to the IEEE Std 610.12-1990 (IEEE, 1990):

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

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.

Failure: (1) An incorrect result. For example, a computed result of 12 when the correct result is 10. (2) The inability of a system or component to perform its required functions within specified performance requirements.

Because of the definition, 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 causes an error because sometimes a part of the source code has never been exercised neither during the testing activities nor after the product was delivered to the customer.

Defect is another term used as a synonym of fault. So, an incorrect instruction can be considered both a fault or a defect.

2.2 Unified Modeling Language - UML

UML (OMG, 2015) is a visual language that has been developed to support the design of complex object-oriented systems. It was introduced in the late 90s, and is currently in version 2.5 (when implementing the methodology/tool of this work, UML 2.4.1 was used, because it was the available version). It contains two parts: a model, and a set of diagrams. The model can be considered as a description of the diagrams, which are used for visualization (MAKINEN, 2007). UML diagrams can be divided into two broad categories: structural and behavioral diagrams. The UML structural diagrams are used to model the static organization of the different elements in the system, whereas behavioral diagrams focus on the dynamic aspects of the system (SARMA; MALL, 2009). As the interest is in verifying the system behavior, UML behavioral diagrams were used. This section discusses only the UML diagrams that are relevant in the context of this thesis.

2.2.1 Use Case

"A use case is a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal" (COCKBURN, 2000). Use cases describe the business rules, and because of this, they are excellent to understand what the system may or may not do, according to the user perpective. A use case describes how a user interacts with the system by defining the steps required to accomplish a specific goal (PRESSMAN, 2010). The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system.

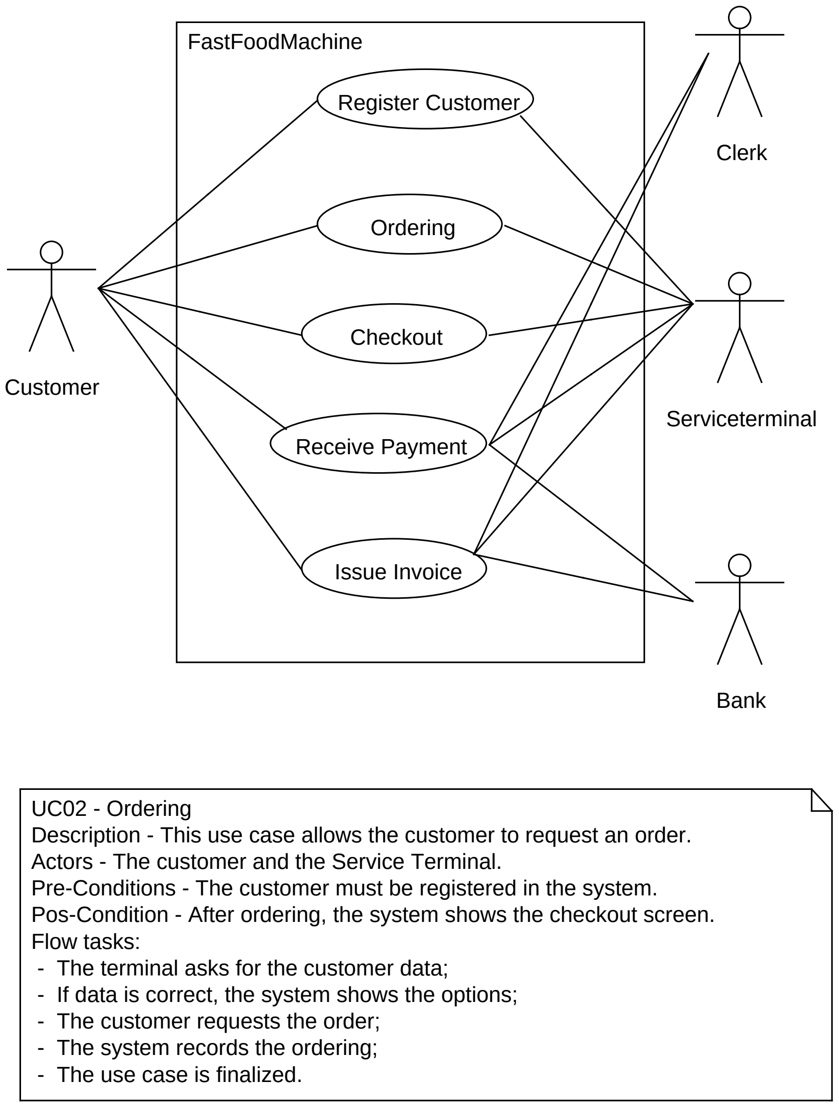
A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence ('main success scenario') and some variations ('extensions and sub-variations') (COCKBURN, 2000). The mainline sequence represents the normal interaction between a user and the system, that is, the most occurring sequence of interactions. In this thesis, the mainline sequence and each one of its variations are considered **scenarios**.

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing, as can be seen in Figure 2.1. It shows a use case diagram representing a fast food machine, as well as a description of the use case *Ordering*. In this thesis, requirements and properties are extracted from the use cases description or from requirements described in Natural Language.

2.2.2 Sequence Diagram

A sequence diagram describes how groups of objects collaborate on some behavior over time. It registers the behavior of a single use case and displays objects and messages passed between these objects in the use case. The sequence diagram follows the approach based on temporal order of the messages, that is, the emphasis is on the temporal distribution of messages.

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. The vertical dashed line is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur. Each message is



UC02 - Ordering
Description - This use case allows the customer to request an order.
Actors - The customer and the Service Terminal.
Pre-Conditions - The customer must be registered in the system.
Pos-Condition - After ordering, the system shows the checkout screen.
Flow tasks:
- The terminal asks for the customer data;
- If data is correct, the system shows the options;
- The customer requests the order;
- The system records the ordering;
- The use case is finalized.

Figure 2.1 - Example of UML use case diagram and its description

labeled with the message name. Some control information can also be included.

UML 2.0 introduced the concept of a combined fragment to capture complex procedural logic in a sequence diagram. A combined fragment is one or more processing sequences grouped together and executed under specific named circunstances. Some of the important fragments are (OMG, 2011):

alternatives (*alt*): it works as an *if-else* in procedural logic.
option (*opt*): it is like the *alt* fragment, but without *else*.
parallel (*par*): it models parallel messages.

loop (*loop*): it represents the *loop* in procedural logic.

The fragments *alt, opt*, and *loop* have guards which can assume values *true* or *false*. Figure 2.2 shows a sequence diagram for the use case *Ordering* of the fast food machine. The messages that are within the combined fragment *opt* only occur if the value of guard *data ok* is *true*. Otherwise, only the first and the last messages are sent. This occurs for all fragments which have guards: the messages within the fragment only are sent if the guard is set with value *true*.

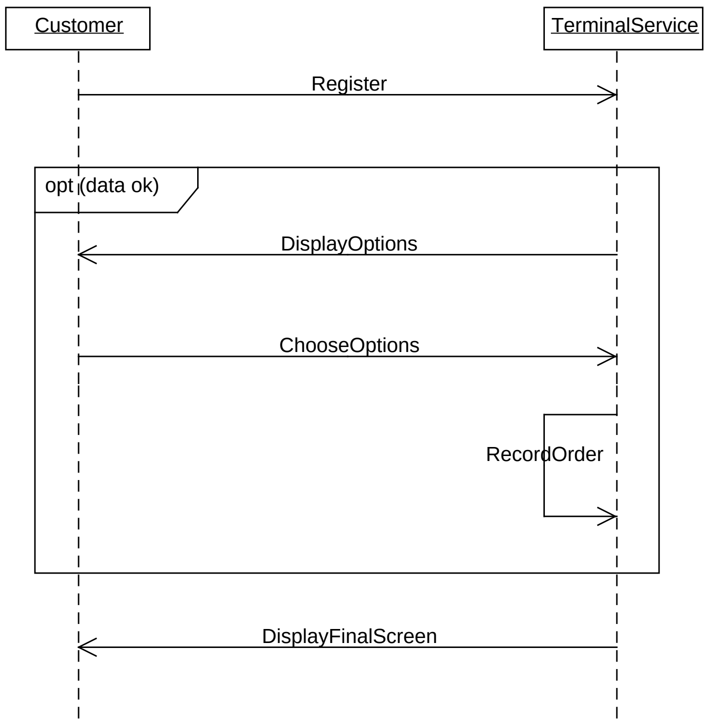


Figure 2.2 - Example of UML sequence diagram using combined fragment *opt*

In short, system requirements are represented in the use cases, i.e., use cases model **what is** the problem. The sequence diagrams show **how** the model will get the desired objective. This diagram is constructed from the use cases diagrams, showing interactions between objects in a scenario.

2.2.3 Activity Diagram

An activity diagram depicts the dynamic behavior of a system (or part of a system) through the flow of control between actions that the system performs (PRESSMAN, 2010). It is similar to a flowchart and can show concurrent flows. They evaluate better the conditions by which the instances come to certain decisions. It is common to find definitions of activity diagrams that consider them as flowcharts.

Besides, an activity diagram focuses on representing activities or parts of processing which may or may not correspond to the methods of classes. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, this must be identified through conditions.

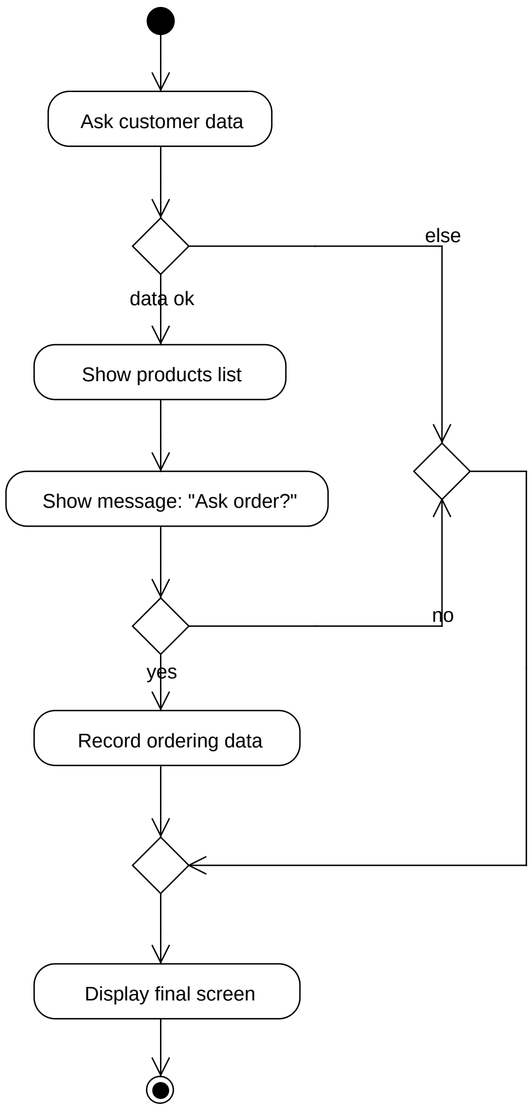


Figure 2.3 - Activity diagram for the use case *Ordering*

Activity diagrams can be very useful to understand complex processing activities involving many components. Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes. They support description of parallel activities and synchronization aspects involved in different activities. Main components of such diagrams are (OMG, 2011):

Activities: model the behavior to be performed.
Transition: models the flow of an activity to another.
Action: models transformation.
Decision: depending on a condition, it shows different transitions. Decisions are accompanied by boolean guards.
Fork: it separates a transition in several other transitions that are executed at the same time.
Join: junction of transitions that come from fork.

Figure 2.3 shows an activity diagram for the use case *Ordering*.

2.2.4 Behavioral State Machine Diagram

Behavioral state machine models an object's states, the actions that are performed depending on those states, and the transitions between the states of the objects. They are designed to evaluate the behavior of instances, i.e., the sequence of actions that affect the progress of instances, based on a reaction to events. State machine can be used to specify behavior of various model elements. For example, they can be used to model the behavior of individual entities (e.g., class instances). The state machine formalism described is an object-based variant of Harel statecharts (OMG, 2011).

Behavioral state machine diagram is also called statechart diagram. In UML, each class has an optional state machine that describes the behavior of its instances (the objects). It is normally used to model how the state of an object changes in its lifetime. Statechart diagrams are good at describing how the behavior of an object changes across several use case executions. Statecharts diagrams are based on the finite state machine (FSM) formalism. This state machine receives events from the environment and reacts to them. This diagram specifies the possible states that an object may assume, the transitions allowed at each state, the events that can cause transitions to occur and the actions that may occur in response to events.

States of an object are essentially determined by the values that certain variables (attributes) of the object may assume. A Statechart is hierarchical model of a system and introduces the concept of a composite state (also called nested state). Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with states and can take longer. An activity can be interrupted by an event.

A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is placed along side the arrow. A guard to the transition can also be assigned. A guard is a boolean logic condition. The transition can take place only if the guard evaluates to *true*. The syntax for the label of the transition is shown in 3 parts: event[guard]/action.

A state in a statechart diagram can either be simple or composite type. A simple state, also known as a basic state, does not have any sub-states. A composite state, on the other hand, consists of one or more regions. A region is a container for sub-states. The notion of a composite state makes a statechart model a hierarchical diagram. A composite state can either be sequential or concurrent. In a sequential type of composite state, the state is considered to be an *exclusive-or* of its sub-states. That is, a composite state can be in any one of its sub-states, but not in more than one sub-state at any time. But, in a concurrent type, the state is determined by an *and* logic of its sub-states and the object is considered to be in all the concurrent states at the same time. Figure 2.4 shows an example of Behavioral State Machine for the use case *Ordering*. It is possible to see a composite state (Serving customer).

Activity X Behavioral State Machine

Both state machine and activity diagrams are state machines. Any state machine aims to assess the dynamic aspects of a system model and the following elements are always identified: states, inputs, outputs, transitions, an initial state, and a final state (MATOS, 2002).

Both activity and Statechart diagrams model the dynamic behavior of the system. Activity diagram is essentially a flowchart showing flow of control from activity to activity. A statechart diagram shows a state machine emphasizing the flow of control from state to state.

Activity diagrams may stand alone to visualize, specify, and document the dynamics of a society of objects or they may be used to model the flow of control of an operation. Statechart diagrams may be attached to classes, use cases, or entire systems

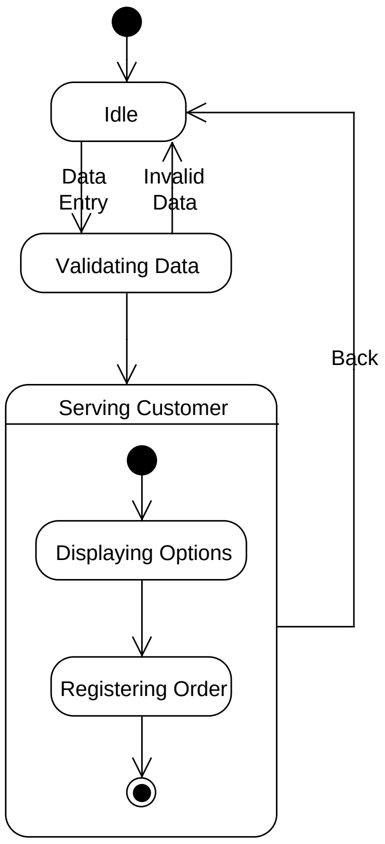


Figure 2.4 - State Machine for the use case *Ordering*

in order to visualize, specify, and document the dynamics of an individual object.

2.3 Formal Verification Methods

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). Formal Verification constructs mathematical proofs about the behavior of computer hardware or software, and has strong connections with theoretical computing. The methods for analysis are known as *Formal Verification Methods* and they can be broadly classified into: **Theorem Proving** and **Model Checking**. Both are explained in the next sections.

The properties to be validated are mainly obtained from the system's specification. The specification describes what the system has to do and what not, and thus can be used as basis for any verification activity. A defect is found if the system does not satisfy one of the specification's properties. The system is said to be "correct" whenever it satisfies all properties obtained from its specification. Therefore, correctness is relative to a specification, and is not an absolute property of a system (BAIER; KATOEN, 2008). A view of verification is presented in Figure 2.5.

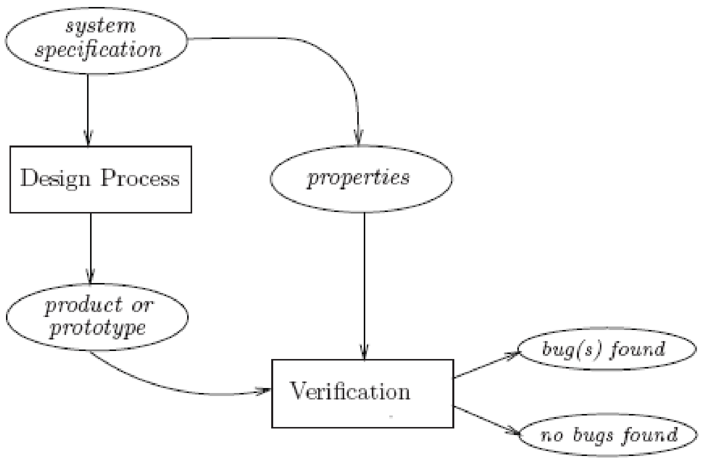


Figure 2.5 - View of a system verification. Baier and Katoen (2008)

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques such as exhaustive exploration (Model Checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing) (BAIER; KATOEN, 2008). The next section gives a brief description of Theorem Proving.

2.3.1 Theorem Proving

Theorem Proving is a proof-based approach to Formal Verification. In this method, the system that is being analyzed is modelled as a set of mathematical definitions using formal mathematical logic. The desired properties of the system are derived as theorems that follow from these definitions (AMJAD, 2004). There is a need to prove theorems in order to establish mathematical theorems, as well as in order to establish the correctness of software and hardware.

According to (SETZER, 2008) there are four ways of Theorem Proving:

a) Theorem Proving by hand: this is what mathematicians do all the time. As it is very human-dependent, there is the problem of errors and it is unsuitable for verifying large software and hardware systems;

b) Theorem Proving with some machine support: machine checks the syntax of the statements, creates a good layout, translates it into different languages, but Theorem Proving is still done by hand;

c) Interactive Theorem Proving: proofs are fully checked by the system. However, proof steps have to be carried out by the user;

d) Automated Theorem Proving: the theorem is shown by the machine. It is the task of the user to state the theorem, bring it into a form so that it can be solved.

Techniques have been developed to automate the process of derivation or proof, by using computers. 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 (BAIER; KATOEN, 2008).

The advantage of proof-based approach is that it can handle complex systems because it does not have to directly check each and every state. The disadvantage is that it requires human insight and creativity to complete the proofs, which requires time-consuming manual labour (AMJAD, 2004).

The other Formal Verification method discussed in the present work is Model Checking (CLARKE; EMERSON, 2008) (QUEILLE; SIFAKIS, 1982). Model Checking is a Formal Verification method that starts from a formal system specification. It is detailed in the following section.

2.3.2 Model Checking

Model Checking is a method that is executed automatically to verify if a model of a system meets certain specifications. According to Baier (BAIER; 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 design of complex systems, large time and effort are spent on V&V. Techniques are developed to reduce and ease the V&V efforts while increasing their coverage. During the last decades, research in Formal Methods has led to the development of some very promising verification techniques that facilitate the early detection of defects. Investigations have shown that Formal Verification procedures would have revealed the exposed defects in, e.g., the Ariane-5 missile, Mars Pathfinder, Intel's Pentium II processor, and the Therac-25 therapy radiation machine (BAIER; KATOEN, 2008).

Model Checking is a verification technique that explores all possible system states in a brute-force manner. The software tool that performs the Model Checking, examines all possible system scenarios in a systematic manner. Hence, it can be proved that a given system model truly satisfies a determined property. Baier (BAIER; KATOEN, 2008) states that "even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using Model Checking". However, it is a challenge to examine all possible state spaces.

The Model Checking approach can be viewed in Figure 2.6. There are properties obtained from the requirements that reveals what the system should do and not to do. The properties are formalized using some sort of temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). A model is generated usually from the system's pseudocode and describes the behavior of the transition system (finite-state model). The model checker examines all relevant system states to check whether they satisfy the desired property. If a state violates the property under consideration, the model checker provides a counterexample showing a trace that indicates the violation.

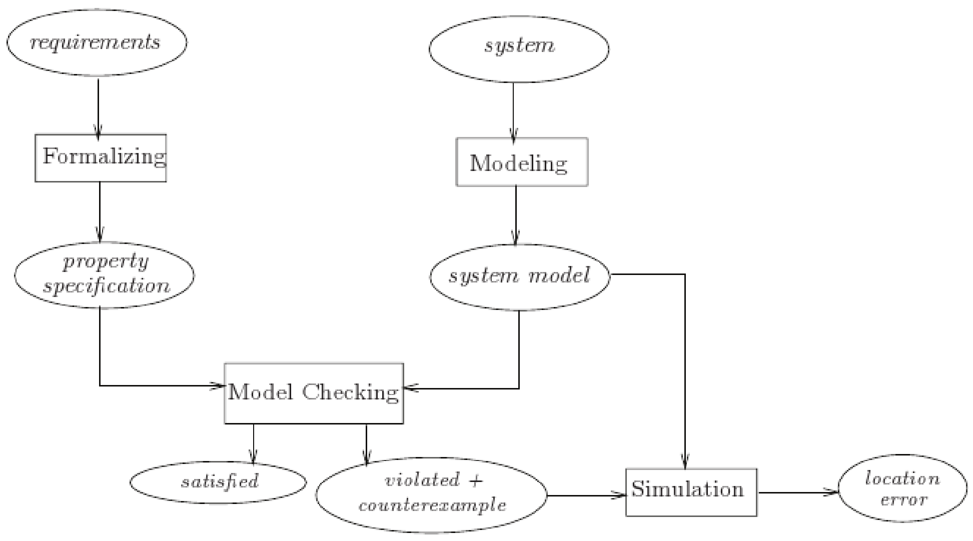


Figure 2.6 - Schematic view of Model Checking. Baier and Katoen (2008)

Model Checking Process

In applying Model Checking to a design, the following different phases can be distinguished:

- Modeling phase:
  - model the system under consideration using the model description language of the model checker at hand;
  - formalize the property to be checked using the property specification language.
- Running phase: run the model checker to check the validity of the property in the system model.
- Analysis phase:
  - property satisfied? So check next property (if any);
  - property violated?
    a) analyze generated counterexample;
    b) refine the model, design, or property;
    c) repeat the entire procedure.
  - out of memory? So try to reduce the model and try again.

The next subsection explains some issues on properties and their formalization using temporal logic, once Model Checking requires formalized properties using LTL or CTL.

2.3.2.1 Temporal Logic and Properties

Before going into detail about temporal logic, it is important to show the meaning and notation of the logical connectives, temporal modalities (operators), and path quantifiers. Table 2.1 shows the notation and meaning of each one. Temporal modalities (operators) and path quantifiers are further discussed in more details.

The semantics of propositional logic is specified by a *satisfaction relation* $\models$ indicating the evaluations μ for which a formula Φ is true. It is written as:

$$\mu \models \Phi$$

Table 2.1 - Logical connectives, path quantifiers, and temporal modalities

Logical connective		Path quantifier		Temporal modality	
Notation	Meaning	Notation	Meaning	Notation	Meaning
$\wedge$	and	$\forall$	for all paths	$\bigcirc$	next
$\vee$	or	$\exists$	for some path	$\cup$	until
$\neg$	not			$\square$	always (globally)
				$\lozenge$	eventually

Once basic symbols and notations are exposed, some principles can be discussed.

Clarke (CLARKE, 2008) discusses that Model Checking problem is easy to state:

> Let M be a transition system (i.e., state-transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of M such that $M, s \models f$.

The term Model Checking is used because the objective is to determine if the temporal formula f was true in the structure M, i.e., whether the structure M was a model for the formula f. The term *Kripke structure* is usually used instead of transition system in honor of the logician Saul A. Kripke, who used transition systems to define the semantics of modal logics (MERZ, 2001).

The following definitions are based on Fraser (FRASER et al., 2009). Some few differences of syntax may occur when compared with other papers.

A Kripke structure M is a tuple $M = (S, S_0, T, L)$, where:

- S is a set of states.
- $S_0 \subseteq S$ is an initial state set.
- $T \subseteq S \times S$ is a total transition relation, that is, for every $s \in S$ there is a $s' \in S$ such that $(s, s') \in T$.
- $L : S \to 2^{AP}$ is a labeling function that maps each state to a set of atomic propositions that hold in this state.
  AP is a countable set of atomic propositions.

Path: a path $p := \langle s_0, s_1, ... \rangle$ of a Kripke structure M is a infinite sequence such that $\forall i \geq 0 : (s_i, s_{i+1}) \in TforM$. A set of paths of a Kripke structure M that start in state s is denoted as $Paths(M, s)$.

As infinite paths are not usable in practice, Model Checking uses finite sequences, commonly referred to as *traces*. The number of transitions in a trace is referred to as its *length*. For example, trace $t := \langle s_0, s_1, ..., s_n \rangle$ has a length of $length(t) = n$.

Temporal logic describes the ordering of events in time without introducing time explicitly. The meaning of a temporal logic formula is determined with respect to a Kripke structure. Most temporal logics have an operator like $\Box f$ that is true in the present if f is always true in the future. Temporal logic is often classified according to whether time is assumed to have a linear or a branching structure. A linear time LTL assertion h is interpreted with respect to a single *path*. An assertion of a branching time logic is interpreted over computation *trees*.

2.3.2.1.1 Linear Temporal Logic (LTL)

An LTL formula consists of atomic propositions, boolean operators and temporal operators. The operator "$\bigcirc$" refers to the *next* state. So, "$\bigcirc$ a" expresses that a has to be true in the next state. "$\cup$" is the until operator, where "$a \cup b$" means that a has to hold from the current state up to a state where b is true. "$\Box$" is the *always* operator, stating that a condition has to hold at all states of a path, and "$\Diamond$" is the *eventually* operator that requires a certain condition to eventually hold at some time in the future.

If a property ϕ is satisfied by the path π of model M, this is denoted by "$M, \pi \models \phi$" and "$M, \pi \not\models \phi$" expresses that the property is not satisfied by the path. The semantics of LTL is expressed for infinite paths of a Kripke structure. π^i denotes the suffix of the path π starting from the i-th state, and π_i denotes the i-th state of the path π. The initial state of a path π is π_0.

2.3.2.1.2 Computation Tree Logic (CTL)

CTL formulas are similar to LTL but with an extra element, path quantifiers. The path quantifiers $\forall$ (*for all*) and $\exists$ (*there exists*) require formulas to hold on all or some paths, respectively. A schematic view of the validity of $\exists\Box$, $\exists\Diamond$, $\forall\Diamond$, and $\forall\Box$ is given in Figure 2.7, where black-colored states satisfy the proposition *black*.

As all temporal operators are preceded by a path quantifier in CTL, the semantics of CTL can be expressed by satisfaction relations for state formulas. $M, s \models \phi$ denotes a state formula ϕ that is satified in state s of Kripke structure M. Most model checkers use either CTL or LTL in practice, as there are some formulas that can be only formalized in CLT, and vice-versa.

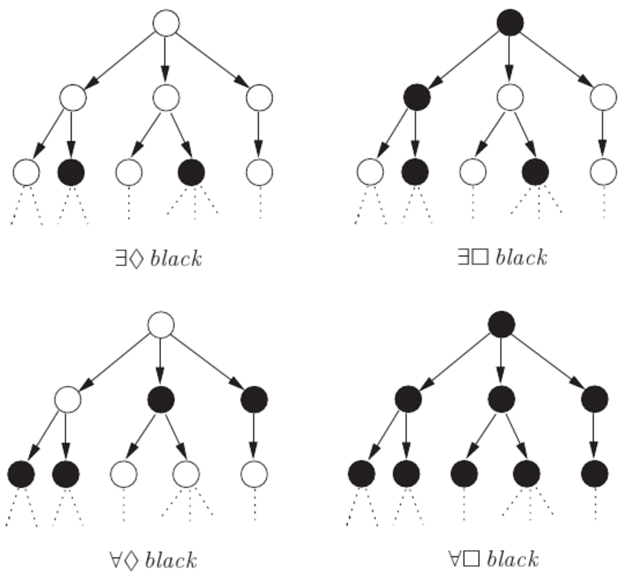


Figure 2.7 - Visualization of semantics of some basic CTL formulae. Baier and Katoen (2008)

2.3.2.1.3 Properties

Commonly, three different types of verifiable properties are distinguished:

Safety Property: A safety property describes a behavior that may not occur on any path ("Something bad may not happen"). To verify a safety property, all execution paths have to be checked exhaustively. Safety properties are of the type $\Box\neg\phi$ or $\forall\Box\neg\phi$, where ϕ is a propositional formula. Examples of safety property: *mutual exclusion property* (always at most one process is in its critical section) and *deadlock freedom*.

Invariance Property: An invariance property describes a behavior that is required to hold on all execution paths. It is logically complementary to a safety property. Invariance properties are of the type $\Box\phi$ or $\forall\Box\phi$, where ϕ is a propositional formula.

Liveness Property: A liveness property describes that "something good eventually happens". With linear time logic, this means that a certain state will always be reached. For example, $\Box\phi_1 \rightarrow \Diamond\phi_2$ and $\forall(\Box\ \phi_1 \rightarrow \forall\Diamond\ \phi_2)$ are liveness properties.

Dwyer (DWYER et al., 1999) proposed a system of property specification patterns for finite-state verification. They proposed 8 patterns and 5 pattern's 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. For instance, some descriptions of pattern/pattern scope are presented below with the correponding CTL state formulae (SANTIAGO JÚNIOR, 2011):

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

b) **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: $\forall\Box(p \rightarrow \forall\Diamond q)$;

c) **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 Patern and Globally Scope is indeed a safety property which is often characterized as "nothing bad should happen". In the above descriptions, the sentence "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" (SANTIAGO JÚNIOR, 2011).

The aim of Model Checking is to determine if a given model fulfills a property. Several different algorithms have been successfully used for this task, using different temporal logics and data structures. Once property violation or satisfaction is determined, a model checker can return an example of how this violation or satisfaction occurs. This is illustrated with a counterexample in the case of violation, or witness in the case of satisfaction (FRASER et al., 2009). Satisfaction of LTL properties is defined using linear sequences. Consequently, witnesses and counterexamples for LTL formulas are also linear sequences. In contrast, CTL properties are state formulas. Therefore, the CTL Model Checking problem is to find the set of states that satisfy a given formula in a given Kripke structure (FRASER et al., 2009). Special algorithms are used to derive trace examples for witness or counterexample states.

2.3.2.2 NuSMV

There is a wide range of available tools for applying Model Checking, for instance, SPIN (HOLZMANN, 2004), NuSMV (KESSLER, 2015), UPPAAL (BEHRMANN et al., 2004), and JavaPathfinder (NASA, 2015). NuSMV is open, flexible, and documented platform. NuSMV was previously used in version 2.0 of SOLIMVA methodology and it is also used in this PhD thesis. Following, a discussion is presented to show an overview of the NuSMV syntax.

The NuSMV language allows the description of finite state models. Finite state models consist of a set of variables and predicates on these variables. Predicates use the logical operators & (and), | (or), and ! (not). Constant 1 denotes *true* whereas 0 denotes *false*. Variables are declared using the `VAR` keyword, followed by a list of typed variable declarations. Variables can be of type boolean or can be enumerative. For example,

```
VAR
  b : boolean;
  a : { a1,a2,a3 };
```

declares a boolean variable `b` and a variable `a` that has an enumerative type, i.e., the value of `a` is either `a1, a2,` or `a3`.

There are basically two kinds of predicates: predicates defining the initial state and predicates defining the transition relation, relating the current values of some variables with their possible next values. A state is an assignment of values to a set of variables. Predicates defining the initial state are preceded by the `INIT` keyword. If there is more than one `INIT` declaration, the initial state is characterized by the conjunction of all the `INIT` predicates.

Predicates defining the transition relation can be defined using assignment declarations for each variable. Assignments are preceded by the `ASSIGN` keyword. In assignments, `next(v)` refers to the next value of variable `v`. If different next values are possible, depending upon some current condition, the case construct is used. The list of case expressions is evaluated sequentially, starting at the top. For example,

```
ASSIGN
  next(b):=
    case(b)
      !b: 1;
      1: 0;
    esac;
```

says that the next value of `b` will be true if `b` is currently false, and false otherwise.

All assignments are made concurrently, i.e., all variables change value at the same time. Two concurrent assignments to the same variable are forbidden.

The main purpose of a model checker is to verify that a model satisfies a set of desired properties specified by the user. Table 2.2 shows how temporal logic elements can be expressed in NuSMV input language.

Table 2.2 - Logical connectives, path quantifiers, and temporal modalities expressed in NuSMV notation

Logical connective		Path quantifier		Temporal modality	
Notation	Meaning	Notation	Meaning	Notation	Meaning
&	and	A	for all paths	X	next
|	or	E	for some path	U	until
!	not			G	always (globally)
				F	eventually

Basically, these are the main points related to the NuSMV syntax that are relevant to this PhD thesis.

2.3.2.3 Strengths and Weaknesses

A set of strengths and weaknesses of Model Checking have been exhibited in Baier (BAIER; KATOEN, 2008). It is shown those that are considered relevant for this thesis.

Strengths:

a) It is a general verification approach, applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.

b) It supports partial verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first.

c) It is not vulnerable to the likelihood that an defect is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.

d) It provides diagnostic information in case a property is invalidated; this is very useful for debugging purposes.

Weaknesses:

a) It suffers from the state-space explosion problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory.

b) Its usage requires some expertise in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.

In summary, Model Checking can not be applied in some areas, such as systems with a great number of components. However, it can suggest results for arbitrary parameters that may be verified using proof assistants, and therefore it has been used in a broad range of systems.

2.4 SOLIMVA 1.0 and 2.0 Methodologies

This section aims to give an overview of the SOLIMVA methodology (SANTIAGO JÚNIOR, 2011) (SANTIAGO JÚNIOR; VIJAYKUMAR, 2012) versions 1.0 and 2.0. Version 1.0 of the SOLIMVA methodology aims at the generation of model-based system and acceptance test cases considering Natural Language (NL) requirements deliverables. The 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 *Geração Automática de Casos de Teste Baseada em Statecharts* (GTSC - Automated Test Case Generation based on Statecharts) (SANTIAGO et al., 2008),(SANTIAGO JÚNIOR et al., 2012) environment is used to generate Abstract Test Cases which are later translated into Executable Test Cases.

Version 1.0 of the SOLIMVA methodology is illustrated in the activity diagram of Figure 2.8. The first activity is the definition of a Dictionary by the user/test designer. The Dictionary defines the application domain. After the definition of the Dictionary, scenarios are identified. A scenario is defined as an interaction between a user and the Implementation Under Test (IUT). Associated with each scenario there is a set of requirements which characterize such an interaction. After the previous steps, the user must select and input a set of NL requirements. The user must search these requirements in documents such as software requirements specifications.

The Dictionary does not necessarily have to be defined completely at once. This is shown as the optional activity *Update Dictionary*. Hence, the creation of the Dictionary is incremental and dependent on the selected set of NL requirements. After that, the generation of the Statechart model follows. After generating the

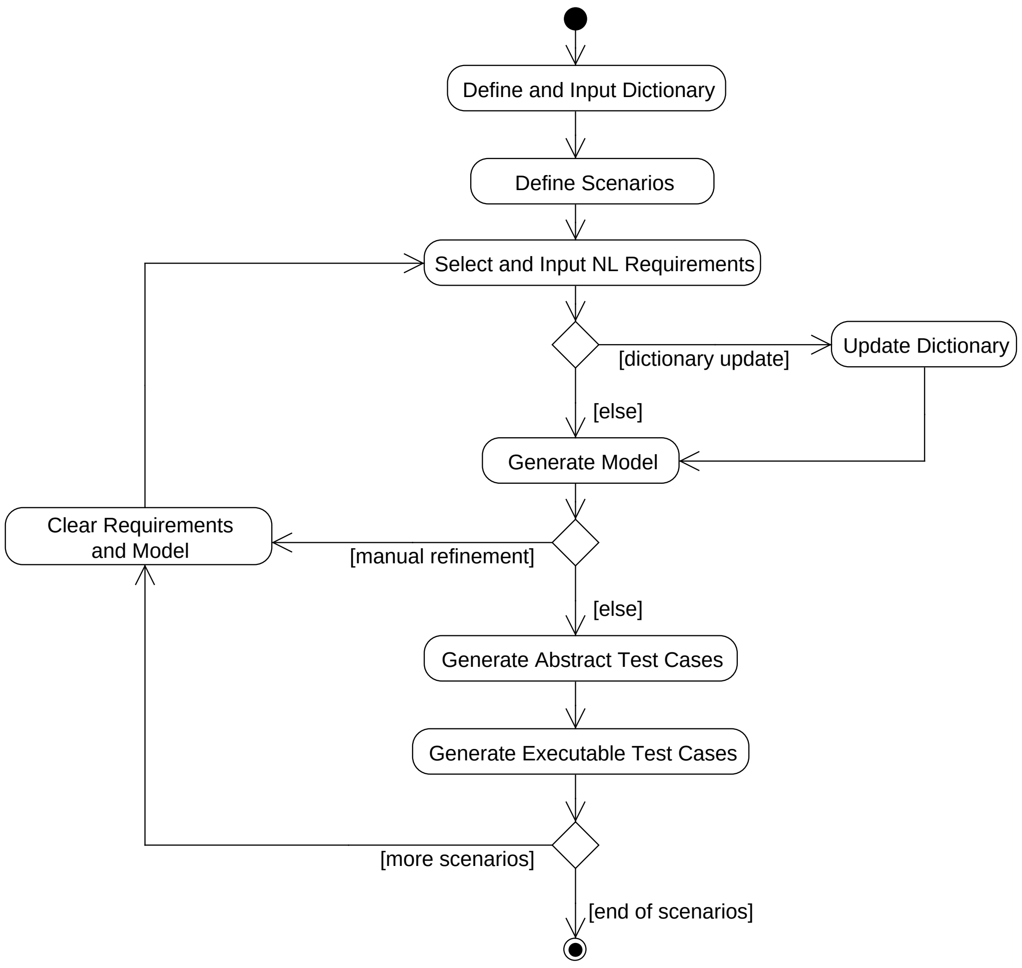

Define and Input Dictionary
Define Scenarios
Select and Input NL Requirements
[dictionary update]
Update Dictionary
[else]
Generate Model
Clear Requirements and Model
[manual refinement]
[else]
Generate Abstract Test Cases
Generate Executable Test Cases
[more scenarios]
[end of scenarios]

Figure 2.8 - Version 1.0 of SOLIMVA methodology. SANTIAGO JÚNIOR (2011)

model, the test designer may decide to manually refine it.

After these steps, Abstract Test Cases are generated by using the GTSC environment. GTSC allows test designers to model software behavior using Statecharts and/or FSMs in order to automatically generate test cases based on some test criteria for FSM and some for Statecharts. Then, 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.

Incompleteness, inconsistency, and, especially in NL requirements specifications, ambiguity are among the types of defects found in software requirements specifications. Since software requirements specifications are created early within the software development lifecycle, their defects affect the next software artifacts, including source code, to be developed.

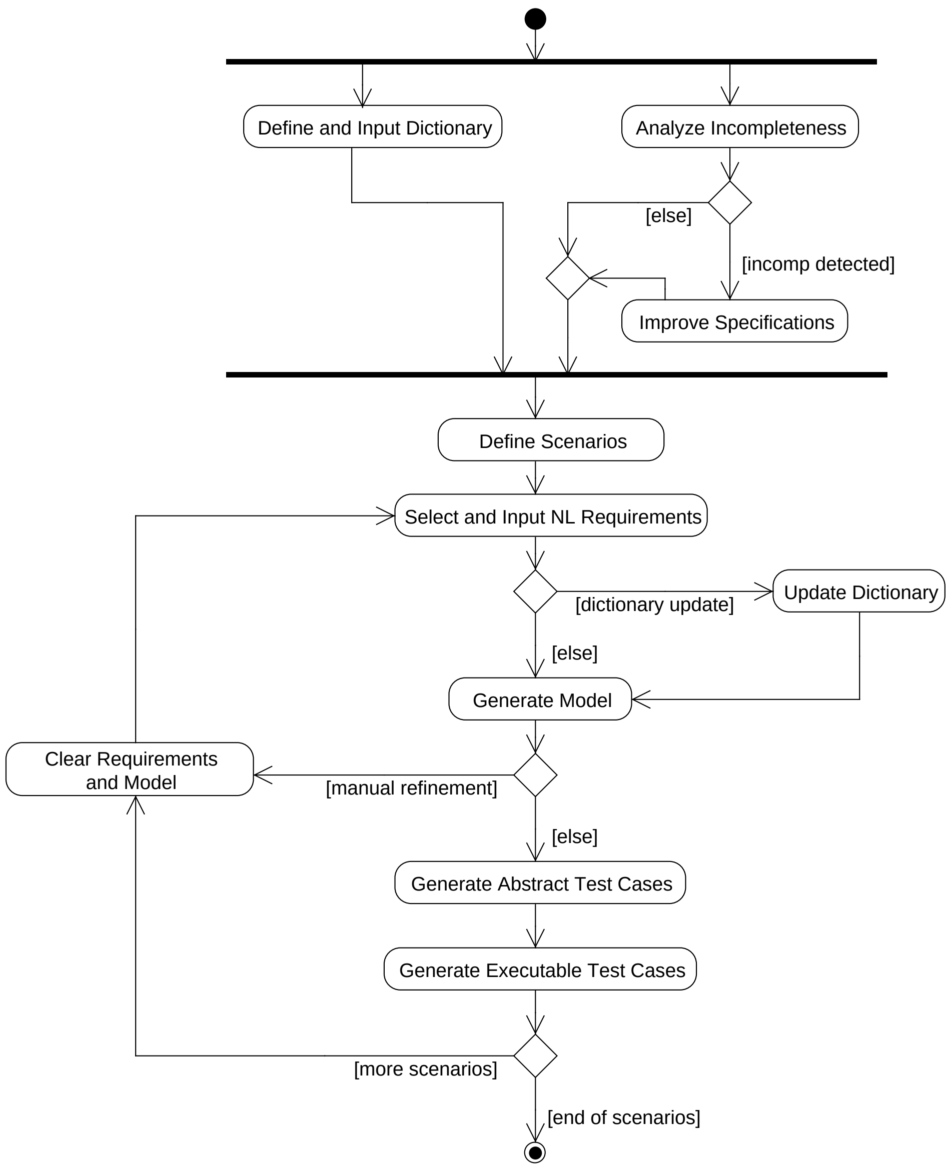


Figure 2.9 - Version 2.0 of SOLIMVA methodology. SANTIAGO JÚNIOR (2011)

Version 2.0 is an extension of version 1.0 of the SOLIMVA methodology in order to address the goal of detecting incompleteness in software specifications (SANTIAGO JÚNIOR, 2011). Model Checking combined with k-permutations of n values of variables and specification patterns (DWYER et al., 1999) were used to tackle this problem. The new version is shown in Figure 2.9. The new activities are *Analyze Incompleteness* and *Improve Specifications*. The activities present in version 1.0 are also present in version 2.0, and the workflow is essentially the same. The only difference is that the execution of these new activities should proceed in parallel with the *Define and Input Dictionary* activity.

The most important activity to deal with the problem of incompleteness is *Analyze Incompleteness*. It is by means of this activity that incompleteness defects are truly detected. It is important to realize that Model Checking, in version 2.0 of the SOLIMVA methodology, was used as a tool to aid the software inspection process (detection of incompleteness in software specifications) and not as in the more traditional approach where the finite-state model is verified against formalized properties to realize whether the behavior of the system meets its specifications. 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.

2.5 Formal Verification and UML

This section presents some of the research literature related to this PhD thesis, showing approaches that made efforts on the use of Formal Verification and UML with diverse purposes. Table 2.3 summarizes the main characteristics of the referred published literature.

Sarma and Mall (SARMA; MALL, 2009) proposed a system testing approach to cover elementary transition paths. The technique consisted on derivation of a System State Graph (SSG) based on UML 2.0 use case models, sequence diagrams, and Statechart models to design system test specifications to achieve coverage of system states and transitions among them. They aim to satisfy the test criterion *transition path coverage*. Even though it is related to Software Test instead of Formal Verification, this research was very helpful for the present work because they use the concept of scenarios and explore the relation between sequence and statechart diagrams, which was quite motivating for this PhD thesis. The importance of UML models in designing test cases is recognized in several other investigations, such as (BRIAND; LABICHE, 2002), (HARTMANN et al., 2005), (CHEN et al., 2008), (BRITO et al., 2009), (KIM et al., 1999), and (RIEBISCH et al., 2003). However, all these works used UML to somehow generate test cases. In the present research, the interest is in UML and Formal Verification.

Mikk (MIKK et al., 1998) translated Statecharts into PROMELA, the input language of the SPIN verification system. They used extended hierarchical automata as an intermediate format. The conclusion demonstrates the feasibility of LTL Model Checking for statecharts. Latella (LATELLA et al., 1999) showed a translation from a subset of UML Statechart Diagrams - covering essential aspects of both concurrent behaviour, like sequentialization, parallelism, non-determinism and priority, and state refinement - into PROMELA. Both works used Statecharts to perform the translation into PROMELA language. They use one single UML diagram, the Statechart.

In (ANDERSON et al., 1996), Anderson translates the specification of TCAS (Traffic Alert and Collision Avoidance System), which is specified in RSML (Requirements State Machine Language) into the input language of NuSMV. The objective was to investigate if Model Checking could be used in large software specifications. Dubrovin (DUBROVIN; JUNTTILA, 2008) implemented a tool that translates UML hierarchical state machine models to the input language of NuSMV too. They have defined a semantics and a compact symbolic encoding for a class of UML models which are basically hierarchical state machine. Uchitel (UCHITEL; KRAMER, 2001) proposes translation of scenarios, specified as Message Sequence Charts (MSCs), into a specification in the form of Finite Sequential Processes. This can then be fed to the Labelled Transition System Analyser model checker to support system requirements validation.

Lam (LAM, 2007) examined how activity diagrams defined in UML 2.0 standard are formally analyzed using NuSMV model checker. A model represented as activity diagrams is first transformed into NuSMV input language and then verified that a set of system specifications is satisfied using NuSMV. The objective was determining the correctness of activity diagrams.

Eshuis (ESHUIS, 2006) presented two translations from activity diagrams to the input language of NuSMV. Both translations map an activity diagram into a finite state machine (FSM) and are inspired by existing Statechart semantics. The requirements-level translation defines state machine that can be efficiently verified, but they assume the perfect synchrony hypothesis. The implementation-level translation defines state machine that cannot be verified so efficiently, but that are more realistic since they do not use the perfect synchrony hypothesis. The aim was to assess the activity diagrams from the point of view of requirements and also from the point of view of implementation, which represents the actual system behavior. These two last research works, also used one single diagram, the activity diagram.

Konrad and Cheng (KONRAD; CHENG, 2006) presented a process that supports the specification and analysis of UML models wih respect to behavioral properties specified in NL. This process has been implemented using the SPIDER tool. This approach is a Model Checking of UML models against NL properties. UML models are read and formal specification language PROMELA for the model checker SPIN is generated. NL properties are derived using a previous work (KONRAD; CHENG, 2005), where a grammar was developed. The grammar enables the NL representation of specification patterns, and it is used to formalize properties in LTL. The aim is to check UML models against the NL properties (requirements). Hence, the focus of their work is checking NL requirements. On the other hand, the present work focuses on checking UML models. They use version 1.4 of UML.

All the investigations presented so far, related to UML and Formal Verification, deal with a single UML or UML-like diagram to perform Formal Verification. Rather, the present research allows to work with up to three UML behavioral diagrams. In addition, it is not clear if in the previous studies the authors used specification patterns to formalize the properties. Specification patterns provide clear guidelines to such formalization. The present work is proposing a full approach to detect defects in the design of software developed in accordance with UML. In addition, it also was developed a tool to translate UML diagrams into a unified TS to support Model Checking.

Schäfer (SCHÄFER et al., 2001) and Knapp (KNAPP; MERZ, 2002) used two complementary UML notations for the specification of dynamic system behavior - state machine and collaborations - to automatically verify whether the interactions expressed by a collaboration can indeed be accomplished by a set of state machines. The first used the model checker SPIN to verify the model against the automata while the second used the model checker UPPAAL to perform the same task. Both works aim to validate the two diagrams, applying consistency checking between diagrams.

Calinescu (JOHNSON et al., 2013) used a probabilistic model checker (PRISM) to verify critical systems, after changes. Verifying these software systems only at design time is insufficient, they have to be reverified after each change. They did not work with UML diagrams, but with components and deterministic finite automata.

Cortellessa (CORTELLESSA; MIRANDOLA, 2002) suggested an interesting approach to encompass performance validation task as an integrated activity within the development process. They propose a methodology called Performance Incremental Validation in UML (PRIMA-UML) aimed at generating a queueing network based performance model from UML diagrams that are usually available early in the software lifecycle (use case, sequence, and deployment). Bernardi (MERSEGUER et al., 2002) translated sequence and statechart diagrams into Generalized Stochastic Petri Nets. Both works aimed analyzing performance aspects of systems. The present research is related to functional aspects of the software product, aiming at detecting design defects within the solution, but considering exclusively functional requirements of the software product.

To the best of our knowledge, only three studies consider the translation from several UML or UML-like diagrams into a formal notation to perform validation of UML diagrams, aiming functional requirements. Baresi (BARESI et al., 2011) developed MADES, a tool to carry out Formal Verification of UML-based models, mainly interested in the timing aspects of systems. It is composed by: static part (class diagrams); dynamic aspects and behavior are rendered through: (a) state diagrams and activity; (b) sequence diagrams; and (c) interaction overview diagrams, used to relate different sequence diagrams; Clocks (and time diagrams) are used to add the time dimension to systems. All these diagrams seem to be required to construct the approach. However, this assorted number of diagrams is not always available in the documentation. Adversely, this proposal requires only one diagrama as mandatory (the sequence diagram), which provides a higher chance of being used in real applications.

In (MIYAZAWA et al., 2013), a proposal of a formal semantics of SysML based on the COMPASS Modeling Language (CML) (WOODCOCK et al., 2012) notation is presented. The semantics of SysML is given as a translation from the abstract syntax of SysML to the abstract syntax of CML. They addressed the translation to CML of model that include several structural and behavioral SysML diagrams: block definition diagrams, internal block diagrams, activity diagrams, sequence diagrams, and state machine diagrams. The basic difference between the present work and theirs is that we translate the behavioral diagrams to the input language of NuSMV and not to CML. A context-free grammar that guide the derivation of the code in NuSMV was defined, with a very simple structure making it easier to undestand the translation. Also, they translate SysML diagrams, which has distinctive features from UML. Moreover, it is not clear if their approach has been applied to real and complex case studies as it was done in this PhD thesis.

Encarnación Beato (BEATO et al., 2005) presents a tool (TABU - Tool for the Active Behaviour of UML) to convert three UML diagrams into a SMV input for Formal Verification: class, state and activity diagrams. It seems to see a close approach to the present solution, using a tool for convert the XMI inputs into a SMV file. The difference is in the use of Cadence SMV as a Formal Verification tool and the need to use all the diagrams to get an output. Besides, the present research works exclusively with behavioral diagrams, while they use class diagram.

The main motivation of this approach is the practical use of formal methods in software development, through automation. This can be done throughout the life-cycle, even before software coding. Besides, the present research suggests to get a single vision of the system captured from three different diagrams (sequence, activity, and behavioral state machine). These diagrams represent complementary views of system behavior and are often used in different phases of software specification and design, allowing thus a wider system range to be verified. Most of the works mentioned deal with a specific type of UML diagram.

Another important difference of the present research related to the others is that even though there are works which use various UML diagrams, the checking is performed on individual diagrams. None of them seems to use an unified model to perform the verification, as the present research does.

A comparison between the most relevant studies related to this PhD thesis is shown in Table 2.3. The table shows the research objectives (Software Testing or Formal Verification), the type of requirements that is being analyzed in the research (functional or non-functional), and also if the research uses one single UML diagram or various UML diagrams, as well as if the diagrams are structural or behavioral.

Table 2.3 - Comparison between the most relevant research related to SOLIMVA 3.0

Research					UML			
	Software Testing	Formal Verification	Functional	Non-functional	One single diagram	Various diagrams	Structural	Behavioral
(SARMA; MALL, 2009)	X		X			X	X	X
(BRIAND; LABICHE, 2002) (HARTMANN et al., 2005) (CHEN et al., 2008) (BRITO et al., 2009) (KIM et al., 1999) (RIEBISCH et al., 2003)	X		X		X			
(MIKK et al., 1998)		X	X		X			X
(LATELLA et al., 1999)		X	X		X			X
(ANDERSON et al., 1996)		X	X		X			X
(DUBROVIN; JUNTTILA, 2008)		X	X		X			X
(UCHITEL; KRAMER, 2001)		X	X		X			X
(LAM, 2007)		X	X		X			X
(ESHUIS, 2006)		X	X		X			X
(KONRAD; CHENG, 2006) (KONRAD; CHENG, 2005)		X	X		X			X
(SCHÄFER et al., 2001)		X	X			X		X
(KNAPP; MERZ, 2002)		X	X			X		X
(CORTELLESSA; MIRANDOLA, 2002)		X		X		X	X	X
(MERSEGUER et al., 2002)		X		X		X		X
(BARESI et al., 2011)		X	X			X	X	X
(MIYAZAWA et al., 2013)		X	X			X	X	X
(BEATO et al., 2005)		X	X			X	X	X
SOLIMVA 3.0		X	X			X		X

2.6 Final Remarks

This chapter presented the theory and research related to this PhD thesis. The areas of knowledge associated with the present research include UML, Formal Verification, Model Checking, Temporal Logic, SOLIMVA 1.0 and 2.0 methodologies, among others. The chapter presented only the information necessary for understanding this thesis, as all these areas are very large and complex.

It also was presented works related to this PhD thesis, always trying to emphasize their main differences related to the present research. Table 2.3 summarizes the main characteristics of the studies. Some investigations have the objective of applying Software Testing instead of Formal Verification. This thesis keep the idea of using multiple behavioral UML diagrams, representing complementary views of the system behaviors. The goal of the present work is to let the user free to use any number of the three accepted diagrams (with at last one sequence diagram when working with more than one diagram). Most of the tools mentioned seems to work with a only diagram at time or multiple required diagrams as input.

The next chapter presents SOLIMVA 3.0 methodology, the proposed solution for Model Checking UML-based software. The activities that make up the methodology are explained. It also is detailed how the unified TS is obtained and translated to the model checker notation. For better understanding, a running example is presented.

3 APPROACH TO APPLY FORMAL VERIFICATION TO UML-BASED SOFTWARE

This chapter presents SOLIMVA 3.0 methodology aiming to address the objective stated in Chapter 1: to transform a non-formal language (UML) to a formal language (language of a model checker) in order to detect defects within the design of the software product. An abridged version of this chapter can be seen in (SANTOS et al., 2014a).

Literature review has shown the wide adoption of UML to the design and modeling of object-oriented software and that UML has received attention from researchers, as well as practitioners (SARMA; MALL, 2009). UML is popular not only for designing and documenting systems, but its importance is being recognized in providing ways to allow the application of Validation and Verification techniques (SARMA; MALL, 2009) (KNAPP; MERZ, 2002) (SCHÄFER et al., 2001) (MIKK et al., 1998) (BRITO et al., 2009) (ANDERSON et al., 1996) (DUBROVIN; JUNTTILA, 2008) (BARESI et al., 2011), among others. UML diagrams can be used to capture different views of a system, such as users', structural, behavioral, implementation, and environmental (OMG, 2011). This thesis used the UML behavioral view.

As the basis of this research, it is considered that professionals specify the functionality of the system with use cases. In the case of use cases descriptions are not available, properties are obtained from requirements described in Natural Language. Each use case is detailed (at least, it should be) by a set of UML sequence diagrams. For each use case, there should exist at least one sequence diagram that represents and describes its main scenario.

This strategy as an extension of SOLIMVA methodology is explained in the next section. The rest of the chapter is as follows: Section 3.2 describes the transformation of UML diagrams into Transition Systems. Section 3.3 shows how to obtain the NuSMV notation from the unified TS. In Section 3.4, a running example is presented for a better understanding of the methodology. Final remarks are in Section 3.5.

3.1 The SOLIMVA 3.0 Methodology

As presented in Section 2.4, version 1.0 of the SOLIMVA methodology aims at model-based test case generation considering NL requirements deliverables. Version 1.0 of the SOLIMVA methodology was later extended to address the detection of incompleteness in software specifications. This generated version 2.0 of SOLIMVA and, therefore, the new activities of version 2.0 of the methodology are more related to the automation of software inspection, specifically for detecting defects in software specifications (SANTIAGO JÚNIOR, 2011).

Testing and Formal Verification are best considered as complementary techniques. While Formal Verification techniques aim at showing that a given program/system satisfies certain formalized properties or that a proof-based approach to correctness is followed, testing aims to show that the given program is reliable in that no defects of any significance were found (MARTHUR, 2008). In this context, this research is intended to improve the existing SOLIMVA methodology to address not only software testing and inspection, but also Formal Verification in the more traditional approach.

Version 3.0 of the SOLIMVA methodology is illustrated in the activity diagram of Figure 3.1. The contribution of this PhD thesis is encompassed by the dashed lines in red. It is worth mentioning that the same activities, with the same features, present in version 2.0 of the SOLIMVA methodology are also present in version 3.0, as well as the workflow, that is the same. The difference is that it is possible to execute the new activities of version 3.0 of the SOLIMVA methodology in parallel with the older activities. In practical terms, version 3.0 of SOLIMVA proposes that the activities of testing/inspection and Formal Verification can be performed independently by different teams and even at different phases of the software development lifecycle.

Figure 3.2 shows in detail the contribution of this PhD thesis so that Formal Verification can now be addressed by SOLIMVA. In figure 3.2, activities which are shown in dashed line have been automated by the XMITS tool. XMITS is explained in the next chapter. In the following, the activities of the SOLIMVA 3.0 methodology (SANTOS et al., 2014a) are explained:

a) **Identify Scenarios**. As mentioned earlier, in this work, verification consists of sequence of scenarios to be checked. Basically, scenarios focus on how the system behaves to implement its functionalities. A scenario is a sequence of events expected during the system operation, which includes environment conditions, expected stimuli and responses (SIAU; HALPIN, 2001).

   In this work, a scenario is identified by looking at use case models. A use case can be viewed as a scenario. Each scenario is a set of related subscenarios tied together by a common goal. The mainline sequence ('main success

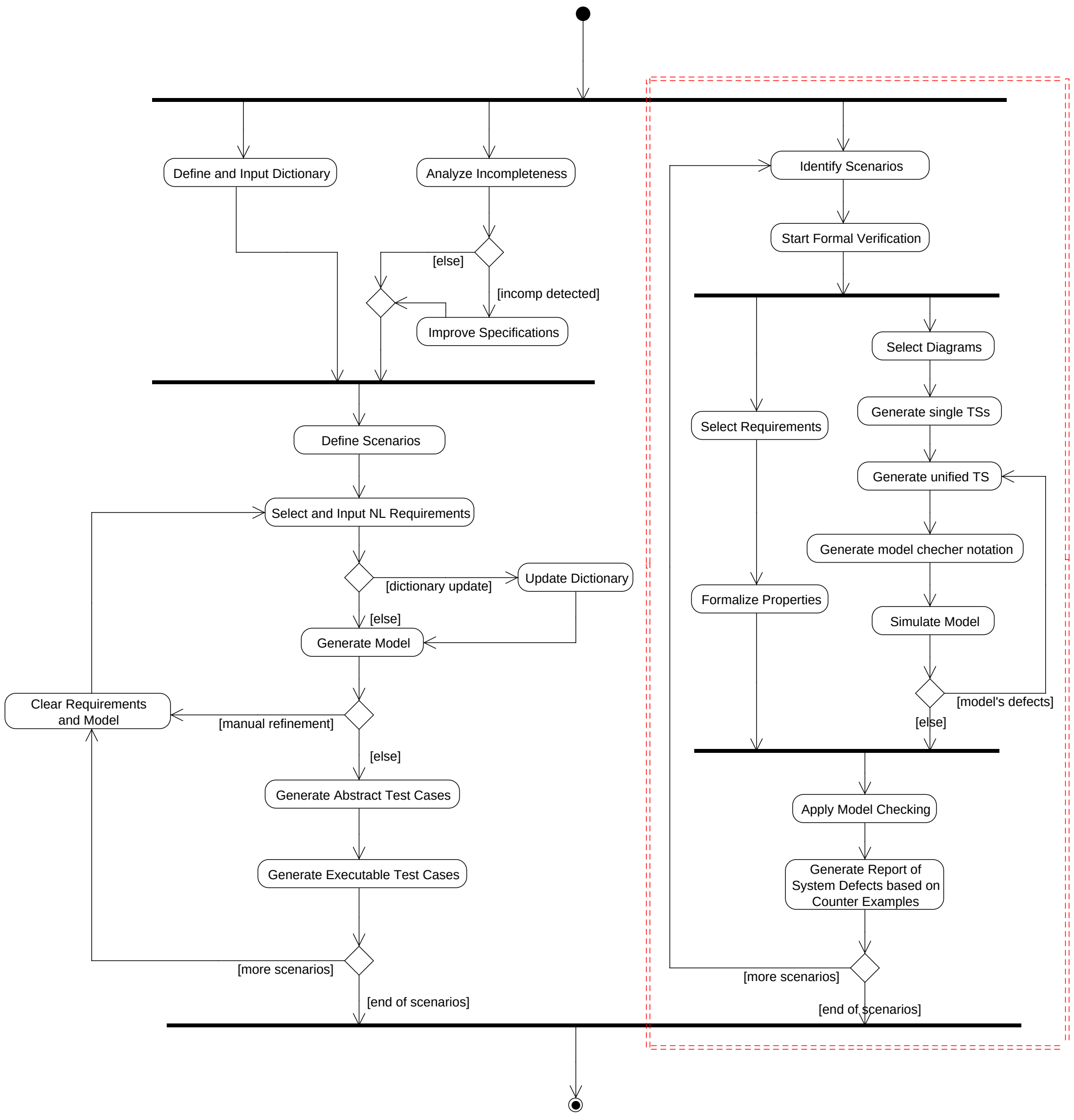


Figure 3.1 - Version 3.0 of SOLIMVA methodology

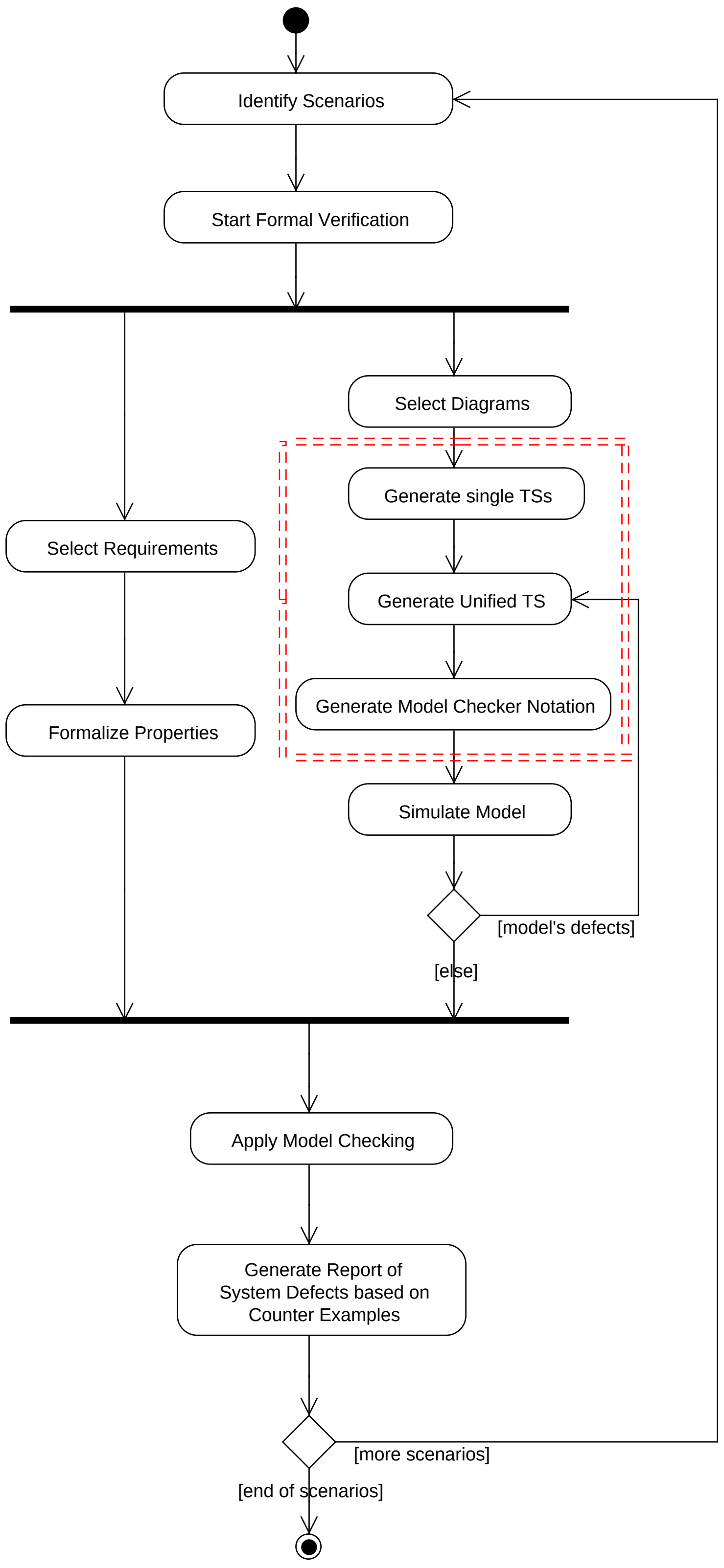


Figure 3.2 - Extending SOLIMVA: contribution of this PhD thesis

scenario' (COCKBURN, 2000)) and each of the variations ('extensions and sub-variations') are the scenarios identified by the present approach. For example, considering the classical ATM (Automated Teller Machine) system, there is a use case called *Perform Transaction* that states (BJORK, 2012):

"The session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her Personal Identification Number (PIN), and is then allowed to perform a transaction, choosing from a menu of possible types of transaction. When the customer finishes performing transactions, the card is ejected from the machine and the session ends. The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type."

In the present approach, one can consider the *Perform Transaction Use Case* as a scenario (containing the main success scenario and its variations). Observing its description, it is easy to identify its 'main success scenario': it occurs when all goes well (the card is read, the PIN is correct, the user perform the transaction with success, the card is ejected). Sub-variations may also occur, as the PIN is incorrect, the card can not be read, and so on.

b) **Start Formal Verification**. Once the scenario is identified, it is time to begin, in fact, the Formal Verification. This activity means that from this moment on, everything will be prepared to start the Formal Verification. This can be observed in the next activities, which can be executed in parallel.

c) **Select Requirements**. Here, the requirements which need to be properly executed by the system are selected. For each selected scenario, requirements are extracted from the textual description of use cases or from Natural Language descriptions. The user should identify the suitable requirements which will be verified in the system model during the Model Checking process. Considering the *Perform Transaction Use Case*, one possible requirement to be chosen is: *the customer can perform transactions only if he/she has a valid card and a valid personal identification number (PIN). Otherwise, he/she can not perform any kind of transaction.* Certainly this is an important feature that the system should perform properly, and liable to be checked.

d) **Formalize Properties**. Once the requirements are selected, it is time to formalize the properties [1]. The properties folllow the formalization proposed by means of specification patterns (DWYER et al., 1999) in LTL or CTL. To generate the properties from the available requirements, the first step is to identify the atomic propositions within the requirements, once LTL and CTL formula consists of atomic propositions, operators, and path quantifiers (this last only for CTL).

   Consider the description of a requirement in the *Perform Transaction Use Case*, presented in the previous item. It is possible to identify three atomic propositions in this description: (i) a = "the customer can perform transactions"; (ii) b = "valid card"; (iii) c = "valid personal identification number". After identifying the atomic propositions, the temporal sequence in which they occur must be analyzed, so one can verify in which scope and pattern (DWYER et al., 1999) the property fits. In this example, proposition a should happen only after propositions b and c.

e) **Select Diagrams**. The activities related to model creation, which are *Select Diagrams*, *Generate single TSs*, *Generate Unified TS*, *Generate Model Checker Notation*, and *Simulate Model*, can be executed in parallel with the *Select Requirements* and *Formalize Properties* activities. The first activity related to model creation, *Select Diagrams*, is when the respective diagrams that represent the behavior related to the use case selected are identified. Eventually, the use case selected does not have a representation in all the diagrams that this approach is intended to use. For example, a use case can be associated with a sequence and activity diagram but not with a state machine diagram; or there are only the sequence and state machine diagram for that use case but not the activity. In these cases, the model must be generated from the available diagrams for that use case. But, it is considered that **at least one sequence diagram** is available for the selected scenario. The choice for the sequence diagram occurred due to its acceptance. Most of the software specification that have been consulted presented sequence diagrams modelling the systems. As the objective is that the methodology is actually applied in practice, the choice for the sequence diagram provides greater possibility for this to happen.

[1]In the present work, requirement and property are considered synonyms

The next three activities which are shown in dashed line in Figure 3.2 are automated by XMITS tool. These activities are detailed in the next subsection.

f) **Generate Single TSs**. Based on the available UML behavioral diagrams, a single TS (finite-state model) is generated and then a unified TS **(Generate Unified TS)** is also generated. These activities along with **Generate Model Checker Notation** are performed by XMITS tool. As explained before, the present approach does not demand that all three UML behavioral diagrams (sequence, activity, behavioral state machines) exist: it is enough to have a sequence diagram to generate the TS.

g) **Generate Model Checker Notation**. The created TS is translated by XMITS into the input language of the NuSMV model checker (KESSLER, 2015). Guidelines, which take into account the NuSMV syntax and the language description discussed in Section 2.3.2.2, are detailed to accomplish the translation. This is explained in Section 3.3.

h) **Simulate Model**. The model of the system is simulated prior to Model Checking 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). If more model defects are identified then the workflow returns to the *Generate Unified TS* activity and restart from this point. When there is no more remaining defect in the model and all properties are created, Model Checking can be applied.

i) **Apply Model Checking**. Finally, Model Checking is applied to identify defects on the behavioral description of the system represented by the UML diagrams.

j) **Generate Report of System Defects based on Counterexamples**. Detected system defects are then reported. Having generated the report for one scenario, the user starts again selecting the next scenario. This process must be repeated until there is no more scenarios and the process is finalized.

In the next sections, details of the main activities of the present approach are presented, i.e., a solution to generate a single TS based on UML behavioral diagrams.

The focus is on activities **Generate Single TSs**, **Generate Unified TS**, and **Generate Model Checker Notation**, which are automated by XMITS tool. First, the single TS from each one of the diagrams is generated separately. Second, it is shown how to obtain the definitive/unified TS from the combination of the individual TSs. Finally, the generation of the NuSMV notation is explained.

3.2 Transforming UML Behavioral Diagrams into Transition Systems (TS)

This section shows how to translate scenarios which are represented in UML behavioral diagrams (by sequence, state machine, and activity) into a single Transition System, so that Formal Verification can be applied to the considered system. Once the properties to be checked and the scenario are identified, it is time to find out the UML behavioral diagrams that model these scenarios.

First, this approach achieves individuals TSs from each one of the diagrams separately. Then, the definitive/unified TS is obtained from the combination of the three individual TSs.

3.2.1 Generation of Individual TSs

This section presents the translation of individual diagrams for simple examples. For brevity, sequence diagram is denoted as SD, behavioral state machine diagram as SMD, and activity diagram as AD. It is assumed that a scenario must have at least one diagram describing it: the SD, which is mandatory, due to the explanations given in the last section.

3.2.1.1 Translating Sequence Diagrams

The first diagram to analyze is the SD. In this approach, the SD is considered to be mandatory, that is, it is taken for granted that every use case has at least one SD describing it. As will be explained in Section 3.2.2, the SD gives the directives for the combination of the diagrams, and because of this it is mandatory.

To extract the corresponding TS from an SD, it is necessary to look over the diagram starting from the initial interaction. In general, an SD has a mainline sequence and optionally several variations. The variations are usually represented using various combined fragments, like *alternatives, option, parallel,* and *loop*. Depending on the type of the fragment, different variations can be easily identified. The variations were identified from the approach presented in (SARMA; MALL, 2009), which were adapted to the present work. The rules are described below:

a) *Fragment option (opt)*: If the guard of the fragment *opt* is *true*, then all the items specified within this fragment are executed. Thus, two variations occur, one with the additional interactions of the *opt* fragment and the other without it.

b) *Fragment alternatives (alt)*: For an *alt* fragment, depending on the outcome of the guard condition for each of the operand, different variations can occur. Suppose the fragment *alt* has n number of operands, then n different possibilities would occur.

c) *Fragment loop*: For a *loop* fragment, for simplicity, we restrict that the loop be executed at most one time. That is, either the loop is not executed at all (*false* condition of the loop), or the loop is executed once (truth condition of the loop). Thus, it gives rise to at least two different possibilities.

d) *Fragment parallel (par)*: Here there is a subtle difference fom the approach proposed in (SARMA; MALL, 2009). All regions are executed to occur in parallel. Identifying all possibilities in a *par* fragment is very difficult and it leads to a large number of alternatives.

Based on these rules, by extracting all possible execution paths of a given SD, one can construct a corresponding TS. A state in the TS is composed by a message and all the guards present in the SD, along with their values for that state. By default, the TS initial state is set as *null.* Actually, the first state is the head of the list. The syntax of a state in the TS is identified by a tuple $\ll (Message)$, $g0, g1, ..., gn \gg$ where *Message* is each one of the messages present in the SD and *g0,g1,...,gn* represent all the existing guards in the SD along with their respective values. In the initial state, the guard values are assigned as '*dc*', which means 'do not care', because at the beginning the values can be either *true* or *false.* If there is parallelism, the parallel messages are combined with an '*and*'. Table 3.1 shows the elements in UML and how they are represented in the TS. Last line represents two parallel messages.

Combined fragments dictate the possible paths that an SD can reach. Transitions are based on the sequencing of exchanged messages, that is, as the sequence diagram evolves, the TS also evolves. Algorithm 1 shows how to generate a single TS from a given SD.

Table 3.1 - Translation from SD into TS

UML	Transition System	Initial Value
Message = *Register*	State = $\ll (Register), ... \gg$	*null*
Guard = *dataok*	State = $\ll ..., dataok = dc, ... \gg$	*dc*
Message1 = *Register* Message2 = *DisplayOptions*	State = $\ll (RegisterandDisplayOptions), ... \gg$	-

As will be seen, the initialization is the same for the three diagrams: first, the algorithm calls the Reader, which is the first module responsible for processing the SD. The SD is inputted as an XMI file. The operation of the modules is explained in Chapter 4. The Reader identifies if it is a valid XMI file and which type of diagram it is (SD, AD, or SMD). Then, the function *Collector.run*, which is specific for each diagram, is called to process it. The SD Collector gathers the main elements: Messages, Transitions, Combined Fragments, and Guards. These elements are then classified into six categories: State, Fork, Join, Decision, Connection, or Default. Table 3.2 shows how the elements of an SD are mapped into these generic categories.

Table 3.2 - Mapping elements of the SD into the generic categories

SD Element	Category
Initial State	Default
Message	State
Combined Fragments *opt, loop,* and *alt*	Decision
Combined Fragment *par*	Fork
Other elements	Connection

All elements in a UML diagram are classified in this way, regardless of their original diagram. Then, the Converter Logic is called to processes the lists containing the elements, classified into these categories. Based on these elements, the Converter module calls the functions. It begins from the initial state and goes on untill all states have been processed. The Converter Logic calls the Transition Function, which performs a specific function, depending on the element:

Algorithm 1: Generate the TS from a given SD

```
Input: SD
Output: single TS
1  reader.read(xml) // Call the Reader to process the XMI file
2  diagramHandler.process(reader.getOutput()) // Identify if it is a valid XMI
3  catch (Exception e) // If it is not a valid XMI, an exception is shown
4  if reader.getOutput() is a valid XMI file then
5      identifyDiagram() // Identify which diagram it is (SD, AD, or SMD)
6      type = collectTypeDiagram();
7      return type;
8      SearchSpecificCollector(type) // Call the Collector, according to the type of
          diagram (SD, AD, or SMD) to process the output of Reader
9      Collector.run(reader.getOutput());
10     if type=SD then
           // The file is iterated by specific functions to collect the main SD
              elements: Messages, Transitions, CombinedFragments, and Guards
11         MessageCollectorFunction() // The text from the messages are saved in a
              key/value data structure called Message Dictionary
12         TransitionCollectorFunction() // The system uses the Message ID to generate
              the transitions, which are stored in a Transition Dictionary
13         ElementCollectorFunction() // The combined fragments are classified and
              stored in the Element Dictionary
14         GuardCollectorFunction() // All guards are stored in the Guards Dictionary
       end
       // Then, a new processing is performed on these elements, starting with the
          first state. All collected elements are classified and saved into lists,
          according to the following categories: State, Fork, Join, Decision,
          Connection, or Default
15     Collector.classify();
       // The Converter Logic is called to process the lists of the classified
          elements
16     Converter.logic();
       // The Converter starts processing the Initial State. The Initial State calls
          the Transition Function to processes its transitions
17     repeat
18         Converter.TransitionFunction() // Call the Transition Function to process all
              the transitions of this state
19         builder.saveTransitionSystem(output) // Build the Transition System
       until all the states are processed;
       // The Transition Function processes the next state. After processed, the
          state calls the Transition Function again to process its transitions. The
          process ends when all the states are processed
   end
```

a) The Connection Function simply calls a Transition Function to process the transitions.

b) The Decision Function splits the flow process according to the decision node's guard value (true or false). It also saves the guard values in the output.

c) The Default Function does nothing. It is used, for instance, for the final node.

d) The Fork Function deals with parallel elements. It creates the parallel flow.

e) The Join Function deals with the synchronism of parallels flows after a Fork. It is responsible to terminate a parallel processing correctly.

f) The State Function is responsible for processing the states of the diagram and to create the TS, which is the output of the Converter.

While the list is being processed, the functions call an instance of Builder, an important class of the Global module, responsible for creating the Transition System output.

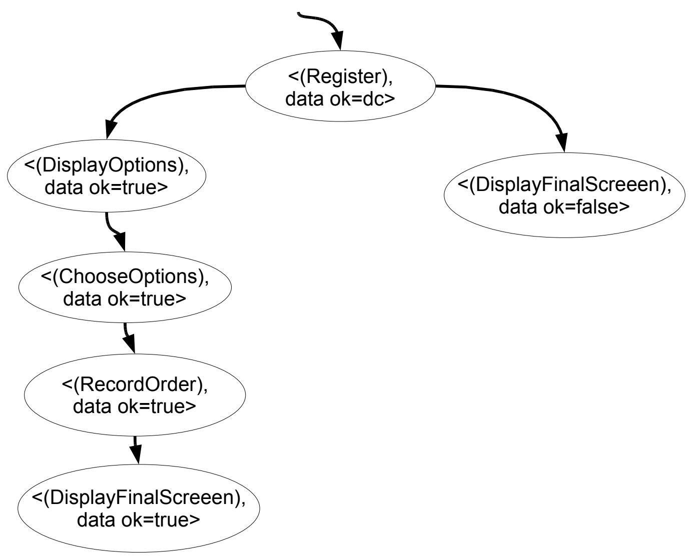
Returning to the example of the previous chapter, consider the SD shown in Figure 2.2, for the *Ordering* use case. The resulting TS for it is presented in Figure 3.3. One can observe the two possible paths reached from *opt* fragment.

3.2.1.2 Translating Activity Diagrams

AD is not compulsory in the present approach to obtain the unified TS. AD focuses on representing activities which may or may not correspond to methods of classes, so they can model the behavior of a class as well of a system. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transition, then these must be identified through conditions. AD's typically support description of sequencing, conditional dependency, parallel activities, and synchronization aspects involved in different activities.

To build the corresponding TS, each activity and transitions are analyzed. To translate the transition behavior of the activities some specific nodes are identified:

a) *Fork node*: it separates a transition in several other transitions that are executed at the same time (parallel): there are n possible variations.



<(Register), data ok=dc>
<(DisplayOptions), data ok=true>
<(DisplayFinalScreeen), data ok=false>
<(ChooseOptions), data ok=true>
<(RecordOrder), data ok=true>
<(DisplayFinalScreeen), data ok=true>

Figure 3.3 - TS generated from the SD of Figure 2.2

b) *Join node*: it is a synchronization. The next state is activated only when all the transitions arrive in this node.

c) *Merge node*: junction of transitions. Every transition that arrives, passes by this node and the next state is activated.

d) *Decision node*: depending on the condition, it shows different transitions. Two possible paths can occur here. Guards are present in decision nodes.

Activities in AD are mapped as states in the TS. As in SD, guards compose a state in the TS, along with its value for that state. Also, parallel activities are represented with an '*and*'. By default, the TS initial state is set as *null*. Actually, the first state is the head of the list. The syntax of a state in the TS is identified by a tuple $\ll (Activity), g0, g1, ..., gn \gg$ where *Activity* is each one of the activities present in the AD and *g0,g1,...,gn* represent all the existing guards in the AD along with their respective values. In the same way, in the initial state, guard values are assigned as '*dc*'. Note that guard values are also being used in the states to be coherent with the approach proposed for the SD. This will be valuable when combining the diagrams, as will be seen in Section 3.2.2. Table 3.3 shows the elements in UML and how they are represented in the TS. Last line represents two parallel activities.

Based on these rules, one can construct a corresponding TS for a given AD. Algorithm 2 shows how to generate a single TS from a given AD. It is very similar to

Table 3.3 - Translation from AD into TS

UML	Transition System	Initial Value
Activity = $Showproductslist$	State = $\ll (Showproductslist), ... \gg$	$null$
Guard = $dataok$	State = $\ll ..., dataok = dc, ... \gg$	dc
Activity1 = $Askcustomerdata$ Activity2 = $Showproductslist$	State = $\ll (Askcustomerdata$ $andShowproductslist), ... \gg$	-

the algorithm presented to convert the SD. The main difference is the Collector, which classifies the elements that are specific to AD: nodes, edges, or guards. Then, the behavior is the same of Algorithm 1: the elements are classified into categories, saved in lists, and these lists are processed to generate the TS. Table 3.4 shows how the elements of an AD are mapped into the generic categories.

Table 3.4 - Mapping elements of the AD into the generic categories

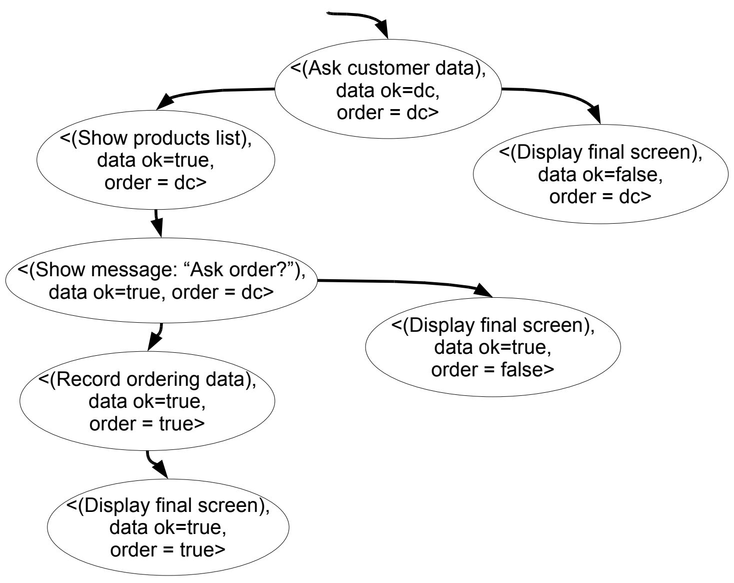
SD Element	Category
Initial Node	Default
Activity	State
Decision Node	Decision
Fork Node	Fork
Join Node	Join
Merge Node	Connection
Other elements	Default

Figure 3.4 shows the corresponding TS for the activity diagram presented in Figure 2.3 of previous chapter. This AD is modeling the same case study (*Ordering*) previously presented for the SD. It is possible to observe that each branch in the AD

leads to different paths in the TS.

Algorithm 2: Generate the TS from a given AD

```
Input: AD
Output: single TS
1  reader.read(xml) // Call the Reader to process the XMI file
2  diagramHandler.process(reader.getOutput()) // Identify if it is a valid XMI
3  catch (Exception e) // If it is not a valid XMI, an exception is shown
4  if reader.getOutput() is a valid XMI file then
5      identifyDiagram() // Identify which diagram it is (SD, AD, or SMD)
6      type = collectTypeDiagram();
7      return type;
8      SearchSpecificCollector(type) // Call the Collector, according to the type of
          diagram (SD, AD, or SMD) to process the output of Reader
9      Collector.run(reader.getOutput());
10     if type=AD then
           // The file is iterated by specific functions to collect the main AD
              elements: Nodes, Edges, or Guards
11         NodeCollectorFunction() // All the elements classified as Node are stored in
              the Element Dictionary
12         EdgeCollectorFunction() // All the Edges are classified as Transition and
              stored in the Transition Dictionary
13         GuardCollectorFunction() // The function collects all guard conditions and
              associates them with is specific decision structure. All the guards are
              stored in the Guards Dictionary
       // Then, a new processing is performed on these elements, starting with the
          first state. All collected elements are classified and saved into lists,
          according to the following categories: State, Fork, Join, Decision,
          Connection, or Default
14     Collector.classify();
       // Now, the Converter Logic is called to process the lists of the classified
          elements
15     Converter.logic();
       // The Converter starts processing the Initial State. The Initial State calls
          the Transition Function to processes its transitions
16     repeat
17         Converter.TransitionFunction() // Call the Transition Function to process all
              the transitions of this state
18         builder.saveTransitionSystem(output) // Build the Transition System
       until all the states are processed;
       // The Transition Function processes the next state. After processed, the
          state calls the Transition Function again to process its transitions. The
          process ends when all the states are processed
```



<(Ask customer data), data ok=dc, order = dc>
<(Show products list), data ok=true, order = dc>
<(Display final screen), data ok=false, order = dc>
<(Show message: "Ask order?"), data ok=true, order = dc>
<(Display final screen), data ok=true, order = false>
<(Record ordering data), data ok=true, order = true>
<(Display final screen), data ok=true, order = true>

Figure 3.4 - TS generated from the AD of Figure 2.3

3.2.1.3 Translating State Machine Diagrams

The SMD, as the AD, is not mandatory either. An SMD is composed of nodes and transitions and each one of them is analyzed to build the corresponding TS. In the following, it is explained how the nodes are analyzed.

a) *Simple node*: it is mapped as a simple state in the TS.

b) *Composite node*: consists of one or more regions. A region is a container for sub-states. A composite state can either be sequential or concurrent. In a sequential type of composite state, the state is considered to be an *exclusive-or* of its sub-states. That is, a composite state can be in any of its sub-states, but not in more than one sub-state at any time. But, in a concurrent type, the state is determined by a logical *and* of its sub-states and the object is considered to be in all the concurrent states at the same time.

c) *Choice node*: it is equivalent to *if/else*. So, here two situations can occur: one when the guard condition is set to *true* and the other when it is set to *false*. So, there are two paths to reach.

d) *Join node*: it represents a synchronization where the next state occurs only when all the arrows arrive on the join node.

e) *Fork node*: the node may have one or more arrows from the node to states. Suppose there are n number of arrows, so n possibilities would occur.

f) *Junction node*: junction of transitions. Every transition that arrives, passes by this node and the next state is activated.

Based on these rules, one can construct a corresponding TS for a given SMD. There is a difference in the syntax when constructing the TS, related to SD and AD. As in SMD one state can have several incoming events, a state in the SMD is mapped as states in the TS plus the event which triggered this state. Besides, guard values for that state are mapped in the same way that they are mapped in SD and AD. States in SMD can be single or parallel (represented as '*and*').

The syntax of a state in the TS is identified by a tuple $\ll (Event - State), g0, g1, ..., gn \gg$ where *Event* is each one of the events present in the SMD, *State* is each one of the states present in the SMD, and *g0,g1,...,gn* represent all the existing guards in the SMD along with their respective values. Also, in the initial state, guard values are assigned as '*dc*' and the TS initial state is set as *null* (list head). Table 3.5 shows the elements in UML and how they are represented in the TS. Note that in the first line, the initial state has no event, so, a dash is put in place of what would be the name of the event. This is repeated every time there is no name for the event which triggered the state. Line two shows the syntax with event.

Table 3.5 - Translation from SMD into TS

UML	Transition System	Initial Value
Initial State = $Idle$	State = $\ll (- - Idle), ... \gg$	$null$

Event = $DataEntry$
State = $ValidatingData$
	State = $\ll (DataEntry-$
	$ValidatingData), ... \gg$
		-

Guard = $dataok$	State = $\ll ..., dataok = dc, ... \gg$	dc

State1 = $Idle$
State2 = $ValidatingData$
	State = $\ll (-Idleand$
	$-ValidatingData), ... \gg$
		-

Algorithm 3 shows how to build a TS from an SMD. It has the same behavior of the last two algorithms presented. The difference is also in the Collector. Here, the SMD is divided as: Elements (which can be State, Pseudo State, or Final State) and Transitions. Table 3.6 shows how the elements of an SMD are mapped into the generic categories.

Algorithm 3: Generate the TS from a given SMD

```
Input: SMD
Output: single TS
reader.read(xml) // Call the Reader to process the XMI file
diagramHandler.process(reader.getOutput()) // Identify if it is a valid XMI
catch (Exception e) // If it is not a valid XMI, an exception is shown
if reader.getOutput() is a valid XMI file then
    identifyDiagram() // Identify which diagram it is (SD, AD, or SMD)
    type = collectTypeDiagram();
    return type;
    SearchSpecificCollector(type) // Call the Collector, according to the type of
        diagram (SD, AD, or SMD) to process the output of Reader
    Collector.run(reader.getOutput());
    if type=SMD then
        // The file is iterated by specific functions to collect the main SMD
            elements: Elements and Transitions
        ElementCollectorFunction() // All elements classified as State, Pseudo State
            or Final State
        TransitionCollectorFunction() // All transitions are stored in the Transition
            Dictionary
    // Then, a new processing is performed on these elements, starting with the
        first state. All collected elements are classified and saved into lists,
        according to the following categories: State, Fork, Join, Decision,
        Connection, or Default
    Collector.classify();
    // Now, the Converter Logic is called to process the lists of the classified
        elements
    Converter.logic();
    // The Converter starts processing the Initial State. The Initial State calls
        the Transition Function to processes its transitions
    repeat
        Converter.TransitionFunction() // Call the Transition Function to process all
            the transitions of this state
        builder.saveTransitionSystem(output) // Build the Transition System
    until all the states are processed;
    // The Transition Function processes the next state. After processed, the
        state calls the Transition Function again to process its transitions. The
        process ends when all the states are processed
```

Figure 3.5 shows the corresponding TS for the state machine diagram presented in Figure 2.4. Also, the SMD is modeling the same case study (*Ordering*). It is possible to observe the different paths extracted from the events.

Table 3.6 - Mapping elements of the SMD into the generic categories

SD Element	Category
State	State
Choice Node	Decision
Fork Node	Fork
Join Node	Join
Junction Node	Connection
Terminate Node	Default
Other elements	Default

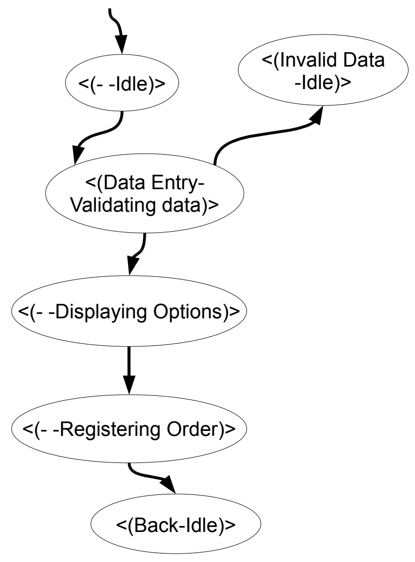


Figure 3.5 - TS generated from the SMD of Figure 2.4

Now it is presented how a unified TS can be achieved from each one of the individual TSs obtained from the UML diagrams.

3.2.2 The Unified Transition System

In this subsection, it is shown how to generate the final/unified TS. Some definitions and notations used in this approach are given in the sequence. As previously mentioned, it is assumed that a scenario must have at least one diagram describing it: the SD. In the following the syntax of the unified TS is detailed.

A state in the unified TS is identified by a tuple $\ll (Message, Activity, Event - State), g0, g1, ..., gn \gg$ where *Message* is from SD; *Activity* from AD; and *Event-State* is from SMD; *g0,g1,...,gn* represent all the existing guards in all the diagrams along with their respective values. At the beginning, all guard values are assigned to '*dc*' for the same reasons presented before, for creating the individual TSs. If there are parallel messages, states, or activities, an '*and*' is inserted in the tuple, such as $\ll (Msg1andMsg2, Activity1andActivity2, Event1State1andEvent2State2), ... \gg$, which means that *Msg1* and *Msg2* are sent in parallel, as well as *Activity1* and *Activity2*, and *State1* and *State2* are parallel states.

There are situations that should be considered. They were divided into two cases:

a) The first one is when a diagram is missing, that is, the scenario that is being analyzed is not represented in all the three diagrams. It is possible that the AD is missing; or the SMD is missing; or both, the AD and SMD are missing. The part representing the diagram that is missing in the corresponding TS is substituted by an underscore '_'. In these situations, states are described as $\ll (Message, Activity, -), g0, g1, ..., gn \gg$ or $\ll (Message, -, Event - State), g0, g1, ..., gn \gg$, or $\ll (Message, -, -), g0, g1, ..., gn \gg$.

b) The second one happens because the diagrams do not model the same behavior all the time. Some situations can be more detailed in an SMD than in an SD diagram, or does not even appear in the SD diagram. When this happens, diagrams are incoherent or inconsistent. In these situations, states in the unified TS are described as $\ll (Message, Activity, -), g0, g1, ..., gn \gg$ or $\ll (Message, -, Event - State), g0, g1, ..., gn \gg$, or $\ll (-, Activity, Event-State), g0, g1, ..., gn \gg$, and all its variations. Note that in this case, it is possible that the SD does not contain a behavior which can be more detailed in the AD. So, the case $\ll (-, Activity, Event - State), g0, g1, ..., gn \gg$ may happen. That does not mean that the SD is missing. Simply, that particular behavior is not modeled on the SD.

The main challenge to combine the three diagrams is to define the correspondence among messages, events, and activities. For instance, an event for a transition of the SMD can be a message of the SD, while the message can be a call to an activity modeled by an AD. Nevertheless, finding where a specific message of the SD is modeled in the SMD or in the AD is very hard. Instead, the final TS is built using the individual TSs. The algorithm looks over the TSs starting from their initial states. The correspondence among states of each TS is assigned by using the flow of transitions. At each iteration, the algorithm parses the three TSs and construct the new states by taking the next possible transitions in each TS.

The flow of transitions, alone, is not enough to construct the final TS, as each TS contains multiple paths. In addition, guard values are used to help finding the correspondence among states. As already stated, the SD is mandatory and conducts the diagrams merging. The basic elements that delineate the possibility of paths in the SDs are guards on the combined fragments. Thus, guard values are adopted along with the flow of transitions to construct the final TS states. If one thinks of an algorithm, at each iteration, the algorithm seeks the next possible transitions in each TS and their guard values.

To find out states that have the same guard values, the *gvs* (guard value structure) was created. The *gvs* represents the values of each one of all available guards on each state. Every time, for each possible next transitions of the TSs created from the SD, SMD, and AD, their *gvs* are verified and a match of their values is created. Thereafter, new states are generated.

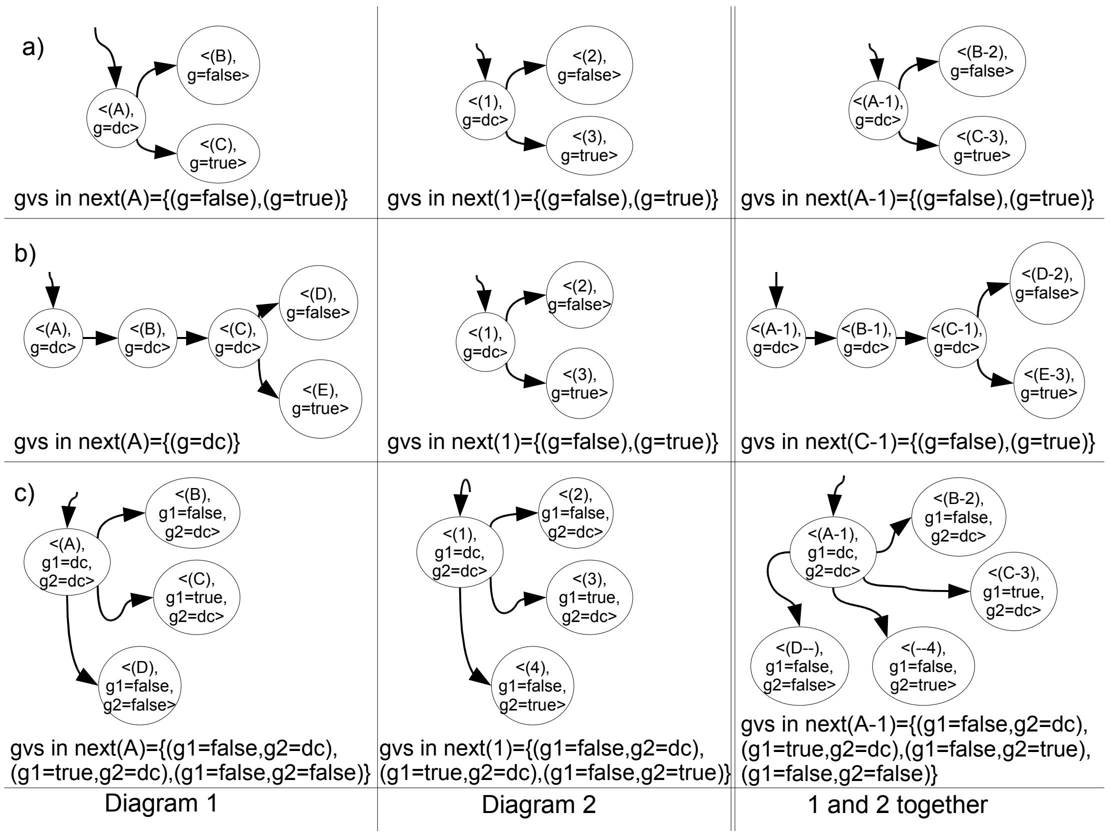


Figure 3.6 - Possible situations to generate the unified TS and its respective gvs

Figure 3.6 shows the three possible situations that may occur to generate a state in the final TS. All other cases found end up falling in one of these three cases. Rows are representing the three cases **a, b,** and **c**. The TS in column 3 is the combination of the TSs in columns 1 and 2.

Situation **a)** is straightforward to see. All the *gvs* match and the combination is explicit. Situation **b)** shows how the algorithm handles cases when in at least one of the next possible states, the value of its *gvs* is the same of the current *gvs* value. This occurs because the diagrams do not model the same behavior during their evolution. In this case, the TSs have different lengths. Here, only the TS which has the value of its next *gvs* equal to the current evolves. The other TS keeps standing until the next *gvs* match, or the TS processing complete. Situation **c)** shows an example of inconsistent states, due to inconsistent diagrams. As previously discussed, this may happen when one of the diagrams models a behavior that is not modeled in the other diagram and vice versa. New transitions are created, as many as necessary to cover all possibilities of *gvs* values.

The three situations presented in Figure 3.6 represent the rules to unify the diagrams based on the *gvs*. These rules were implemented in XMITS as an array list. They were defined as “Rules Dictionary”. The Rules Dictionary is an array containing four rules: First Rule, Second Rule, and Third Rule, which represent, respectively, situations a, b, and c of Figure 3.6; and Fourth Rule, which is applied when there are no next states, that is, when the diagram was all consumed. In this case, the diagram is taken out from the processing. When building the unified TS, these rules are always called in the order in which they are held in the array. Figure 3.7 shows the structure of the Rules Dictionary.

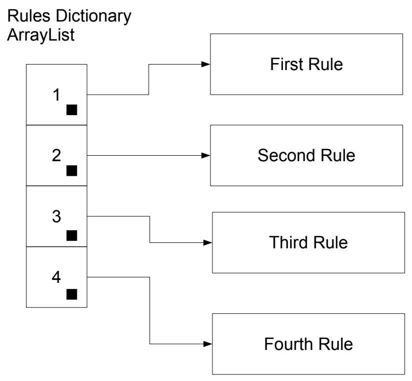


Figure 3.7 - Details of the Rules Dictionary

To perform this approach, before combining the TSs, three situations must be considered:

a) **TSs with no guards**. All the TSs that are being combined have no guards. For example, a sequence diagram with no combined fragments, or only with parallel combined fragment. All other combined fragments require guards. Or an activity diagram with no decision node. Or a behavioral state machine diagram with different incoming events. In the absence of guards, it is quite easy to construct the final TS: all that is required is to follow the flow of transitions. At each iteration, all the next possible states are combined. Note that it is possible to have multiple paths, and that all the combinations must be done, as can be seen in Figure 3.8.

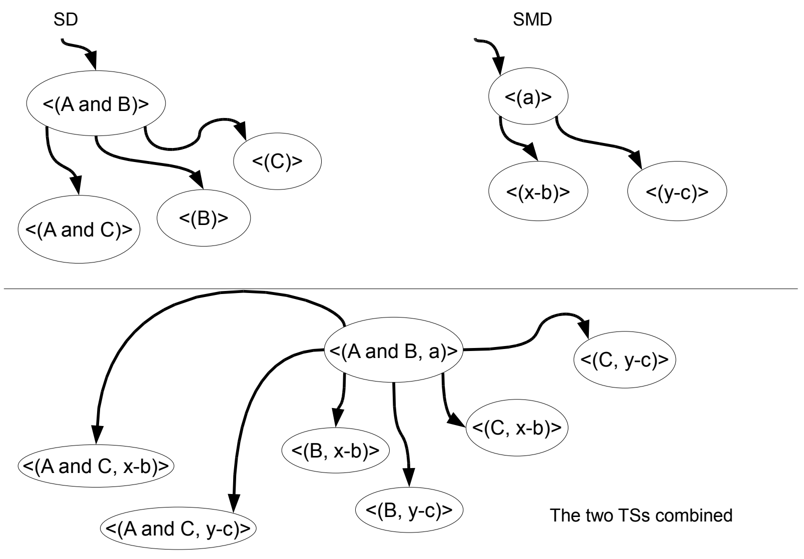


Figure 3.8 - Combining TSs with no guards

b) **One TS has guards and the other one does not**. At the beginning, the algorithm seeks all guards in all diagrams. Suppose TS1 has guards *g1* and *g2* and TS2 has no guards. The *gvs* will be composed by $\{g1, g2\}$ and it is assigned to TS2. As originally TS2 did not contain {g1,g2}, their values are assigned as "*dc*" and will never change. Thenceforth the situations are covered in Figure 3.6. Figure 3.9 shows how the approach handle this situation.

c) **TSs with different guards**. It is similar to the previous case. At the

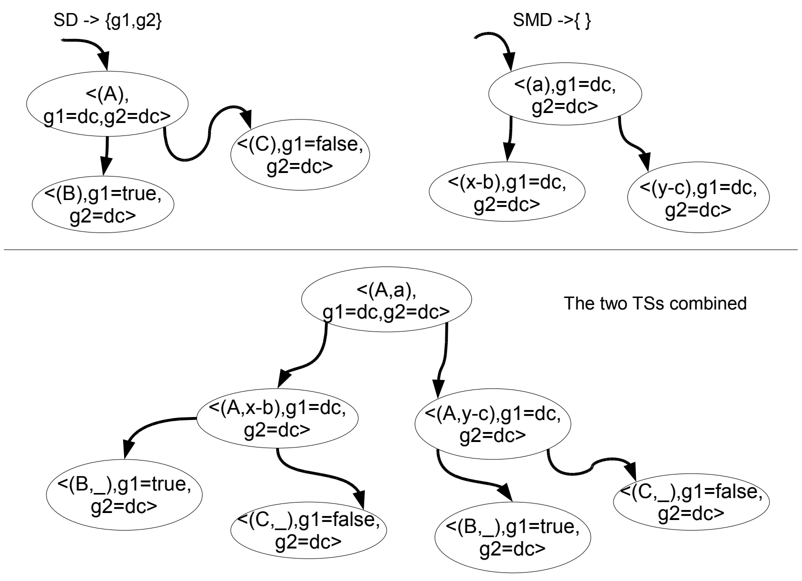


Figure 3.9 - Combining TSs when one TS has guards and the other one does not

beginning, the algorithm seeks all guards in all diagrams. Suppose TS1 has guards *g1* and *g2* and TS2 has guard *g3*. The *gvs* will be composed by $\{g1, g2, g3\}$ and it is assigned to TS1 and TS2. As originally TS2 did not contain *g1,g2*, their values are assigned as "*dc*" and will never change, as well as TS1 will receive *g3*, which is assigned as "*dc*" and will never change. Thenceforth the situations are covered in Figure 3.6. Figure 3.10 shows how the approach handles this situation.

Algorithm 4 shows how to build the unified TS, applying all the guidelines that have been explained so far. The single TSs are the input, which must contain at least one element generated from an SD diagram. The TSs are saved in the TSDictionary (line 2). At each iteration, the Repeat loop tries apply one of the four rules to the diagrams (lines 5 to 16), until all states of all diagrams have been analized:

a) The First Rule looks for equal next states in all diagrams. If, and only if, all the states from all diagrams are equal at the same time, it is possible to create a unified state. If all the next states are equal, the first rule will iterate all diagrams and create a unified state with those states, saving it in the Builder class (lines 15-16).

b) The Second Rule verifies all next states. If at least one diagram does not change its *gvs* in the next state, only this diagram is iterated and then the

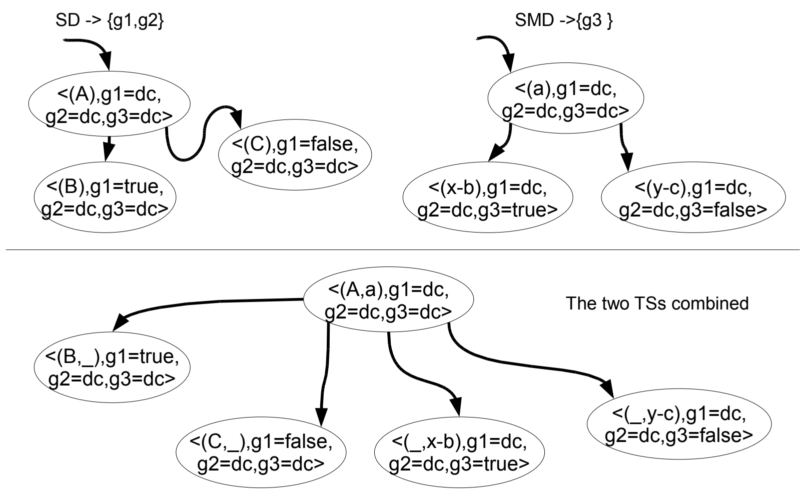

SD -> {g1,g2}
<(A),g1=dc, g2=dc,g3=dc>
<(B),g1=true, g2=dc,g3=dc>
<(C),g1=false, g2=dc,g3=dc>
SMD ->{g3 }
<(a),g1=dc, g2=dc,g3=dc>
<(x-b),g1=dc, g2=dc,g3=true>
<(y-c),g1=dc, g2=dc,g3=false>
<(A,a),g1=dc, g2=dc,g3=dc>
The two TSs combined
<(B,_),g1=true, g2=dc,g3=dc>
<(C,_),g1=false, g2=dc,g3=dc>
<(_,x-b),g1=dc, g2=dc,g3=true>
<(_,y-c),g1=dc, g2=dc,g3=false>

Figure 3.10 - Combining TSs with different guards

First Rule is applied. Note that if there are more diagrams in the same situation, all these diagrams must be iterated.

c) If there are at least two equal next states, the First Rule is applied to the equal states. Otherwise, if all the next states are different, there is no way to create a unified state. If it happens, the diagrams start to be processed individually, no unified state will be created.

d) If there is no more next states, then the last rule is applied. If a diagram has no next state, the last rule takes the diagram out from the loop.

The last line of the algorithm, line 17 shows the unified TS. The next section shows how to obtain the model checker notation from the TS generated.

3.3 Generation of Model Checker Notation

After its creation, the unified TS can be used to systematically generate its corresponding encoding into the model checker input language by constructing declarative divisions (DEBBABI et al., 2010). It is important to emphasize that once a formal unified TS is generated from UML behavioral diagrams, there is the possibility of transforming it into several different languages of available model checkers such as SPIN (HOLZMANN, 2004) and NuSMV (KESSLER, 2015).

In the current approach, the NuSMV model checker was chosen because it is open source, it has a widespread use in academia, and it accepts properties formalized not only in Computation Tree Logic (CTL) but also in Linear Temporal Logic (LTL) (BAIER; KATOEN, 2008). These two logics are well known and have mapping defined in the specification patterns (DWYER et al., 1999).

Algorithm 4: Generate the unifed TS

```
Input: singleTSs
// where singleTSs is defined as follows:
// Given a set Y = { { }, {TS generated from an SD }, {TS generated from an AD },
   {TS generated from an SMD }}
// singleTSs = {TS generated from an SD} ∪ {X}, where
// X = {x| x is "an Arrangement with Repetition" of the elements of Y}
Output: unified TS
1  foreach element ∈ singleTSs do
2      TSDictionary.addTransitionSystem(TransitionSystem) // Save the single TSs in the
          TSDictionary
   // To proceed, the system verifies if at least one single TS is generated from an
      SD
3  if ∃ element ∈ singleTSs | element = {TS generated from an SD} then
       // The loop to unify the diagrams starts
4      forall TSs ∈ TSDictionary do
5          repeat
               // Starting from initial states of all TSs, analyze all the states
6              startRace(TransitionSystem,List<Lane>);
               // The system iterates over a rule list and tries to apply the rules
                  in a specific order to the diagrams to find a way to unify the
                  states
7              tuts.logics.rules.FirstRule;
8              tuts.logics.rules.SecondRule;
9              tuts.logics.rules.ThirdRule ;
10             tuts.logics.rules.FourthRule ;
11             for rule1 to rule4 do
                   // If the rule apply, the diagrams are iterated to the next state.
                      If the rule do not apply it tries the next rule
12                 if rule is true then
13                     apply rule;
14                     break;
15             if rule=FirstRule then
16                 builder.saveTransitionSystem(output) // Build the Transition System
           until all states in all TSs have been analyzed;
17     tuts.getOtput() // This command shows the unified TS
```

Considering the NuSMV model checker, declaration of variables is relatively easy, as presented in Subsection 2.3.2.2. One variable called *State* is related to the element ≪*(Message,Activity,Event-State)*≫ of the tuple that identifies a certain state of the TS. In addition, there will be as many variables as the guards identified within the UML behavioral diagrams, i.e. $g0$ is transformed into a variable $vg0$ of enumerative type, $g1$ into a variable $vg1$, and so on. All these variables derived from the guards will be enumerated with the following values: $\{dc, false, true\}$. The dc value is important because in many occasions and especially at the beginning of the behavioral system modeling, the values of certain guards either do not care or do not make sense be true or false. Another remark is that the use of only state variables of NuSMV.

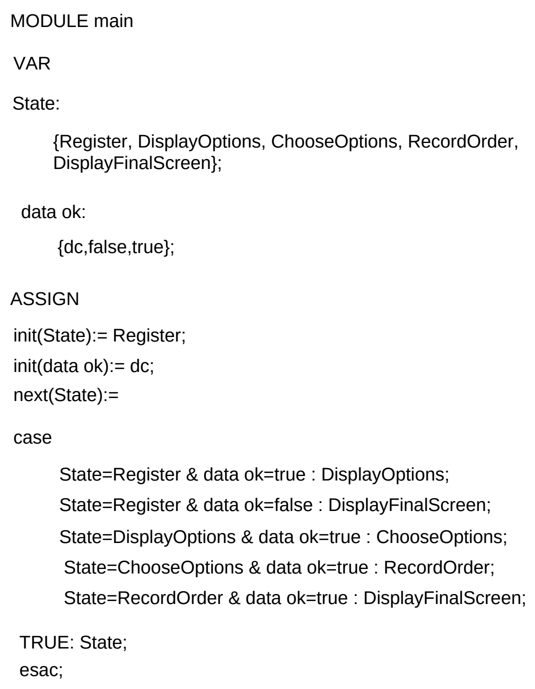
Table 3.7 shows the elements in the TS and how they are represented in the NuSMV. States are grouped into enumerations, as well as the guards. Each transition is represented by the CASE construct.

Table 3.7 - Translation from the unified TS into NuSMV

Unified TS	NuSMV	Initial Value
State	Variable State of Enumerative type	
Guard	Variable of Enumerative type	dc
Transition	CASE construct	

The initial value of the *State* variable of the final TS in NuSMV is always initialized with the initial state to mark the starting point of the diagram. Note that this is not the “full” state characterization of the initial state of the NuSMV model because we depend on the values of the variables representing the guards to know this. To generate the model in NuSMV we simply follow the transitions of the unified TS, making appropriate assignments to the *State* variable and for each variable derived from the guards. To illustrate the proposed translation, consider the TS of Figure 3.3. Applying the proposed guidelines, one can obtain the code presented in Figure 3.11.

To automate this activity (**Generate Model Checker Notation**), a grammar was developed. At first, the process to translate a TS into NuSMV imposes no restriction. Name of messages in sequence diagrams, activities in activity diagrams, and events and states in state machine diagrams were not adequate to become values of variables



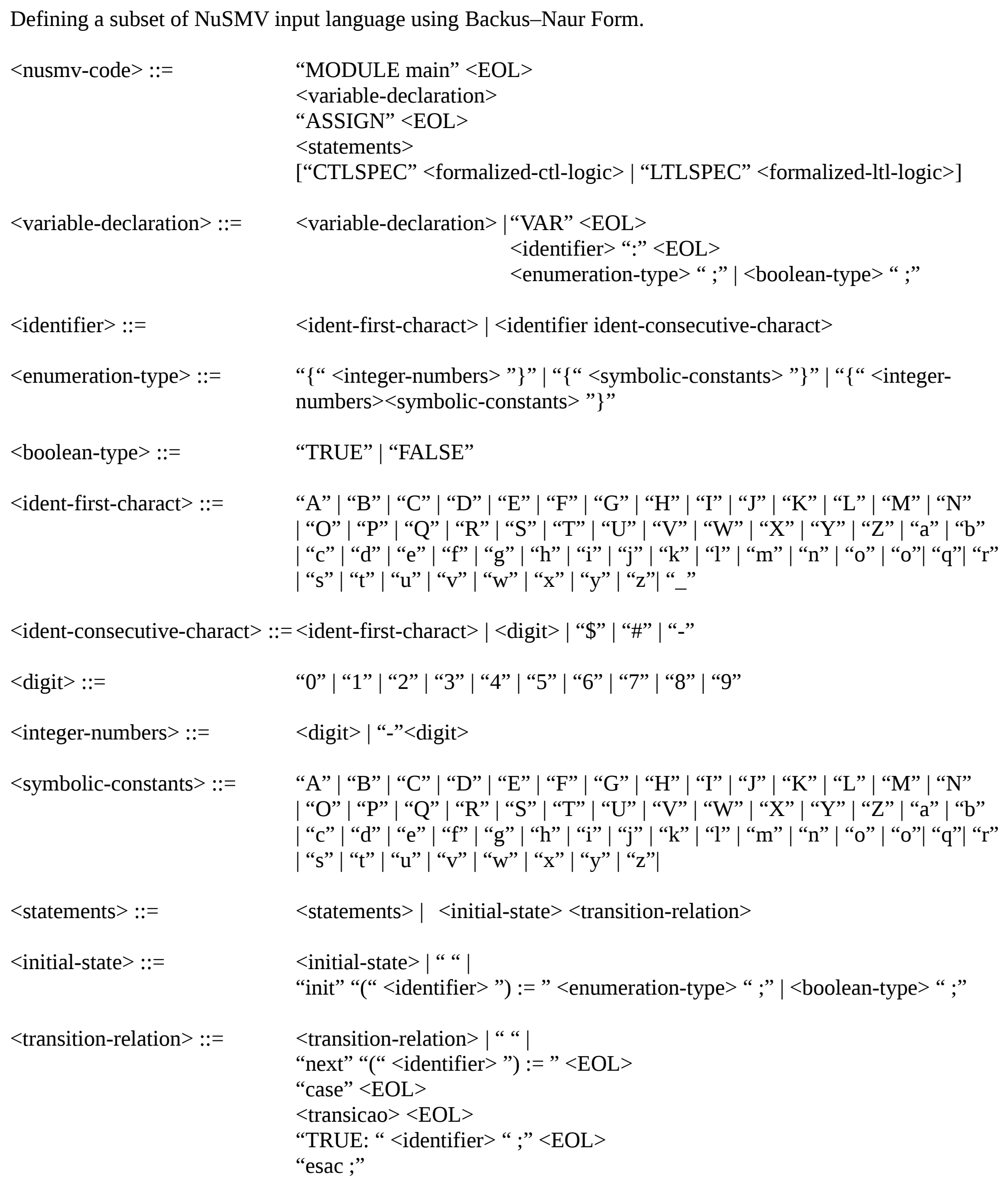
```
MODULE main
VAR
State:
      {Register, DisplayOptions, ChooseOptions, RecordOrder,
      DisplayFinalScreen};
 data ok:
      {dc,false,true};
ASSIGN
init(State):= Register;
init(data ok):= dc;
next(State):=
case
      State=Register & data ok=true : DisplayOptions;
      State=Register & data ok=false : DisplayFinalScreen;
      State=DisplayOptions & data ok=true : ChooseOptions;
       State=ChooseOptions & data ok=true : RecordOrder;
       State=RecordOrder & data ok=true : DisplayFinalScreen;
 TRUE: State;
 esac;
```

Figure 3.11 - NuSMV code for the TS of Figure 3.3

in NuSMV. Thus, a context-free grammar was defined in order to structure the code in NuSMV, as can be seen in Figure 3.12. The grammar was written using the BNF (Backus-Naur Form) (NAUR et al., 1963), one of the main notation techniques for context-free grammars.

The grammar defines all the syntax to build the smv file: the variable declaration, identifiers, and so on, exactly as defined in the NuSMV manual (CAVADA et al., 2005). The grammar defines the main parts of the NuSMV file: the header, variable declaration, and statements. The header is the same for every NuSMV code. There are a number of restrictions related to variable declaration: the variable identifier is composed by first character and consecutive character. Each one has specific allowed types of digits, leters, numbers, and symbols. Statements are composed of statements, initial state, and transition relation. The grammar defines each one of them. There is also the properties to be verified, which start with CTLSPEC or LTLSPEC. XMITS uses this grammar to generate the NuSMV notation.

The next section presents a running example to illustrate all activities proposed in SOLIMVA 3.0.



Defining a subset of NuSMV input language using Backus–Naur Form.

```
<nusmv-code> ::=                 “MODULE main” <EOL>
                                 <variable-declaration>
                                 “ASSIGN” <EOL>
                                 <statements>
                                 [“CTLSPEC” <formalized-ctl-logic> | “LTLSPEC” <formalized-ltl-logic>]

<variable-declaration> ::=       <variable-declaration> |“VAR” <EOL>
                                                         <identifier> “:” <EOL>
                                                         <enumeration-type> “ ;” | <boolean-type> “ ;”

<identifier> ::=                 <ident-first-charact> | <identifier ident-consecutive-charact>

<enumeration-type> ::=           “{“ <integer-numbers> ”}” | “{“ <symbolic-constants> ”}” | “{“ <integer-
                                 numbers><symbolic-constants> ”}”

<boolean-type> ::=               “TRUE” | “FALSE”

<ident-first-charact> ::=        “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” | “J” | “K” | “L” | “M” | “N”
                                 | “O” | “P” | “Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z” | “a” | “b”
                                 | “c” | “d” | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m” | “n” | “o” | “o”| “q”| “r”
                                 | “s” | “t” | “u” | “v” | “w” | “x” | “y” | “z”| “_”

<ident-consecutive-charact> ::= <ident-first-charact> | <digit> | “$” | “#” | “-”

<digit> ::=                      “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

<integer-numbers> ::=            <digit> | “-”<digit>

<symbolic-constants> ::=         “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” | “J” | “K” | “L” | “M” | “N”
                                 | “O” | “P” | “Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z” | “a” | “b”
                                 | “c” | “d” | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m” | “n” | “o” | “o”| “q”| “r”
                                 | “s” | “t” | “u” | “v” | “w” | “x” | “y” | “z”|

<statements> ::=                 <statements> |   <initial-state> <transition-relation>

<initial-state> ::=              <initial-state> | “ “ |
                                 “init” “(“ <identifier> ”) := ” <enumeration-type> “ ;” | <boolean-type> “ ;”

<transition-relation> ::=        <transition-relation> | “ “ |
                                 “next” “(“ <identifier> ”) := ” <EOL>
                                 “case” <EOL>
                                 <transicao> <EOL>
                                 “TRUE: “ <identifier> “ ;” <EOL>
                                 “esac ;”
```

Figure 3.12 - Context-Free Grammar to convert the output of the Converter or of the TUTS into the NuSMV Model Checker Notation.

3.4 A Running Example

This section presents a running example to illustrate the activities proposed in the SOLIMVA methodology version 3.0. Consider more thoroughly the ATM classical example, where the ATM interacts with a customer via a specific interface and communicates with the bank over an appropriate communication link. A user that requests a service from the ATM has to insert a card and enter a personal identification number (PIN). Both pieces of information (card number and PIN) need to be validated. If the credentials of the customer are not valid, the card is ejected. Otherwise, the customer carries out operations that he/she chooses, such as deposit, cash withdrawal, printing receipts, and so on. When the custumer finishes the operation the card is returned to him/her. Figure 3.13 shows the system use case diagram.

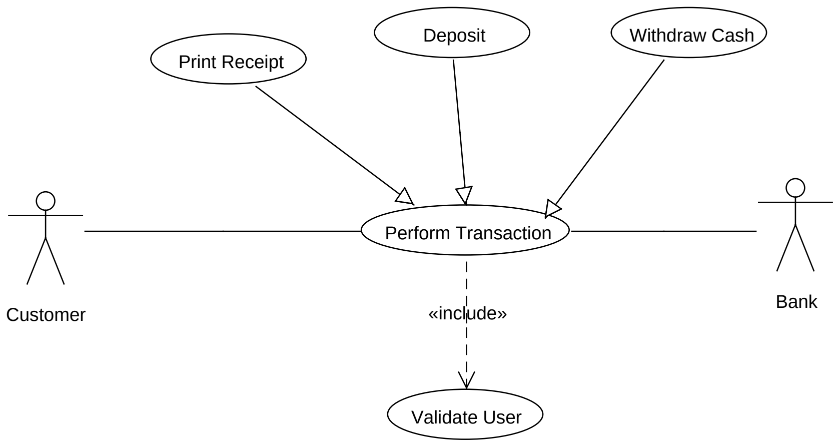


Figure 3.13 - ATM Use Case diagram

In accordance with SOLIMVA 3.0, the first activity is **Identify Scenarios**, observing the use cases. Let's consider the use case *Perform Transaction* (described in Section 3.1) as a scenario to be verified, containing the main success scenario and its variations. Once the scenario is identified, it is time to begin, in fact, the Formal Verification (**Start Formal Verification**). For this, two things are needed: the property to be verified and a model of the system.

To generate a model of the system, five activities must be performed. The first two activities to perform are **Select Diagrams** and **Generate Single TSs**. For simplicity, to facilitate the understanding of the translation, it is presented only the first part of the diagrams, which represents validation of customer credentials.

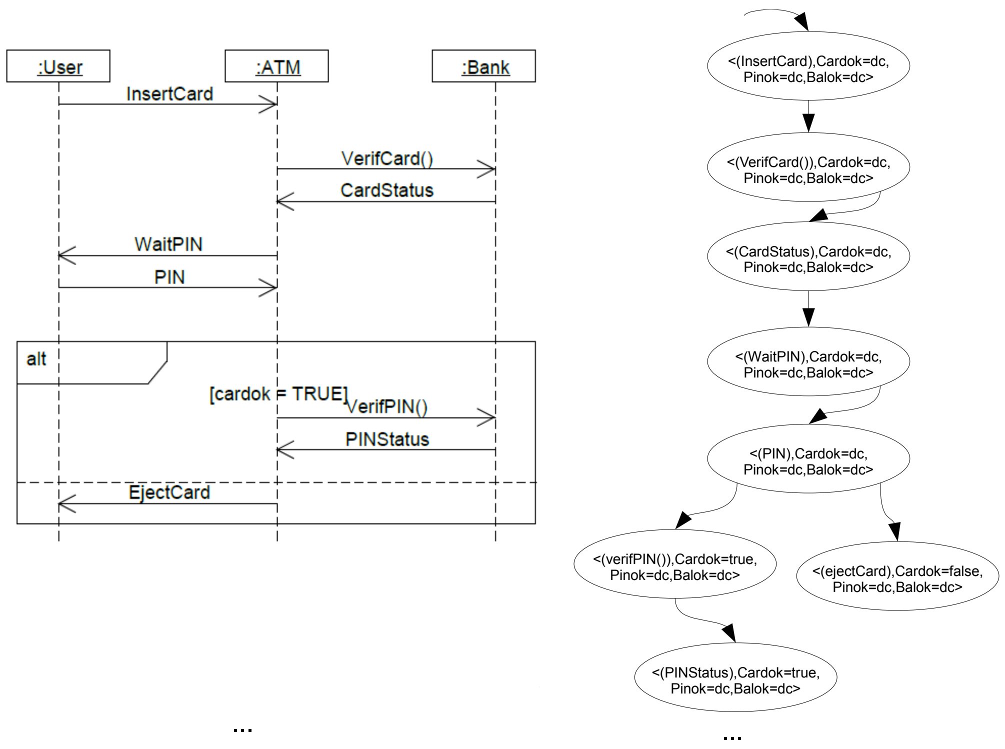
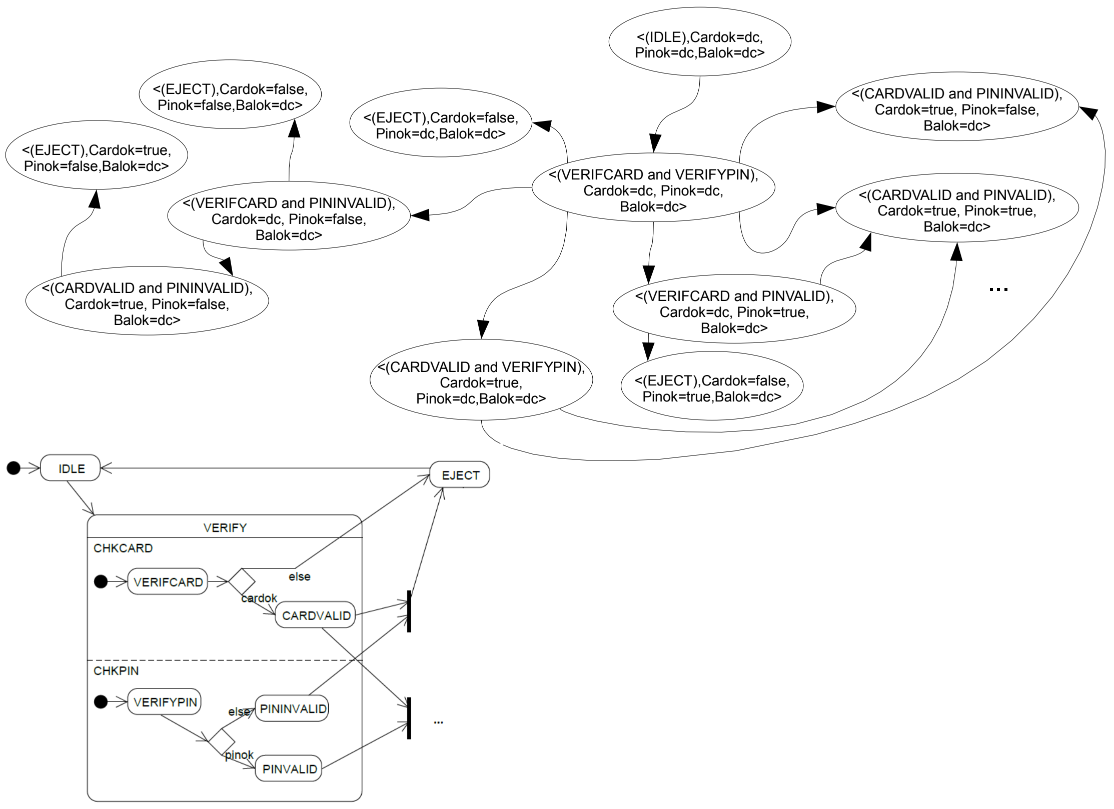

:User
:ATM
:Bank
InsertCard
VerifCard()
CardStatus
WaitPIN
PIN
alt
[cardok = TRUE]
VerifPIN()
PINStatus
EjectCard
...
<(InsertCard),Cardok=dc, Pinok=dc,Balok=dc>
<(VerifCard()),Cardok=dc, Pinok=dc,Balok=dc>
<(CardStatus),Cardok=dc, Pinok=dc,Balok=dc>
<(WaitPIN),Cardok=dc, Pinok=dc,Balok=dc>
<(PIN),Cardok=dc, Pinok=dc,Balok=dc>
<(verifPIN()),Cardok=true, Pinok=dc,Balok=dc>
<(ejectCard),Cardok=false, Pinok=dc,Balok=dc>
<(PINStatus),Cardok=true, Pinok=dc,Balok=dc>
...

Figure 3.14 - Part of ATM Sequence Diagram and its respective TS

Figure 3.14 presents the initial part of ATM sequence diagram and its respective individual TS. Figures 3.15 and 3.16 show the UML behavioral state machine, and activity diagrams modeling the initial part of ATM description, respectively, as well as the individual TSs obtained from each one of the diagrams. The sequence diagram has an *alt* combined fragment, which leads to two paths. The TS obtained from the SMD has multiple paths, due to parallel states. The activity diagram has also parallel activities, which leads to multiple paths.

Continuing, the next activity related to model generation is **Generate Unified TS**. Applying the suggested approach on the three diagrams, the TS shown in Figure 3.17 is achieved. Actually, Figure 3.17 exhibits part of the unified TS, only the parts that are shown in Figures 3.14, 3.15, and 3.16. Note that to generate the unified TS, all rules based on the *gvs* (guard value structure) presented in Figure 3.6 were applied.

Other two activities remaining to model generation are **Generate Model Checker Notation**, and **Simulate Model**. Each state is characterized by the values of the



<(IDLE),Cardok=dc, Pinok=dc,Balok=dc>
<(EJECT),Cardok=false, Pinok=false,Balok=dc>
<(EJECT),Cardok=false, Pinok=dc,Balok=dc>
<(CARDVALID and PININVALID), Cardok=true, Pinok=false, Balok=dc>
<(EJECT),Cardok=true, Pinok=false,Balok=dc>
<(VERIFCARD and VERIFYPIN), Cardok=dc, Pinok=dc, Balok=dc>
<(VERIFCARD and PININVALID), Cardok=dc, Pinok=false, Balok=dc>
<(CARDVALID and PINVALID), Cardok=true, Pinok=true, Balok=dc>
<(CARDVALID and PININVALID), Cardok=true, Pinok=false, Balok=dc>
<(VERIFCARD and PINVALID), Cardok=dc, Pinok=true, Balok=dc>
...
<(CARDVALID and VERIFYPIN), Cardok=true, Pinok=dc,Balok=dc>
<(EJECT),Cardok=false, Pinok=true,Balok=dc>
IDLE
EJECT
VERIFY
CHKCARD
VERIFCARD
else
cardok
CARDVALID
CHKPIN
VERIFYPIN
else
PININVALID
pinok
PINVALID
...

Figure 3.15 - Part of ATM Behavioral State Machine Diagram and its respective TS

variables. There were identified four main variables that characterize the TS:

a) *State={(InsertCard,Idle,InsertCard),...,(PinStatus,VerifyPin,AuthorizePin)}*;

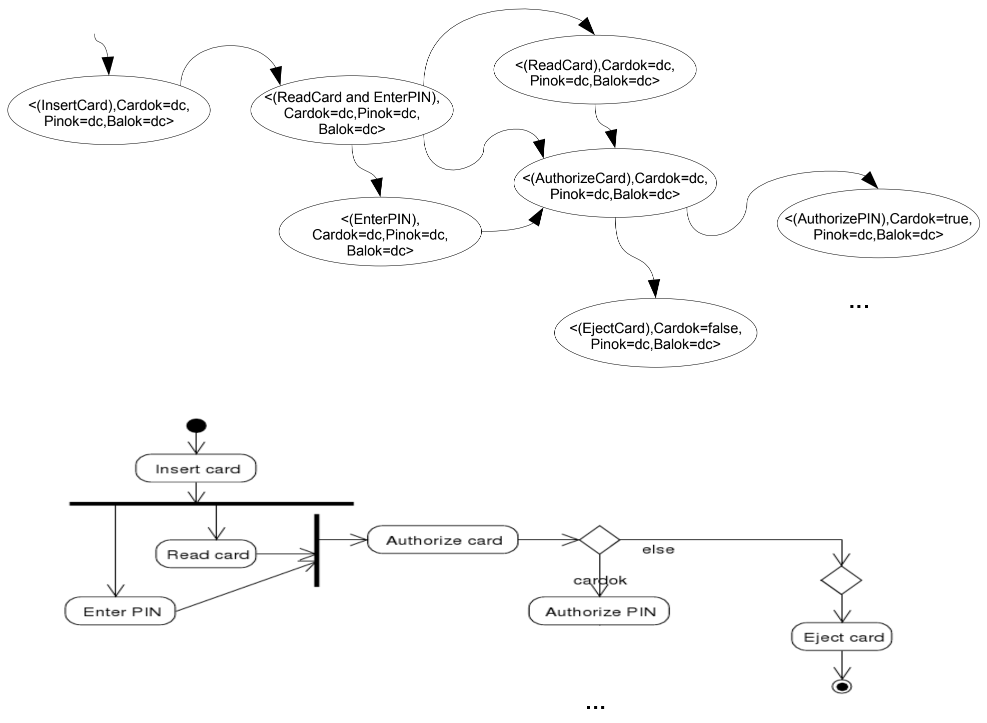
b) *CardOk = {dc,false,true}*. *CardOk* represents the card validation;

c) *PinOk = {dc,false,true}*. *PinOk* represents the PIN validation; and

d) *BalOk = {dc,false,true}*. *BalOk* represents the available amount of the customer account. *CardOk, PinOK* and *BalOk* comes from the guards identified within the UML diagrams.

Using the guidelines and the grammar proposed in last section, it is possible to obtain the NuSMV code for the TS. Figure 3.18 shows the code partially.

Activity **Simulate Model** is performed to validate the model. Before applying Model Checking, it is necessary to simulate the model to see if it is consistent.



<(InsertCard),Cardok=dc, Pinok=dc,Balok=dc>
<(ReadCard and EnterPIN), Cardok=dc,Pinok=dc, Balok=dc>
<(ReadCard),Cardok=dc, Pinok=dc,Balok=dc>
<(EnterPIN), Cardok=dc,Pinok=dc, Balok=dc>
<(AuthorizeCard),Cardok=dc, Pinok=dc,Balok=dc>
<(AuthorizePIN),Cardok=true, Pinok=dc,Balok=dc>
<(EjectCard),Cardok=false, Pinok=dc,Balok=dc>
...
Insert card
Read card
Enter PIN
Authorize card
else
cardok
Authorize PIN
Eject card
...

Figure 3.16 - Part of ATM Activity Diagram and its respective TS

When running the model checker, this activity is very valuable, because one can see all possible states and all reachable states. Indeed, when running NuSMV model checker for this ATM example, in total, there are 756 states, 86 of which are reachable states. Figure 3.19 shows the three UML diagrams, now complete, used to perform the case study. From left to right, one can see the sequence, state machine, and activity diagrams.

With respect to the property, two activities must be performed: **Select Requirements** and **Formalize Properties**. Considering the use case *Perform Transaction*, it is possible to extract two relevant user-defined properties related to requirements. As the TS has multiple paths, Computation Tree Logic (CTL) (BAIER; KATOEN, 2008) was chosen to formalize the properties. Note that the properties could be formalized using LTL as well, considering that NuSMV supports both.

a) Requirement 1: *the customer can perform transactions only if he/she has a valid card and a valid personal identification number (PIN). Otherwise, he/ she can not perform any kind of transaction.* This property can be

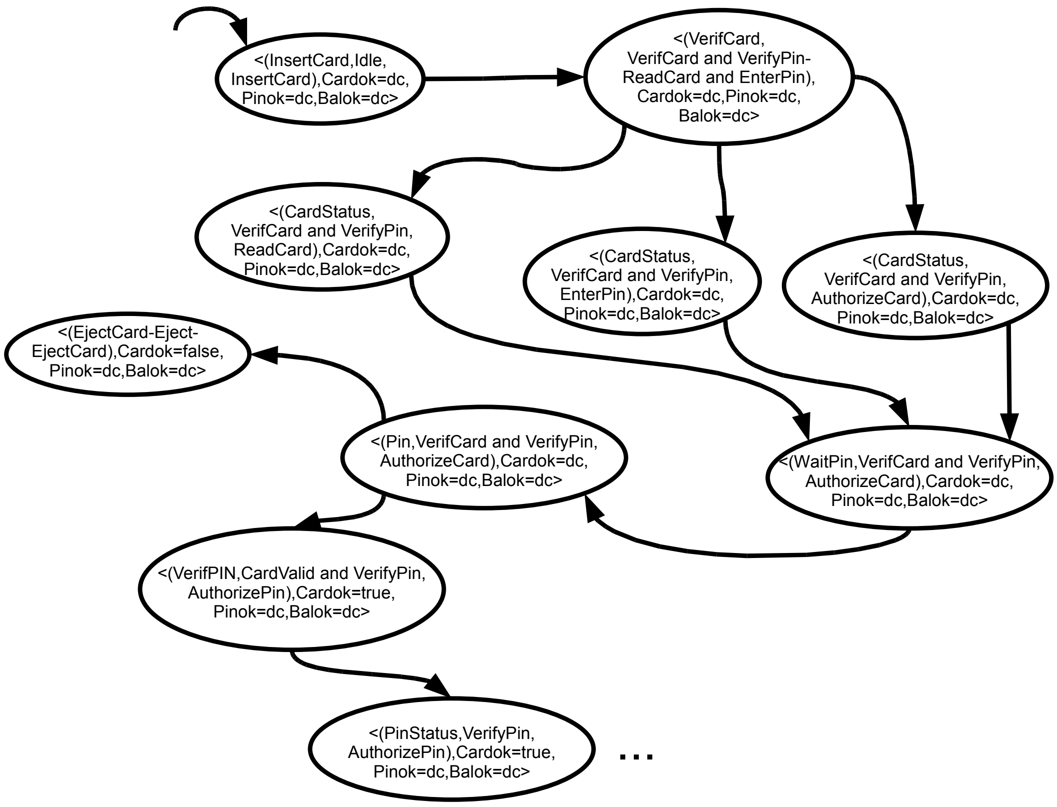
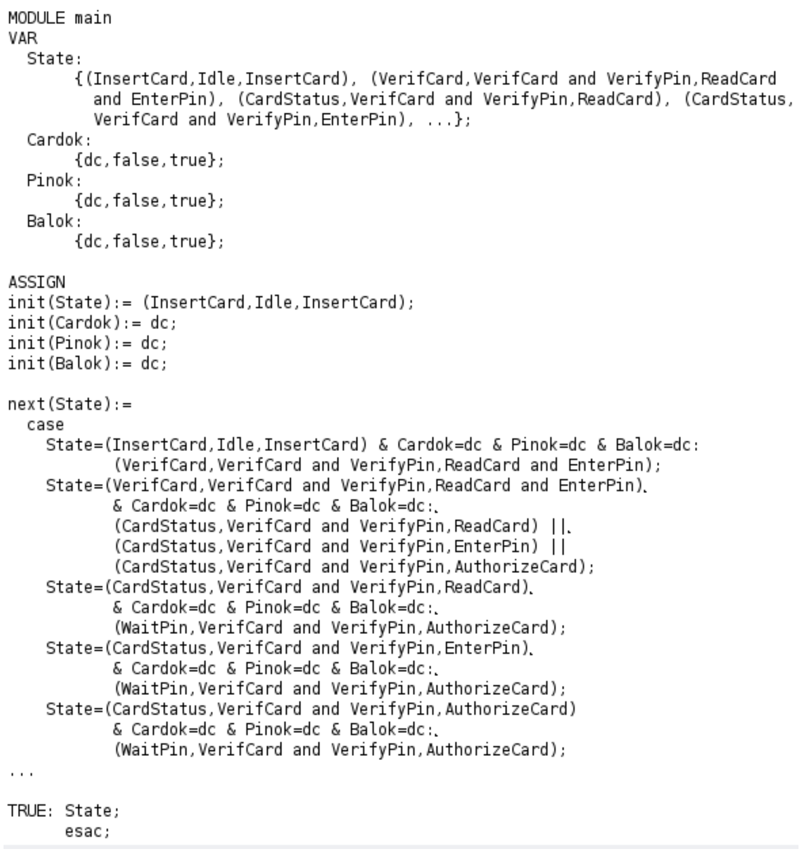

<(InsertCard,Idle, InsertCard),Cardok=dc, Pinok=dc,Balok=dc>
<(VerifCard, VerifCard and VerifyPin-ReadCard and EnterPin), Cardok=dc,Pinok=dc, Balok=dc>
<(CardStatus, VerifCard and VerifyPin, ReadCard),Cardok=dc, Pinok=dc,Balok=dc>
<(CardStatus, VerifCard and VerifyPin, EnterPin),Cardok=dc, Pinok=dc,Balok=dc>
<(CardStatus, VerifCard and VerifyPin, AuthorizeCard),Cardok=dc, Pinok=dc,Balok=dc>
<(EjectCard-Eject-EjectCard),Cardok=false, Pinok=dc,Balok=dc>
<(Pin,VerifCard and VerifyPin, AuthorizeCard),Cardok=dc, Pinok=dc,Balok=dc>
<(WaitPin,VerifCard and VerifyPin, AuthorizeCard),Cardok=dc, Pinok=dc,Balok=dc>
<(VerifPIN,CardValid and VerifyPin, AuthorizePin),Cardok=true, Pinok=dc,Balok=dc>
<(PinStatus,VerifyPin, AuthorizePin),Cardok=true, Pinok=dc,Balok=dc>
...

Figure 3.17 - Part of the final TS obtained from the three diagrams

formalized using the **Absence Pattern and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:
∀□ ((CardOK = false ∨ PinOK = false) → ∀□ (¬(State = WaitAccount-CardValidandPinValid-InitiateTransaction)))

CardOk and *PinOk* refers to the validation of the customer credentials. *CardOk = true* means the card is valid. *PinOk = true* means the identifier is valid. Each value of each variable is considered an atomic proposition. *WaitAccount-CardValidandPinValid-InitiateTransaction* is the first state on the final TS that can only be reached when *CardOk* and *PinOk* are *true*, that is, when the custumer has card and PIN valids (it means that the customer *can perform transactions*). It is easy to see this when looking at Figure 3.19. If one observes the individual UML diagrams, it is possible to see that state *WaitAccount* for the SD is the first reachable state when both *CardOk* and *PinOk* are *true*, for SMD *CardValidandPinValid*, and also *InitiateTransaction* for AD.



```
MODULE main
VAR
  State:
       {(InsertCard,Idle,InsertCard), (VerifCard,VerifCard and VerifyPin,ReadCard
         and EnterPin), (CardStatus,VerifCard and VerifyPin,ReadCard), (CardStatus,
         VerifCard and VerifyPin,EnterPin), ...};
  Cardok:
       {dc,false,true};
  Pinok:
       {dc,false,true};
  Balok:
       {dc,false,true};

ASSIGN
init(State):= (InsertCard,Idle,InsertCard);
init(Cardok):= dc;
init(Pinok):= dc;
init(Balok):= dc;

next(State):=
  case
    State=(InsertCard,Idle,InsertCard) & Cardok=dc & Pinok=dc & Balok=dc:
           (VerifCard,VerifCard and VerifyPin,ReadCard and EnterPin);
    State=(VerifCard,VerifCard and VerifyPin,ReadCard and EnterPin).
           & Cardok=dc & Pinok=dc & Balok=dc:.
           (CardStatus,VerifCard and VerifyPin,ReadCard) ||.
           (CardStatus,VerifCard and VerifyPin,EnterPin) ||
           (CardStatus,VerifCard and VerifyPin,AuthorizeCard);
    State=(CardStatus,VerifCard and VerifyPin,ReadCard).
           & Cardok=dc & Pinok=dc & Balok=dc:.
           (WaitPin,VerifCard and VerifyPin,AuthorizeCard);
    State=(CardStatus,VerifCard and VerifyPin,EnterPin).
           & Cardok=dc & Pinok=dc & Balok=dc:.
           (WaitPin,VerifCard and VerifyPin,AuthorizeCard);
    State=(CardStatus,VerifCard and VerifyPin,AuthorizeCard)
           & Cardok=dc & Pinok=dc & Balok=dc:.
           (WaitPin,VerifCard and VerifyPin,AuthorizeCard);
...

TRUE: State;
      esac;
```

Figure 3.18 - Part of the NuSMV code for the TS of Figure 3.17

b) Requirement 2: *whenever the specified amount exceeds the level of available funds, it should be possible for the user to request a new cash advance operation if the user wishes to correct the amount.* The property can be formalized using the **Existence Pattern and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

$\neg\exists$ [$\neg$ (State = InsuffFunds-Modify-ShowBalance $\wedge$ $\forall\Diamond$ (State = CashAdvance-Chkbal-CheckBalance)) $\cup$ (State = InsuffFunds-Modify-ShowBalance $\wedge$ $\neg$ (State = InsuffFunds-Modify-ShowBalance $\wedge$ $\forall\Diamond$ (State = CashAdvance-Chkbal-CheckBalance)))]

Here, the state that allows the customer to perform a cash operation is *CashAdvance-Chkbal-CheckBalance* and the state that indicates that the available funds are insufficient is *InsuffFunds-Modify-ShowBalance.* To formalize the properties, it is necessary to see the final TS and its variables.

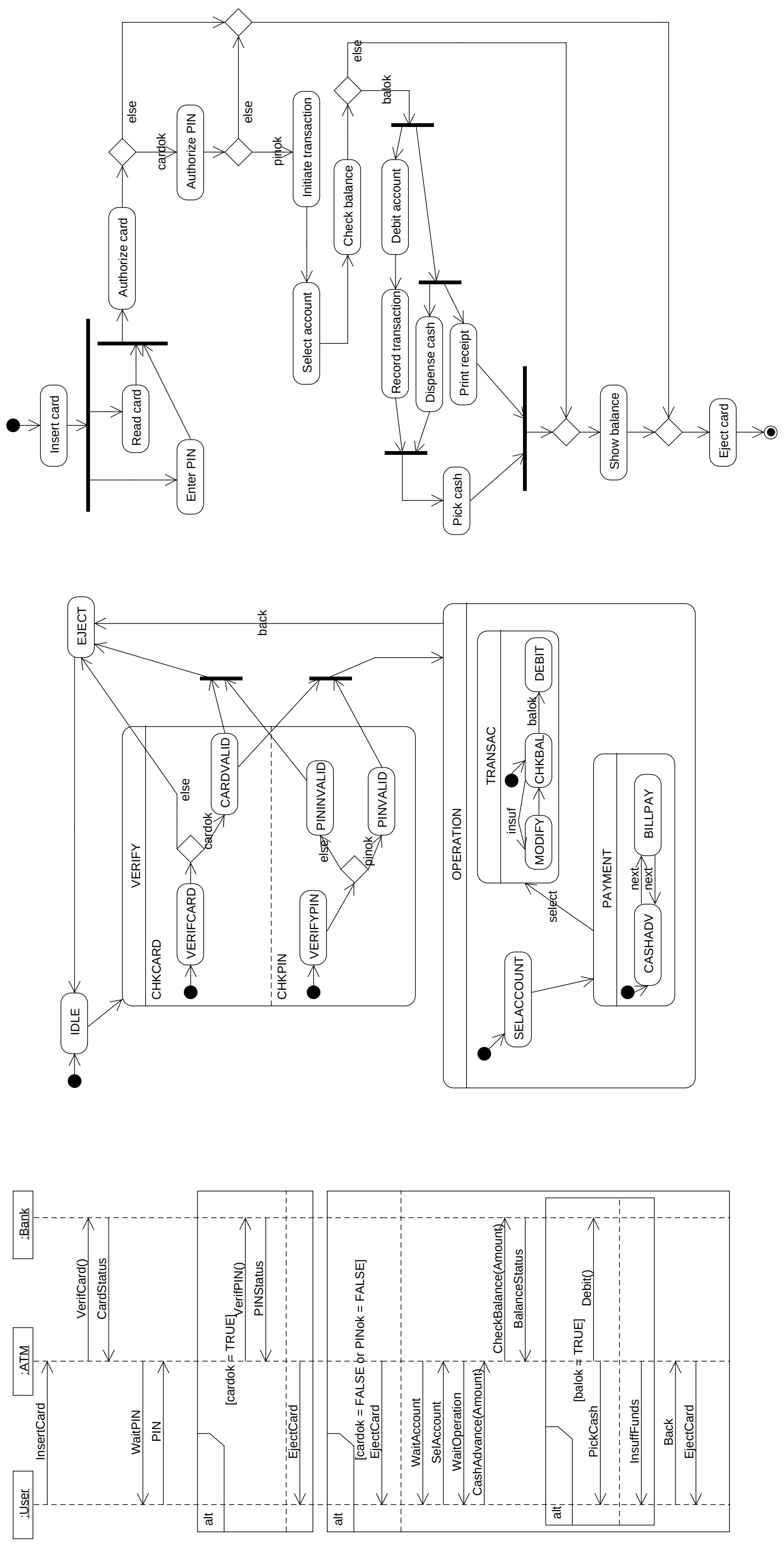
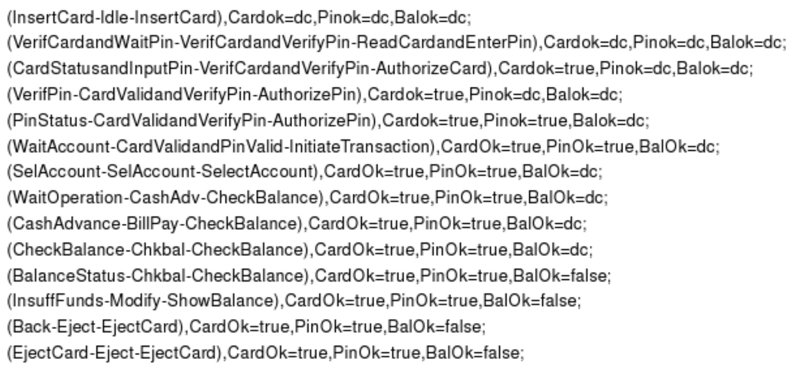


Figure 3.19 - UML diagrams used for ATM example. Adapted from Debbabi et al. (2010)

Next activity of SOLIMVA 3.0 is **Apply Model Checking**. After applying Model Checking, the results indicate that the first property is satisfied. However, the second property is violated, as can be seen in the counterexample shown in Figure 3.20. Each line represents one state in the TS. When analyzing the counterexample, one can note that it is not possible to reach state *CashAdvance-ChkBal-CheckBalance* from state *InsuffFunds-Modify-ShowBalance*.



```
(InsertCard-Idle-InsertCard),Cardok=dc,Pinok=dc,Balok=dc;
(VerifCardandWaitPin-VerifCardandVerifyPin-ReadCardandEnterPin),Cardok=dc,Pinok=dc,Balok=dc;
(CardStatusandInputPin-VerifCardandVerifyPin-AuthorizeCard),Cardok=true,Pinok=dc,Balok=dc;
(VerifPin-CardValidandVerifyPin-AuthorizePin),Cardok=true,Pinok=dc,Balok=dc;
(PinStatus-CardValidandVerifyPin-AuthorizePin),Cardok=true,Pinok=true,Balok=dc;
(WaitAccount-CardValidandPinValid-InitiateTransaction),CardOk=true,PinOk=true,BalOk=dc;
(SelAccount-SelAccount-SelectAccount),CardOk=true,PinOk=true,BalOk=dc;
(WaitOperation-CashAdv-CheckBalance),CardOk=true,PinOk=true,BalOk=dc;
(CashAdvance-BillPay-CheckBalance),CardOk=true,PinOk=true,BalOk=dc;
(CheckBalance-Chkbal-CheckBalance),CardOk=true,PinOk=true,BalOk=dc;
(BalanceStatus-Chkbal-CheckBalance),CardOk=true,PinOk=true,BalOk=false;
(InsuffFunds-Modify-ShowBalance),CardOk=true,PinOk=true,BalOk=false;
(Back-Eject-EjectCard),CardOk=true,PinOk=true,BalOk=false;
(EjectCard-Eject-EjectCard),CardOk=true,PinOk=true,BalOk=false;
```

Figure 3.20 - Counterexample for property 2

Activity **Generate Report of System Defects based on CounterExamples** is implemented by the XMITS, when it generates the final txt file, as can be seen in the next chapter.

3.5 Final Remarks

This chapter presented the proposed solution to allow applying Model Checking to software developed in accordance with UML. It presented SOLIMVA 3.0 and the activities that make up the methodology. The details of the translation of UML diagrams to individual Transition Systems were explained, as well as the algorithms to allow these translations.

After that, the translation to a unified Transition System was detailed. It was explained how a new state is composed, the rules that conducts the unification, and the algorithm that implements the junction. Then, the guidelines and a grammar to transform the unified TS to the input language of NuSMV model checker were described. And finally, a running example was presented to illustrate the methodology.

The next chapter presents XMITS, the tool that was developed for automate the main activities of SOLIMVA 3.0. The XMITS modules and its architecture are explained.

4 XMITS - XML Metadata Interchange to Transition System

This chapter presents XMITS - XML Metadata Interchange to Transition System, a tool developed to support the main three activities proposed in SOLIMVA 3.0 (shown in Figure 4.1), to achieve Model Checking of UML-based software: **Generate single TSs, Generate Unified TS,** and **Generate Model Checker Notation**. An abridged version of this chapter can be seen in (SANTOS et al., 2014b), which presented a very preliminary version of XMITS, and in (ERAS et al., 2015), where the actual version of XMITS is presented.

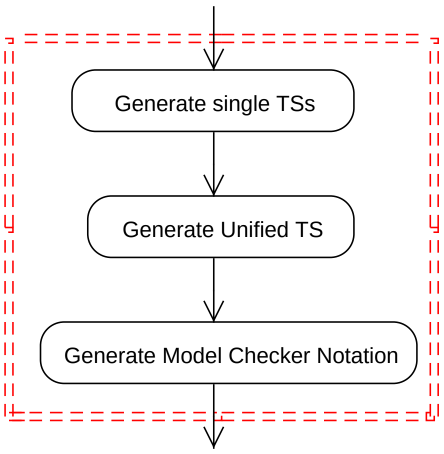


Figure 4.1 - Activities of SOLIMVA 3.0 automated by XMITS

XMITS is essential to the application of SOLIMVA 3.0. The development of XMITS is part of an effort to allow the use of Formal Methods in practice, to real software projects. The tool facilitates the process of applying Formal Verification during the software development, so that it becomes transparent to the user. Briefly explaining, the analyst considers a use case, with its narrative description, or requirements in Natural Language; and also the corresponding UML behavioral diagrams that represent the solution to meet the requirements (use cases or pure textual requirements) are taken into account.

XMITS performs a three-step translation. First, it translates individual types of diagrams (SD, AD, SMD) into a TS in a simple intermediate format. After that, XMITS merges all single TSs into a unified TS. Finally, the tool transforms this unified TS into the notation of the model checker (NuSMV). Requirements are formalized using a temporal logic, such as CTL or LTL, following Dwyer's specification patterns (DWYER et al., 1999). If there is a problem with the design (UML behavioral diagrams), a counterexample is presented by the model checker.

Due to the modular nature proposed for the system, a language that supports the object-oriented programming paradigm was chosen: Java (ORACLE, 2011). The aim was to develop a tool that is extensible. If one wants to add another UML diagram to the approach, this can be easily implemented in XMITS. Eclipse (The Eclipse Foundation, 2015) was the Integrated Development Environment (IDE) chosen for implementing XMITS. UML 2.4.1 specifications were considered to produce the design artifacts. The design artifacts are then exported into XMI (XML Metadata Interchange) format, and are inputted to XMITS. XMI is a standard of OMG (Object Management Group) (OMG, 2015) for exchanging metadata information via Extensible Markup Language (XML). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels). OMG has a documentation of formally released versions of XMI.

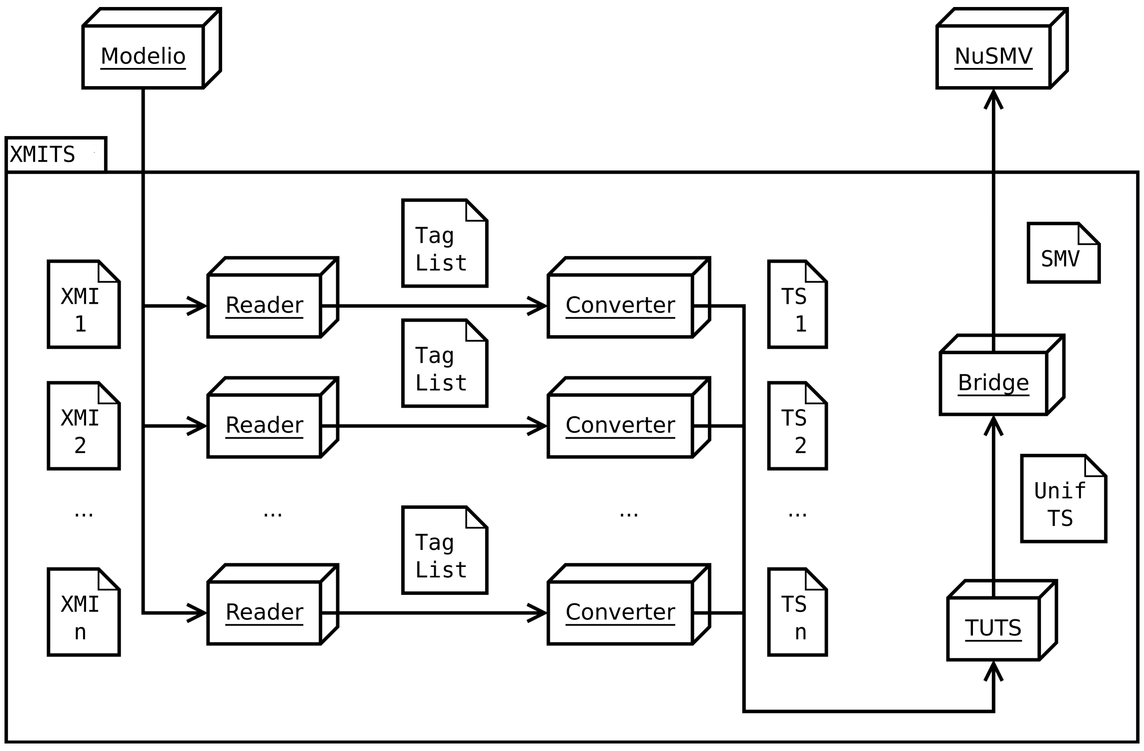


Figure 4.2 - Flow diagram of XMITS

Figure 4.2 presents a data flow within XMITS. The tool interoperates with two other tools: Modelio 3.2 (MODELIOSOFT, 2011), that is the software used to produce the UML artifacts; and the NuSMV model checker. XMITS consists of five modules, of which four of them can be viewed in Figure 4.2: the **Reader**, that transforms the XMI file to a list of tags; the **Converter**, that transforms the list of tags to a single TS; the **TUTS** (The Unified Transition System), that transforms the single TSs to the unified TS; and the **Bridge** module, that transforms the unified TS to the model checker notation. There is still the **Global** module, that is not shown in the figure, which controls several functions. All these modules are detailed in the next section.

4.1 XMITS Architecture

The actual XMITS software architecture is shown in Figure 4.3, where it is possible to visualize all the five modules of the tool. In total, there are 121 classes: 7 for the Reader Module; 61 for the Converter Module; 26 for the TUTS Module; 7 for the Bridge Module; and 20 for the Global Module. There are about 11,120 lines of code instructions.

All XMITS modules work together following a sequence of interactions to produce the final output. Figure 4.4 shows the detailed workflow related to XMITS. The process begins by the XMI files produced in the Modelio software. XMITS can process n XMI files. This is because sometimes there are two activity diagrams related to the scenario being analyzed; or three sequence diagrams. In this way, due to each combination of the three accepted UML behavioral diagrams, it is possible to process n XMI files. The only restriction is that, as previously emphasized, it demands that at least one SD must be available. For n XMI files, XMITS calls n instances of the Reader and n instances of the Converter. All the Converter outputs feed the TUTS, which, in turn, combines the Transition Systems into a unique TS.

Once this is done, it is possible to call the Bridge module to translate the Transition System into the NuSMV notation. The Global module has also a *Printer* function to visualize the Transition System output directly in the Java terminal, or save it as a *txt* file. Follows are the details of XMITS modules.

4.1.1 The Reader Module

To process any XMI input, the first action is to convert the file to a useful format. The Reader module is responsible for converting the XMI into a structure for the

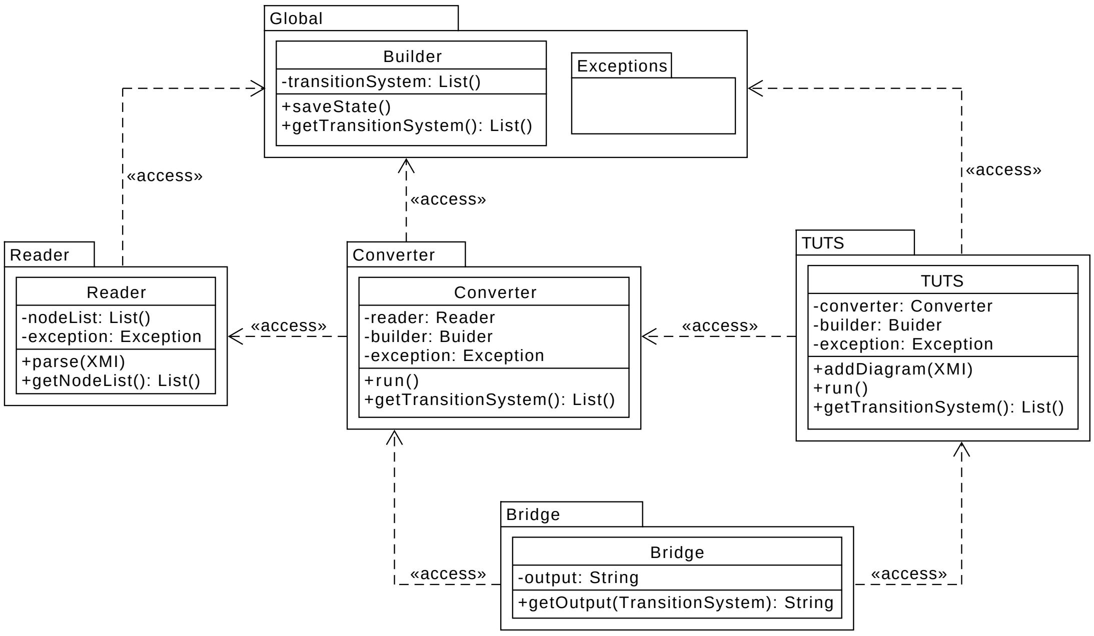


Figure 4.3 - XMITS software architecture

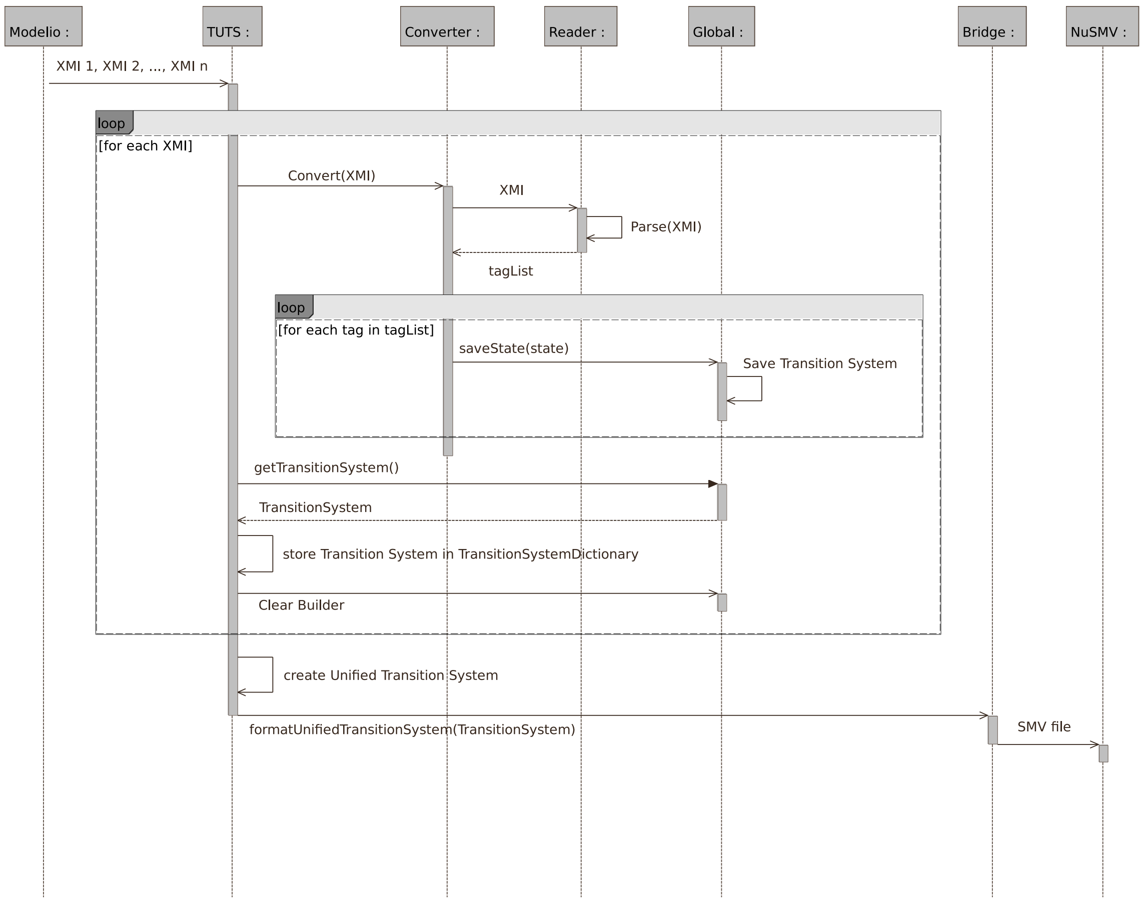


Figure 4.4 - XMITS detailed workflow

Converter module. The input file, which is an XML file, is processed by SAX (Simple API for XML). The Reader module uses this API to save all the content of the input into a linked list, storing each tag in a Java class according to its characteristics. Figure C.8 in Appendix C shows the class diagram for this module.

4.1.2 The Converter Module

Figure 4.5 shows the operation of Converter module. After the XMI file is processed by the Reader, the Converter parses its content to confirm if it is a valid UML diagram and which diagram it is. After identification, the diagram is redirected to its specific Collector depending on the type of the diagram: Sequence, Activity, State Machine. The collectors are responsible for reading the file, line by line, and classifying its elements into six categories: *State*, *Fork*, *Join*, *Decision*, *Connection*, or *Default*. These are the basic categories used by the Converter's Logics to define all possible elements in the behavioral UML diagram that this research is working with. Then, the Converter Logics is called to process the list containing the elements, classified in these categories.

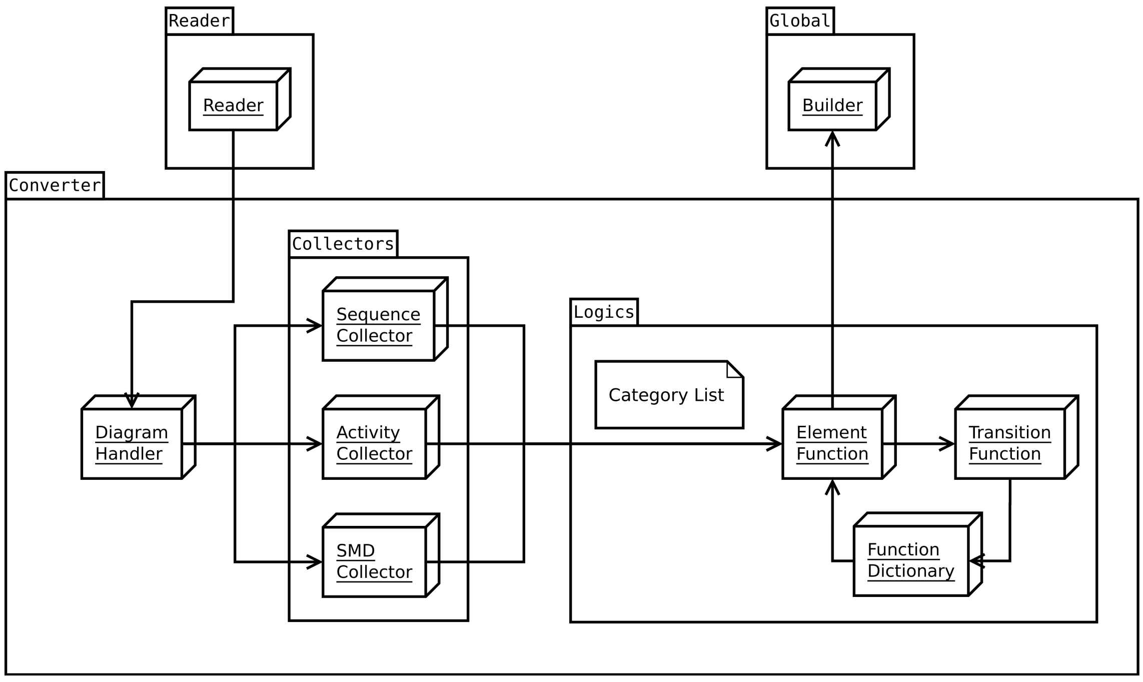


Figure 4.5 - Flow diagram of Converter Module

Currently, XMITS accepts Sequence, Activity, and State Machine UML diagrams but, if at anytime it is necessary to upgrade the system with more diagrams, the only change required is to implement a new collector in the *collectors* package. Once the categories are formatted in a list, they are redirected to the Logics, as can be seen in Figure 4.5. Actually, the list is a kind of state machine, so the Logics gets its first state and starts to process all the transitions and subsequent states. For each kind of element in the mentioned six categries there is a specific function in the *Function Dictionary*, which is the class responsible for calling the correct function for each category. The functions process the information of the incoming element.

While the list is being processed, the functions call an instance of *Builder*, an important class of the Global module, responsible for creating the Transition System output. The *Builder* class always returns its own instance, so there will never be more than one *Builder*, no matter how many times it has been used in the system. Figure 4.6 shows the package diagram for this module.

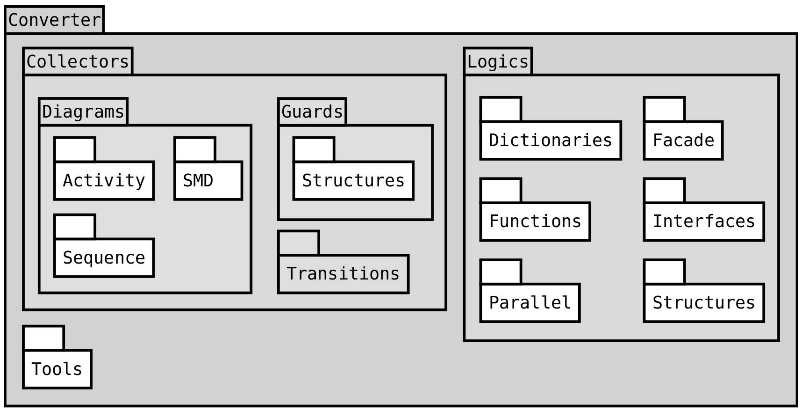


Figure 4.6 - Package Diagram of Converter Module

4.1.3 The TUTS Module

The Unified Transition System (TUTS) module is an implementation of the approach proposed in Subsection 3.2.2, and it is responsible for unifying the single Transition Systems into an unified Transition System output. The TUTS is composed of five packages, as can be seen in Figure 4.7: *dictionaries*, *interfaces*, *facade*, *logics* and *tools*. The *dictionaries* package holds all dictionaries used by the module. A *Dictionary* is a class of unique instance, which keeps an information that can not be duplicated. It was implemented using Singleton design pattern solution. A *Dictionary* class has data structures such as linked lists, arrays, and so on. It has functions to add and to consult information. The *interfaces* package defines the basic function interfaces to generalize the code. All dictionaries are initialized by a *facade* class stored in the *facade* package. The core processing happens inside the *logics* package. Finally, the *tools* package provides many important functions for the unification process.

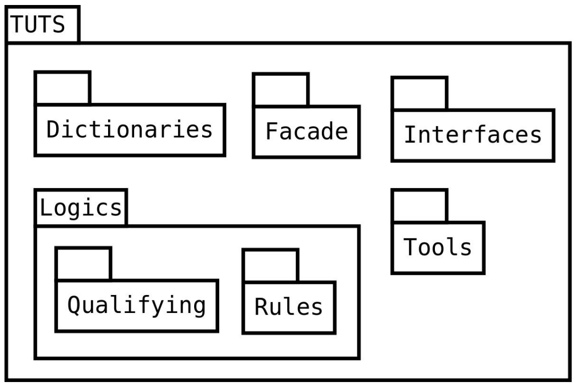


Figure 4.7 - Package Diagram of TUTS Module

The flow of processing inside the TUTS is based on an iteration on the Transition System inputs, unifying each state of each Transition System according to a rule. Algorithm 4, presented in Section 3.2.2, shows how the unified TS is generated. Each Transition System represents a UML diagram transformed by the Converter Module. They are processed all together, side by side, like cars running on a race track. To walk from a state to another, the iterator needs to obey rules in a specific order (actually, these rules are based on the *gvs* - guard value structure - explained in Figure 3.6). All these rules are stored in a *Rule Dictionary*, in the *dictionaries* package. The *Rule Dictionary* is an array list containing the four rules, as can be seen in Figure 3.7. If the states fit in the First Rule, there is a specific kind of iteration; if not, the system tries to apply the Second Rule, and so on. The last rule is an "end of race", a function that simply stops all the iterators because there is nothing more to iterate, *i.e.*, all the diagrams reach their last state. This process can be observed in Figure 4.8.

The *gvs* can be formalized as follows. Given that a Transition System is composed of states and that each state w contains its own *gvs*, the gvs_w is defined as an ordered n-tuple $(x_1, x_2, ..., x_n)$, where n is a positive integer and $x_1, x_2, ..., x_n$ is a distinct sequence of elements. An element x_k of the tuple represents a guard along with its value. Two ordered n-tuples $(x_1, x_2, ..., x_n)$ and $(y_1, y_2, ..., y_n)$ are equal if, and only if, $x_i = y_i$, $\forall\ i = 1...n$.

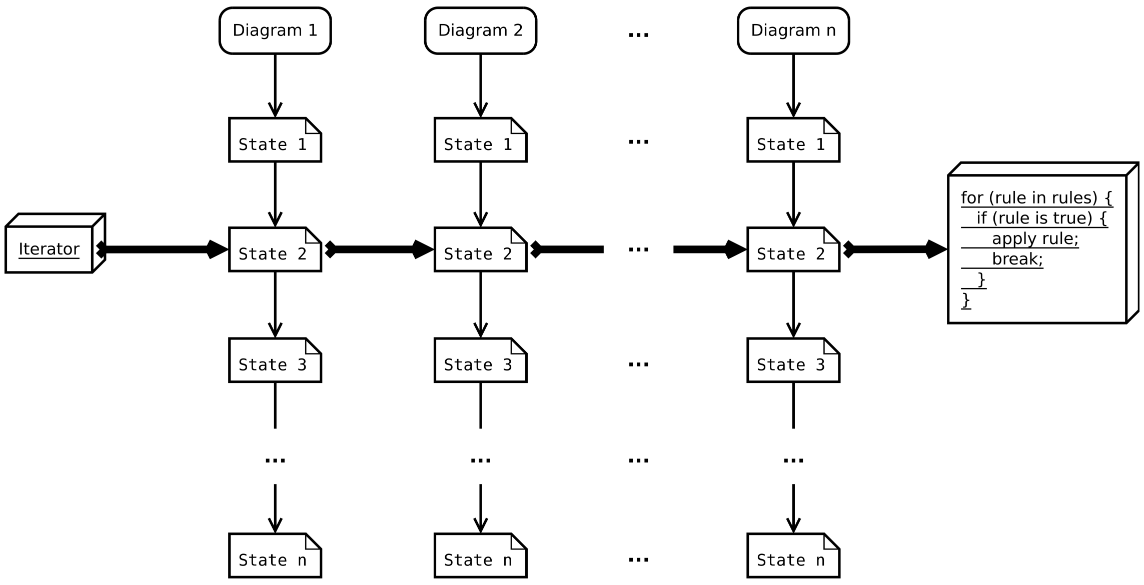


Figure 4.8 - The iteration over the diagrams in the TUTS module

Let p be a state of a Transition System. $Nextgvs(p)$ is defined as the set of all gvs of the next states of p. It is written as $Nextgvs(p) = \{gvs_1, gvs_2, ..., gvs_k\}$, where elements $gvs_1, gvs_2, ..., gvs_k$ represent the gvs of each one of the k next states of p. A state q is a next state of p if, and only if, there exists a transition from state p to state q, written as $p \to q$.

Returning to Algorithm 4, let $S = \{s_1, s_2, ..., s_m\}$ be the set of states of the current iteration, each one representing a digram, as can be seen in Figure 4.8. The four rules are defined as follows:

a) First Rule: $\forall\ a \in Nextgvs(s_1)\ \exists\ b \in Nextgvs(s_j)\ |\ a = b, j = 2...m.$

b) Second Rule: $\exists\ s_k \in S\ |\ gvs_{s_k} = r,\ where\ r \in Nextgvs(s_k).$

c) Third Rule: $if\ \exists\ a \in Nextgvs(s_1),\ then\ a \neq b\ \forall\ b \in Nextgvs(s_j)$, $j = 2...m.$

d) Fourth Rule: *All the other cases.*

The First Rule expects to find all states in the iterators position with the same guard conditions. This rule means "all states are equal", and it is the only time in the process when the *Builder* class is called to generate an output. An unified state is possible if, and only if, all the states are equal. If all states are not equal then the system tries the Second Rule. The Second Rule looks for a scenario where some diagrams changed their guard conditions but others did not. In this hybrid condition the iterator advances just the diagrams which did not change, and the First Rule can be applied. If the Second Rule wasn't applied, then the system tries the Third Rule. Here not all states are equal. If there are at least two equal states, the First Rule is applied to them. Then, if all diagrams have different states, there is no way to create a unified state, so all diagrams start to run in a different thread with no interaction with the others. If even the Third Rule wasn't applied, there is a Final Rule to end all the process.

Every time the diagrams can not be unified, the system divides the process and a new thread is created. This process happens when a division occurs in the diagram or in the Third Rule, when all states are different. In this way, TUTS avoids joining different states and simplifies the process. While there is more than one thread running, the *Builder* class falls into a concurrency of processes, trying to save a Transition System. To solve this, the save function runs into a critical region.

4.1.4 The Bridge Module

The Bridge module is responsible for translating the TSs into the notation of the NuSMV model checker. It allows to translate both the output of the Converter as well as the output of TUTS. So, if one needs to translate only a single TS, it is possible. Thus, the Bridge module allows the translation of both the single TSs or the unified TS.

The Bridge module accomplishes the model transformation using the context-free grammar shown in Figure 3.12. As already explained in Section 3.3, the grammar defines all the syntax of NuSMV. It is possible to visualize the output directly in Java terminal or save it in a file. To make the conversion, the Bridge module iterates all over the Transition System several times, one for creating each section of the output file: header, variables, initial state (*Assign*), transitions (*Next*), and guards. There is a function dedicated to create each one of these sections of the NuSMV file. Then, the *formater()* function iterates over all *String* values looking for non accepted characters and replaces then in order to apply the rules defined by the grammar. Figure C.9 in Appendix C shows the class diagram for this module.

4.1.5 The Global Module

The Global module provides some useful resources. The *Builder* and the main data structure, widely used by the Converter and TUTS modules are here. The *Builder* is responsible for writing the Transition System during the conversion and unification processes. It is also responsible for giving access to the Transition System output after all the processes have been finalized. All data processed by conversion and unification are encapsulated into a data structure defined in the Global module. Figure C.10 in Appendix C shows the class diagram for this module.

The Transition System itself is defined in this module. This module is also responsible for holding all the exception classes used by all other modules. Finally, there are two useful functions for global use: an *ID generator* and a *Printer*. The *ID generator* provides a global unique ID generation for all the inner processes. The *Printer* is used for textual visualization of the Transition System tree. It is possible to see this output on a terminal console or save it directly to a *txt* file.

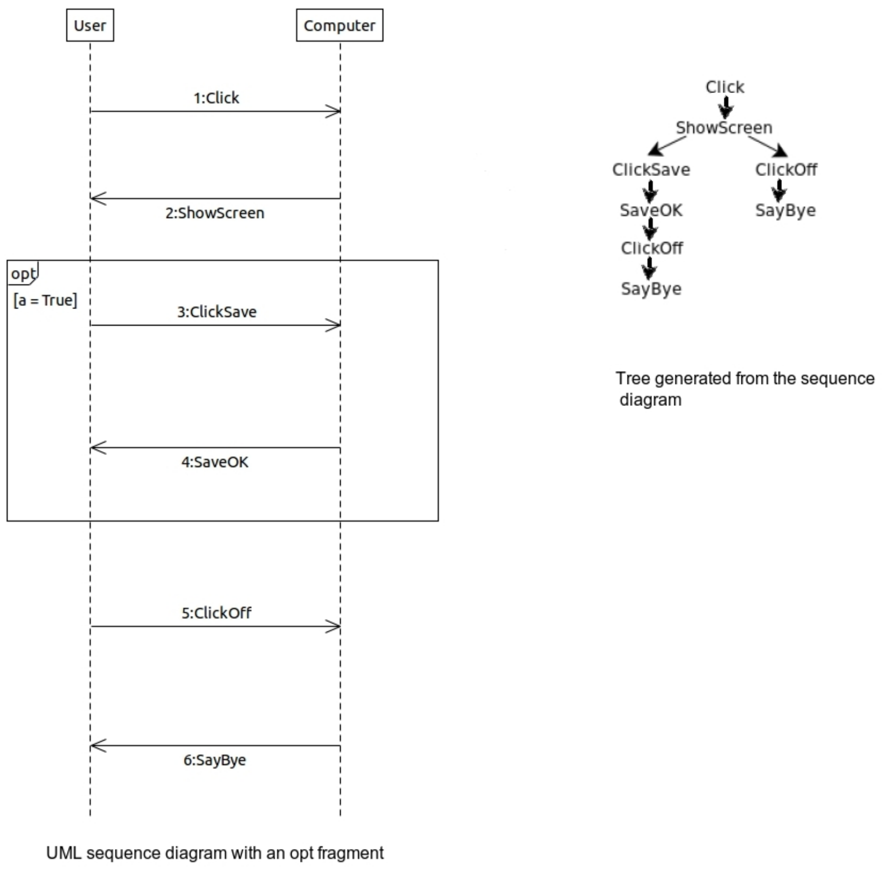


Figure 4.9 - An example of a diagram and its respective tree

The printed version by the tool is a breadth-first search in this Transition System tree. It presents the name of each state, the guard conditions and, in the sequence diagram, who sent and who received each message. Figure 4.9 shows an example to illustrate a diagram and its tree and Figure 4.10 shows the output file for this diagram. Note that the first line of the output file shows the number of children nodes that each node has. Note also that the first state is *null*; the first state is the head of the list. The first state of any diagram is always *null*, because all the data structure used in the output of the application is a list with a head. As the diagram used is a sequence diagram, there is a description of who sent and received the message. In the end, the total number of states identified by the Converter is presented.

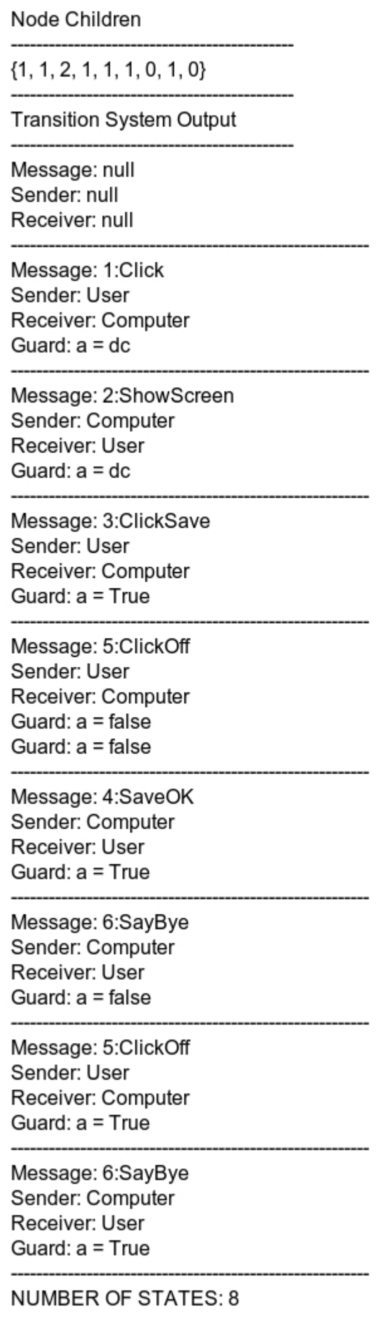
When there is a condition, the tree displays different paths. When the condition is *true*, the stream goes to the left, when it is *false*, it follows to the right, as shown in Figure 4.9. In this example, there is a branch because of the *opt* fragment. In this way, all the possibilities for implementation of the diagrams presented in Section 3.2 are predicted.

4.2 Final Remarks

This chapter has presented XMITS, the tool developed to support three activites proposed in SOLIMVA 3.0 methodology: **Generate single TSs, Generate Unified TS,** and **Generate Model Checker Notation**. XMITS is essential to the application of SOLIMVA 3.0, which aims to allow the use of Formal Methods in practice, to real software projects.

All the modules of XMITS were explained, as well as the operation of each one of them. XMITS is composed of five main modules: the Reader, the Converter, the TUTS, the Bridge, and the Global module. The tool aims to be modular and ready for upgrade. All its architecture was think to be generic and flexible. One of the most important characteristics of the architecture used in XMITS tool is the aperture to implement new modules and add new features in the existent modules.

Appendix C presents step-by-step, how to use the tool. It also shows the class diagrams of Reader, Bridge, and Global modules. The next chapter shows SOLIMVA 3.0 applied to two case studies in the aerospace area.



```
Node Children
--------------------------------------------
{1, 1, 2, 1, 1, 1, 0, 1, 0}
--------------------------------------------
Transition System Output
--------------------------------------------
Message: null
Sender: null
Receiver: null
--------------------------------------------------------
Message: 1:Click
Sender: User
Receiver: Computer
Guard: a = dc
--------------------------------------------------------
Message: 2:ShowScreen
Sender: Computer
Receiver: User
Guard: a = dc
--------------------------------------------------------
Message: 3:ClickSave
Sender: User
Receiver: Computer
Guard: a = True
--------------------------------------------------------
Message: 5:ClickOff
Sender: User
Receiver: Computer
Guard: a = false
Guard: a = false
--------------------------------------------------------
Message: 4:SaveOK
Sender: Computer
Receiver: User
Guard: a = True
--------------------------------------------------------
Message: 6:SayBye
Sender: Computer
Receiver: User
Guard: a = false
--------------------------------------------------------
Message: 5:ClickOff
Sender: User
Receiver: Computer
Guard: a = True
--------------------------------------------------------
Message: 6:SayBye
Sender: Computer
Receiver: User
Guard: a = True
--------------------------------------------------------
NUMBER OF STATES: 8
```

Figure 4.10 - Output file for the diagram shown in Figure 4.9

5 APPLICATION OF SOLIMVA 3.0 TO SPACE SOFTWARE

The goal of this chapter is to present the aplication of SOLIMVA 3.0 and its supporting tool, XMITS, to two real space software products. In total, the methodology was applied to twenty scenarios considering the two case studies.

This chapter is organized as follows. The next section details the first case study: Software for the Payload Data Handling Computer (SWPDC). It shows the project description, its architecture, main objectives, as well as the scenarios explored. The results for each scenario are presented. Section 5.2 shows the same explanation, but related to the Software for the Payload Data Handling Computer - protoMIRAX Experiment (SWPDCpM).

5.1 SWPDC - Software for the Payload Data Handling Computer

This case study is a space application software product, SWPDC, developed in the context of the *Qualidade do Software Embarcado em Aplicações Espaciais* (QSEE - Quality of Space Application Embedded Software) research product (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.

Figure 5.1 shows the physical 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 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 (SANTIAGO JÚNIOR, 2011).

The main functions of SWPDC software product are (SANTIAGO et al., 2007):

a) Interaction with EPPs in order to collect Scientific, Diagnosis, and Test Data;

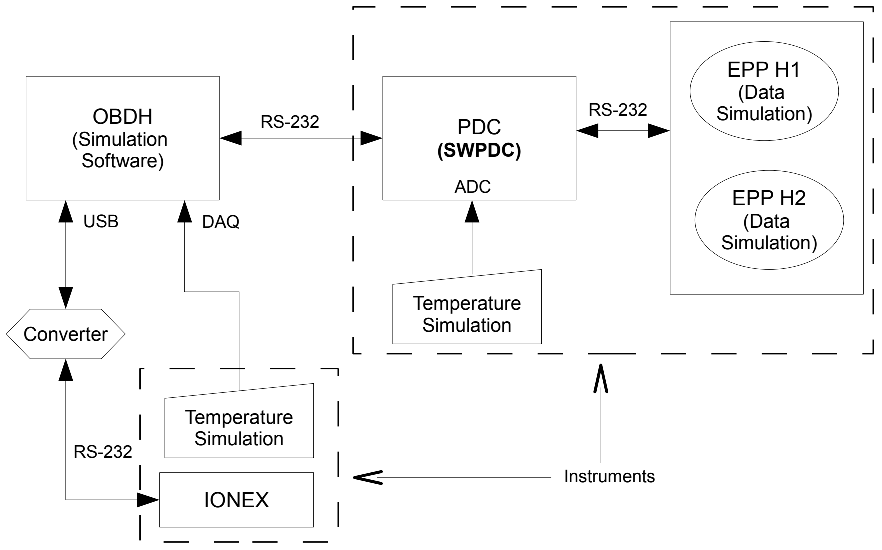


Figure 5.1 - Physical architecture defined for QSEE project. Caption: ADC = Analog-to-Digital Converter; DAQ = Data Acquisition Board; RS-232 = Recommended Standard 232; USB = Universal Serial Bus. SANTIAGO JÚNIOR (2011)

b) Data Formatting;

c) Memory management to store data temporarily before transmission to the OBDH;

d) Implementation of flow control mechanisms;

e) Housekeeping Data generation;

f) Implementation of complex fault tolerance mechanisms; and

g) Loading of new programs of the fly.

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.

In order to apply SOLIMVA 3.0 methodology, two documents were consulted: the Software Requirements Specification of QSEE (SRS), one of the artifacts generated during SWPDC's development lifecycle; and `INPE-16677-RPQ/850` Research Report (JÚNIOR et al., 2010), developed in a partnership between INPE and CIFASIS (*Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas*), whose goal was to accomplish a comparison between Statecharts and Z language (SPIVEY; ABRIAL, 1992), and to propose a complementing approach.

The SRS describes the technical specification of SWPDC, showing the environments, requirements, functional modeling, behavioral modeling, restrictions, and limitations of the software. Behavioral modeling is represented by means of two UML diagrams: sequence diagram and activity diagram. The modeling procedure is given by representation of scenarios. Such scenarios were used as the **scenarios** of the case studies (remembering the first activity of SOLIMVA 3.0: *Identify Scenarios*). The `INPE-16677-RPQ/850` Research Report shows the scenarios proposed by an expert and modeled in Statecharts. Therefore, the two documents together are valuable inputs to apply SOLIMVA 3.0 methodology, since most of the scenarios are specified using sequence diagram, activity diagram, and Statecharts.

As some of the features of Harel's Statecharts are not covered in UML 2.0, the Statecharts models of the Research Report were rewritten as UML state machines. One such example is shown in Figure 5.2. On the left, Scenario 1 is represented using UML state machine. On the right, the same Scenario 1 is represented using Statecharts. Observe that the changes were performed because of the events *tr[In(IniM_-POST)]* and *endtime[In(SafeM_Etered)]*. '*In(IniM_POST)*' predicate returns *true* if the state machine is in the state *IniM_POST*. This predicate allows coordination among parallel regions. As this predicate is not implemented in XMITS, the states from where these transitions start were modeled as parallel with the states *IniM_POST* and *SafeM_Etered*. Actually, that is what happens in practice.

5.1.1 Scenarios of SWPDC

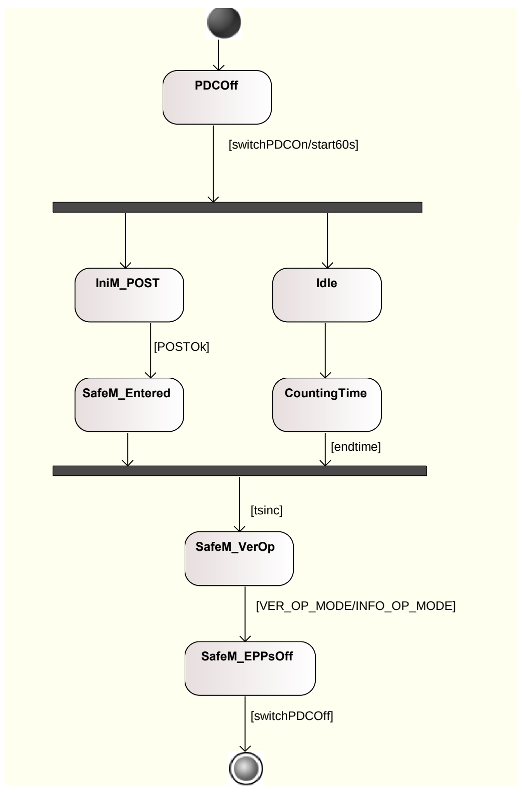
In the case study, eight scenarios of SWPDC were chosen to be carried out. As already mentioned, these scenarios were obtained from SWPDC's SRS, from the INPE-16677-RPQ/850 Research Report, and from (SANTIAGO JÚNIOR, 2011).

a) Scenario 1: PDC initiation process;

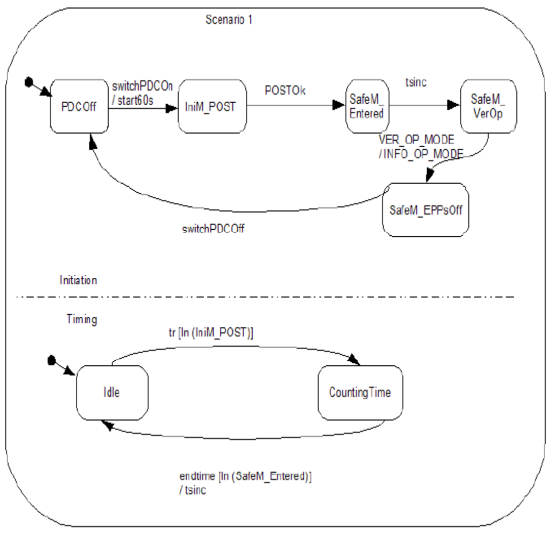
b) Scenario 2: Switching EPP Hxs on and off;

c) Scenario 3: Changing software parameters in the Safety Operation Mode;

d) Scenario 7: Housekeeping Data Transmission in the Nominal Operation Mode;



UML State Machine Model: Scenario 1



Statechart Model: Scenario 1

Figure 5.2 - Scenario 1 represented in Statecharts and adapted to UML State Machine

e) Scenario 8: Housekeeping Data Transmission in the Nominal Operation Mode, Robustness (reception), Load new programs;

f) Scenario 9: Dump Data of Program Memory in the Nominal Operation Mode;

g) Scenario 10: Dump Data of Program Memory in the Nominal Operation Mode, Robustness (command and reception); and

h) Scenario 12: Dump Data of Data Memory (Page 0) in the Nominal Operation Mode, Robustness (command and reception).

In the next subsections scenarios 1, 3, and 8 are detailed. Not all the scenarios are detailed to make text not exhausting. Subsection 5.1.2 summarizes the results obtained from all the scenarios analyzed.

5.1.1.1 Scenario 1: PDC Initiation Process

In this scenario, the main actor is the PCD (Power Conditioning Unit) that switches the PDC computer on. The flow involves: to accomplish hardware verification, to obtain current PDC temperature, to generate startup report, to configure PDC state with standard values, and to divert the control to the main module when in safety operation mode.

There are three UML diagrams which represent Scenario 1: a sequence, an activity, and a state machine diagram, illustrated in Figures 5.3, 5.4, 5.5, respectively. Four user-defined properties for this scenario, related to requirements, were verified.

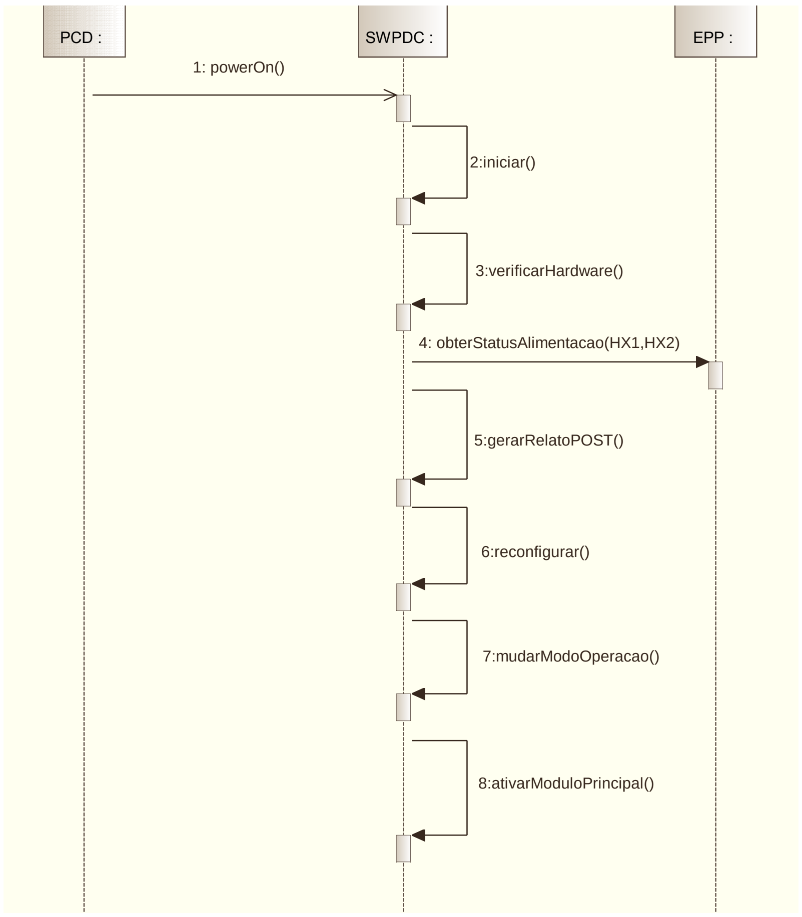
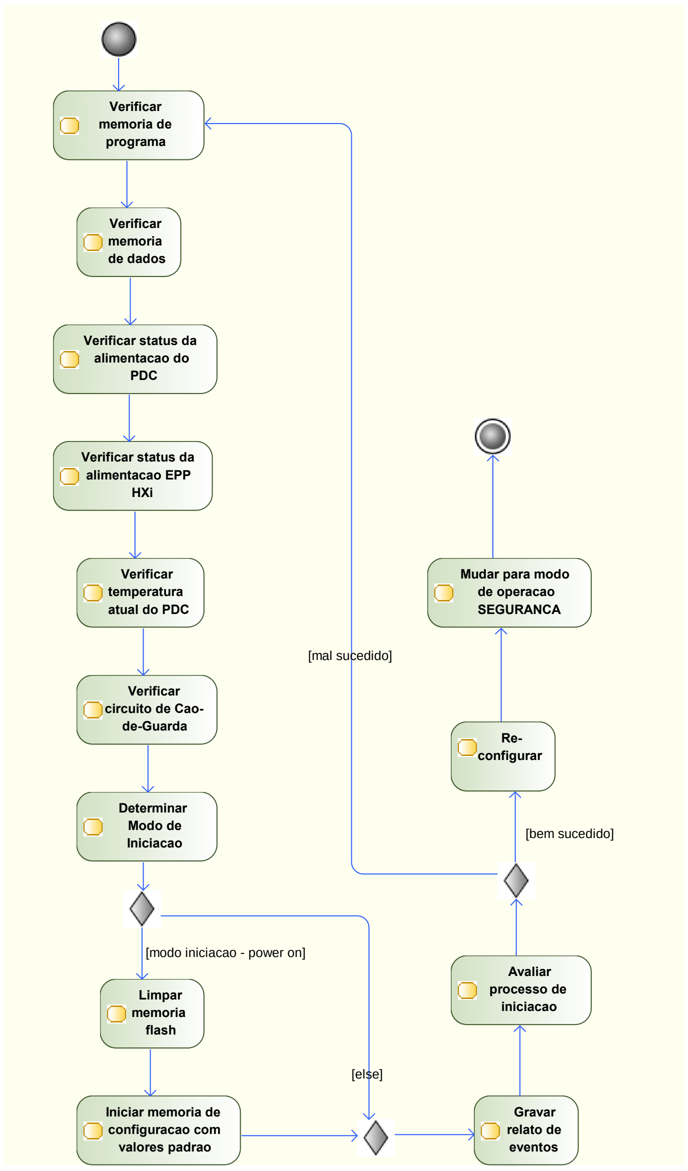


Figure 5.3 - Sequence Diagram for Scenario 1

a) Property 1: *The POST (Power-On Self Test) shall comprise: (i) Power status of the PDC itself; (ii) Power satus of the two EPP-HXI sets; (iii) Current internal temperature of the PDC; (iv) Coherent information of the PDC Program Memory; (v) Reading of SRAM (Data Memory) and;*



Verificar memoria de programa
Verificar memoria de dados
Verificar status da alimentacao do PDC
Verificar status da alimentacao EPP HXi
Verificar temperatura atual do PDC
Verificar circuito de Cao-de-Guarda
Determinar Modo de Iniciacao
[modo iniciacao - power on]
Limpar memoria flash
Iniciar memoria de configuracao com valores padrao
[else]
Gravar relato de eventos
Avaliar processo de iniciacao
[bem sucedido]
[mal sucedido]
Re-configurar
Mudar para modo de operacao SEGURANCA

Figure 5.4 - Activity Diagram for Scenario 1

(vi) Correct operation of the watchdog timer circuit. This property can be formalized using the **Existence Pattern and Globally Scope** proposed by (DWYER et al., 1999), in CTL, as follows:

```
∀◊  (State=_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--
                CountingTimeANDIniM_POST)
∀◊  (State=_4obterStatusAlimentacao$HX1-HX2$--
                VerificarstatusdaalimentacaoEPPHXi--IniM_POST)
∀◊  (State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--
                SafeM_Entered)
```

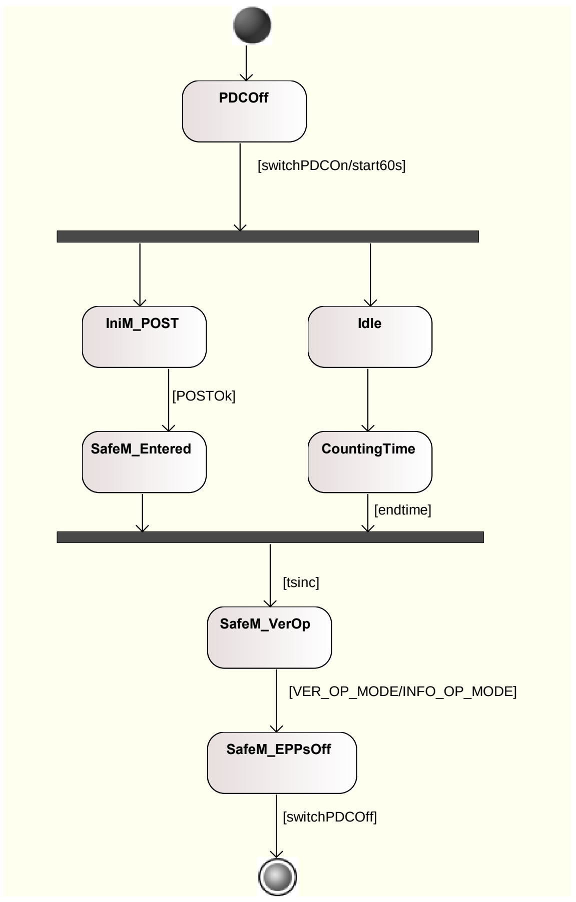


Figure 5.5 - State Machine Diagram for Scenario 1

```
∀◊ (State=_1powerOn$$--Verificarmemoriadeprograma--PDCOff)
∀◊ (State=_2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST)
∀◊ (State=_6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_VerOp)
```

A state is composed of `(Message,Activity,Event-State)`, as detailed in Section 3.2.2. However, some adjustments had to be made to be compatible with NuSMV notation. For example, observe the state `_3verificarHardware$$-Verificarstatusdaalimentacaodo PDC-CountingTimeANDIniM_POST`. Its first part, `_3verificarHardware$$` is inherited from the sequence diagram (`3:VerificarHardware()`). Observing the grammar presented in Figure 3.12, it is possible to see that the first character of a variable can not be a number: only letters and character '_' (underscore) are allowed. That's why all states begin with an '_', as the messages are all numbered. Also, one can observe that name of variables can be formed by letters, numbers, and only the digits '$', '#',

'-'. So, () was replaced by $$, and ':' (colon) was removed. The comma, which was used for separating messages, activities, and event-states, was replaced by '–' (two dashes). Besides, all blank spaces were removed. Thus, `VerificarstatusdaalimentacaodoPDC` was inherited from the activity diagram, originally was `Verificar status da alimentacao do PDC` and `CountingTimeANDIniM_POST` was inherited from the state machine diagram. Missing diagrams are substituted by an underscore '_'.

Each one of the six items related to property 1 are represented by one state in the TS obtained from the diagrams of Figures 5.3, 5.4, 5.5. For example, *Coherent information of the PDC Program Memory* is represented by the state *_1powerOn$$–Verificarmemoriadeprograma–PDCOff*. After running NuSMV, this property is satisfied.

b) Property 2: *The SWPDC should know how to distinguish between a power-on process and a reset process.* This property can be formalized using the **Existence Pattern and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

```
¬∃ [¬ ((State= _7mudarModoOperacao$$--DeterminarMododeIniciacao--
     SafeM_EPPsOff ∧ modoIniciacaoPowerOn=true) ∧
∀◊ (State = _--Limparmemoriaflash--_)) ∪ ((State= _7mudarModoOperacao$$--
    DeterminarMododeIniciacao--SafeM_EPPsOff ∧
 modoIniciacaoPowerOn=true)
∧¬ ((State= _7mudarModoOperacao$$--DeterminarMododeIniciacao--
    SafeM_EPPsOff ∧ modoIniciacaoPowerOn=true) ∧ ∀◊ (State=
    _--Limparmemoriaflash--_)))]
```

The TS obtained from the diagrams of Figures 5.3, 5.4, 5.5 has a variable (obtained from the guards) that distinguish between a power-on process and a reset process (*modoIniciacaoPowerOn*). After the state `_7mudarModoOperacao$$--DeterminarMododeIniciacao--SafeM_EPPOff` if the value of variable *modoIniciacaoPowerOn=true* the next state should be `_--Limparmemoriaflash--_`. When running NuSMV, this property is satisfied.

c) Property 3: *The SWPDC should report processing of the POST through reports of events.* This property can be formalized using the **Existence Pattern and Globally Scope** proposed by (DWYER et al., 1999), in CTL, as follows:

```
∀◊ (_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered)
```

```
∨ ∀◊ (_--Gravarrelatodeeventos--_)
```

Two states on the TS represent this property: `_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered` and also the state `_--Gravarrelatodeeventos--_`. When running NuSMV, this property is satisfied.

d) Property 4: *In the case of any unrecoverable problem not being identified in the PDC after the initiation process, the PDC shall automatically enter into the safety operation mode.* This property can be formalized using the **Response Pattern and Globally Scope** proposed by (DWYER et al., 1999), in CTL, as follows:

```
∀□ ((State=_--Avaliarprocessodeiniciacao--_ ∧ bemSucedido=true) ->
∀◊ (State=_--MudarparamododeoperacaoSEGURANCA--_))
```

In the TS there is a variable that indicates if the initiation process is successful (*bemSucedido=true*) or not (*bemSucedido=false*). After state `_--Avaliarprocessodeiniciacao--_`, in the case of *bemSucedido=true*, it is possible to reach state `_--MudarparamododeoperacaoSEGURANCA--_`, which represents that the PDC enters the safety operation mode. Otherwise, the PDC remains in the initiation operation mode. When running NuSMV, this property is satisfied.

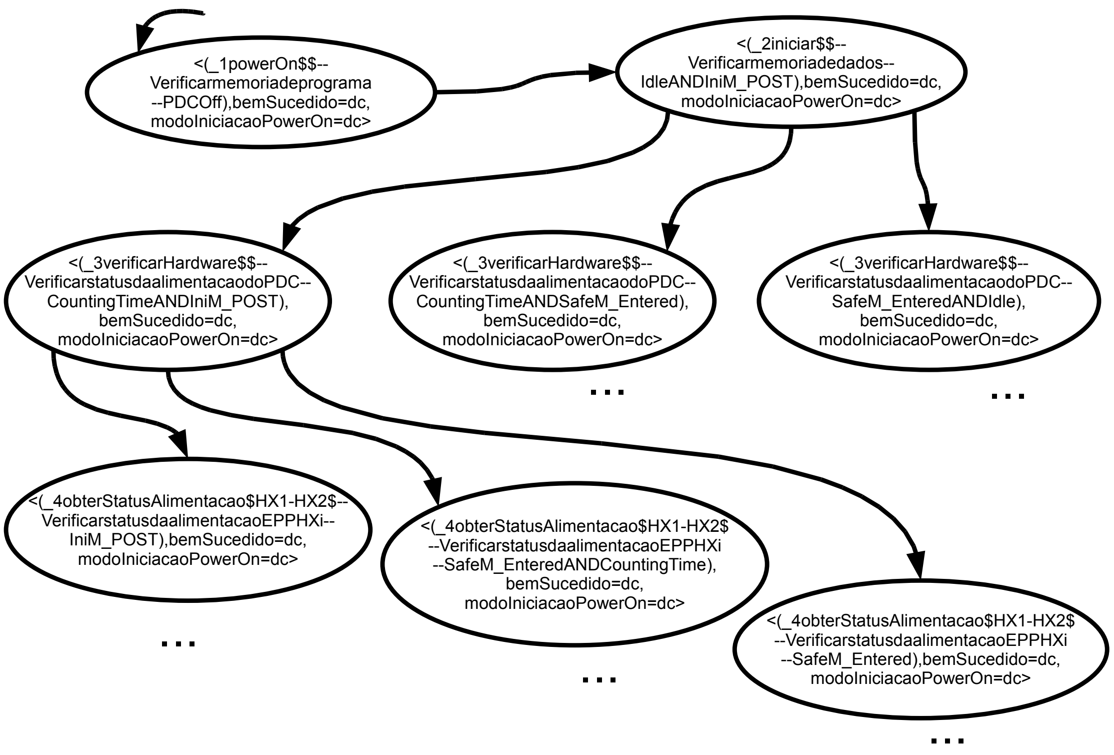
For Scenario 1, 243 states were obtained, 31 of which are reachable states when running NuSMV model checker. Table 5.1 summarizes all information about this scenario. The table is divided into three parts. From left to right: first, it shows the number of each type of UML diagram that was used for this scenario; then, the total number of states and reachable states; and last, the properties verified and which of them have been satisfied. Figure 5.6 exhibits part of the unified TS for Scenario 1. The TS generated as *txt* file as well as the output containing the model checker notation (smv file) for this scenario can be found in Appendix A. Table A.1, also in Appendix A, presents all properties which were verified for all scenarios of SWPDC.

5.1.1.2 Scenario 3: Changing software parameters in the Safety Operation Mode

In this scenario, the modification of SWPDC parameters, may occur in any mode of operation (except on initiation). SWPDC verifies the limits of the values that the

Table 5.1 - Statistics about Scenario 1 of SWPDC

UML diagrams			Transition System		Properties Verified	
SD	AD	SMD	Total States	Reachable States	Property	Satisfied
1	1	1	243	31	P1	Yes
					P2	Yes
					P3	Yes
					P4	Yes



<(_1powerOn$$--Verificarmemoriadeprograma--PDCOff),bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(_2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST),bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(_3verificarHardware$$--Verificarstatusdaalimentacaodo PDC--CountingTimeANDIniM_POST), bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(_3verificarHardware$$--Verificarstatusdaalimentacaodo PDC--CountingTimeANDSafeM_Entered), bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(_3verificarHardware$$--Verificarstatusdaalimentacaodo PDC--SafeM_EnteredANDIdle), bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--IniM_POST),bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime), bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered),bemSucedido=dc, modoIniciacaoPowerOn=dc>
...
...
...
...
...

Figure 5.6 - Part of the unified TS obtained for Scenario 1

parameter in question can assume and proceeds with the change of values in the configuration table.

There are two UML diagrams which represent Scenario 3: a sequence and a state machine diagram, as can be seen in Figures 5.7, 5.8, respectively. Two user-defined properties for this scenario, related to requirements, were verified.

a) Property 1: *The time for generation of housekeeping data can be changed by means of command sent by OBDH. The minimum value of time to generate housekeeping is 60s, and and the maximum value is 1000s.*

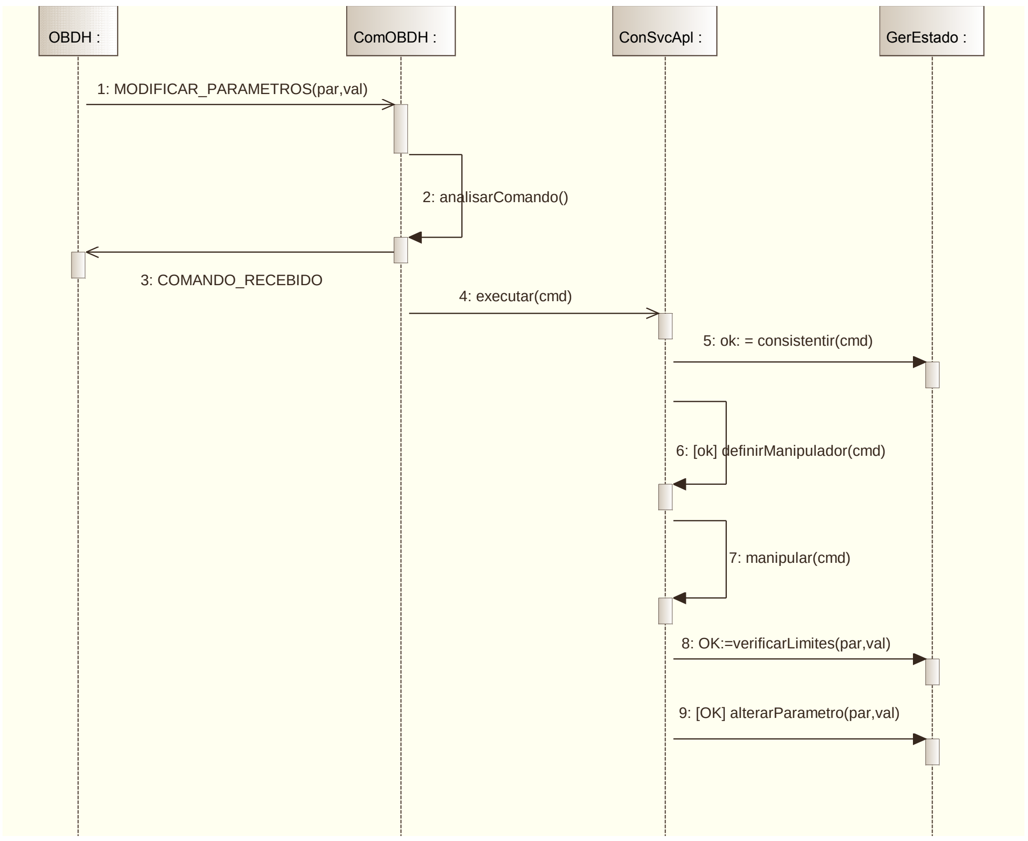

OBDH :
ComOBDH :
ConSvcApl :
GerEstado :
1: MODIFICAR_PARAMETROS(par,val)
2: analisarComando()
3: COMANDO_RECEBIDO
4: executar(cmd)
5: ok: = consistentir(cmd)
6: [ok] definirManipulador(cmd)
7: manipular(cmd)
8: OK:=verificarLimites(par,val)
9: [OK] alterarParametro(par,val)

Figure 5.7 - Sequence Diagram for Scenario 3

This property can be formalized using the **Universality Pattern and Globally Scope** proposed by (DWYER et al., 1999), in CTL, as follows:

```
∀□ (ModifyParameters(HK,60s))
∀□ (ModifyParameters(HK,1000s))
```

The state that represents `ModifyParamtParameters(HK,60s)` is `_-$2$dots$__analisarcomando$_$$_$$coma$__ch_sw_par$minus$hk_-60s$slash$cmd_rec__$and$__safem_parhkr$_$`, but there is no state in the TS that represents `ModifyParameters(HK,1000s)`. So, the property is not satisfied.

b) Property 2: *A session for transmission of housekeeping data begins with the command PREPARE HOUSEKEEPING DATA. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT HOUSEKEEPING DATA (RETRANSMIT RESPONSE, when necessary) and ends with the command TRANSMIT SCIENTIFIC DATA, where SWPDC returns to acquire scientific data of EPPs.* This property can be formalized using the

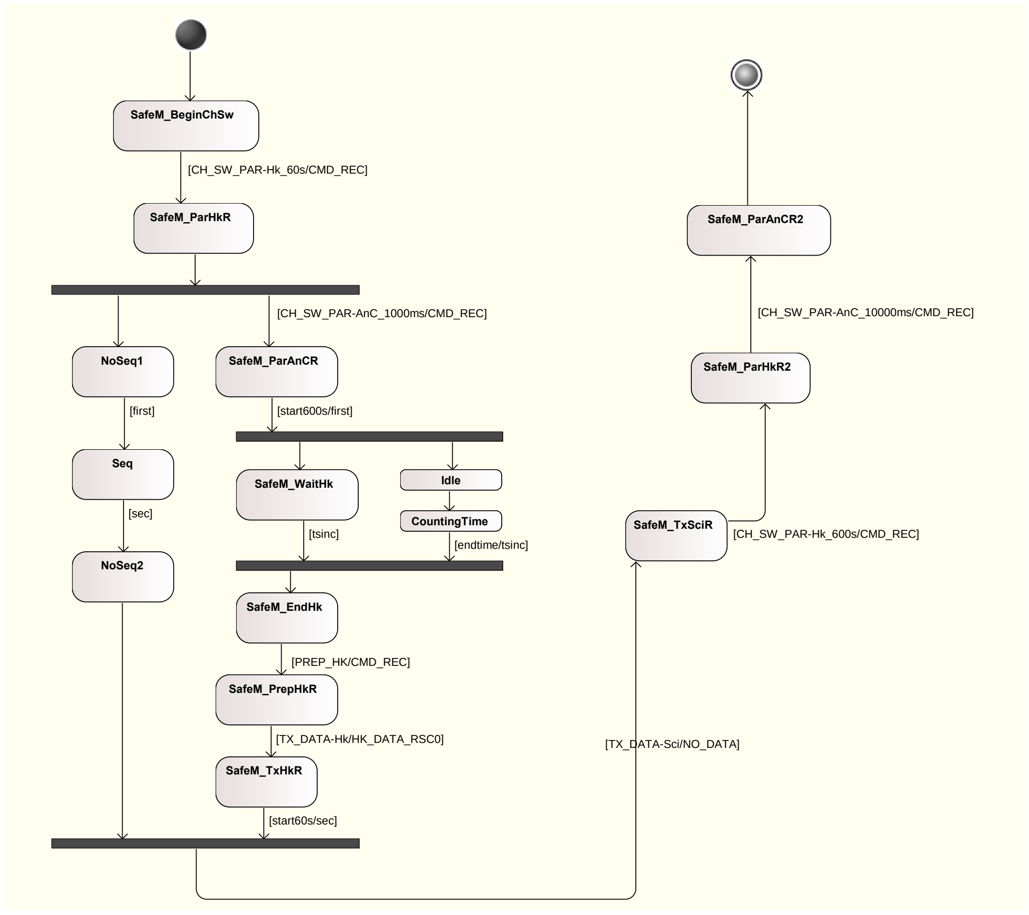


Figure 5.8 - State Machine Diagram for Scenario 3

Precedence Chain and Scope After Q proposed by (DWYER et al., 1999), in CTL, as follows:

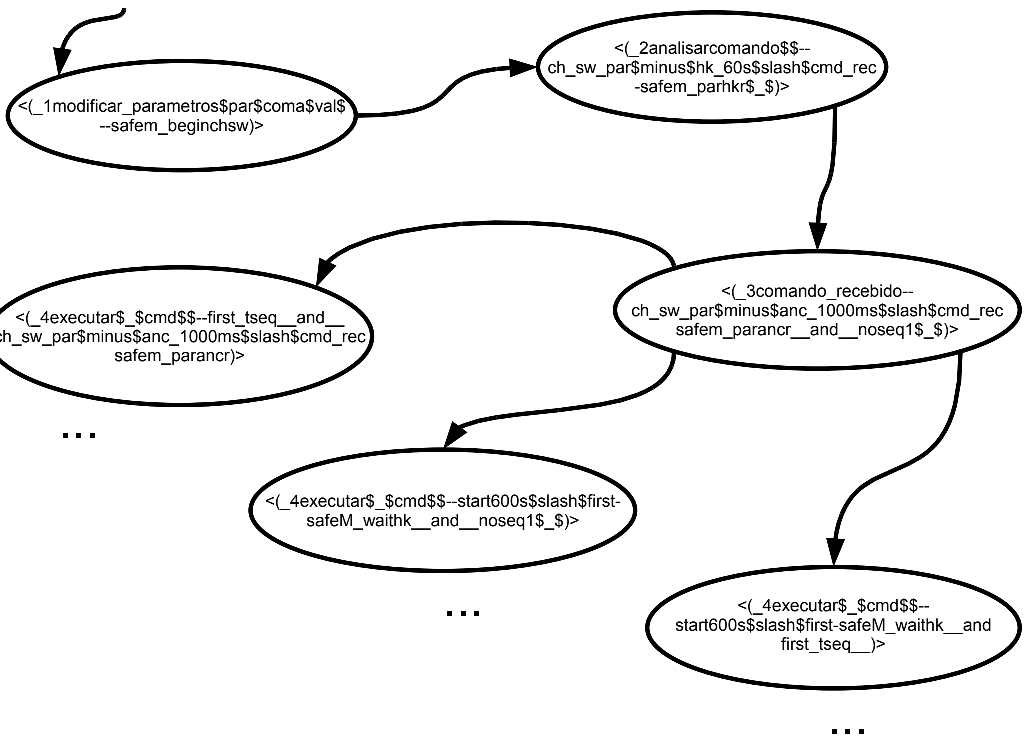
¬∃[¬ Q ∪ (Q ∧∃ [¬S ∪ P] ∧∃ [¬P ∪ (S ∧ ¬P ∧ ∃◯ (∃[¬T ∪ (P ∧ ¬T)]))])]
where,
Q = PREPARE HOUSEKEEPING DATA
S = STOP DATA ACQUISITION
T = TRANSMIT HOUSEKEEPING DATA
P = TRANSMIT SCIENTIFIC DATA

There are no states in the TS that represent the commands Q, S, T or P. The property is not satisfied.

For Scenario 3, it was obtained 3745 states, 63 of which are reachable states when running NuSMV model checker. Table 5.2 summarizes all information about this scenario. Figure 5.9 exhibits part of the unified TS for Scenario 3.

Table 5.2 - Statistics about Scenario 3 of SWPDC

UML diagrams			Transition System		Properties Verified	
SD	AD	SMD	Total States	Reachable States	Property	Satisfied
1	0	1	3745	63	P1	No
					P2	No



<(_1modificar_parametros$par$coma$val$ --safem_beginchsw)>
<(_2analisarcomando$$-- ch_sw_par$minus$hk_60s$slash$cmd_rec -safem_parhkr$_$)>
<(_3comando_recebido-- ch_sw_par$minus$anc_1000ms$slash$cmd_rec safem_parancr__and__noseq1$_$)>
<(_4executar$_$cmd$$--first_tseq__and__ ch_sw_par$minus$anc_1000ms$slash$cmd_rec safem_parancr)>
...
<(_4executar$_$cmd$$--start600s$slash$first- safeM_waithk__and__noseq1$_$)>
...
<(_4executar$_$cmd$$-- start600s$slash$first-safeM_waithk__and first_tseq__)>
...

Figure 5.9 - Part of the unified TS obtained for Scenario 3

5.1.1.3 Scenario 8: Housekeeping Data Transmission in the Nominal Operation Mode, Robustness (reception), Load new programs

In this scenario, during a sequence of commands for transmission of data, the OBDH may request the retransmission of the last reply to a request for data. In this case, the command handler defined by the application service control will be responsible for acquiring and conditioning the response message to retransmission, working with data management.

There are four UML diagrams which represent Scenario 8: three sequence diagrams and one state machine diagram, as can be seen in Figures 5.10, 5.11, 5.12, 5.13, respectively. Four user-defined properties for this scenario, related to requirements, were verified.

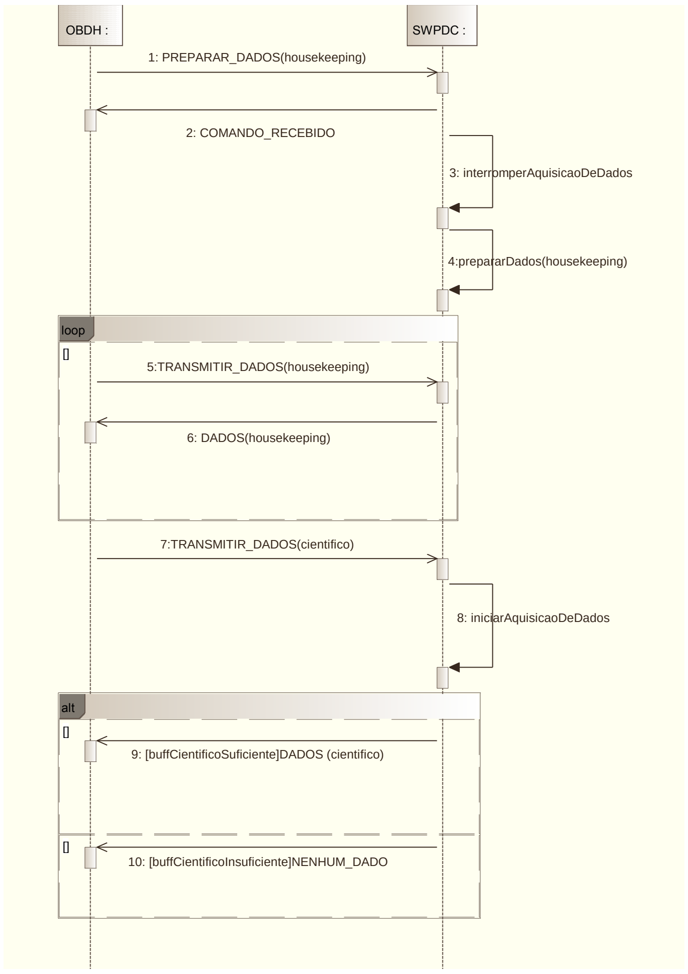

OBDH :
SWPDC :
1: PREPARAR_DADOS(housekeeping)
2: COMANDO_RECEBIDO
3: interromperAquisicaoDeDados
4:prepararDados(housekeeping)
loop
[]
5:TRANSMITIR_DADOS(housekeeping)
6: DADOS(housekeeping)
7:TRANSMITIR_DADOS(cientifico)
8: iniciarAquisicaoDeDados
alt
[]
9: [buffCientificoSuficiente]DADOS (cientifico)
[]
10: [buffCientificoInsuficiente]NENHUM_DADO

Figure 5.10 - Sequence Diagram 1 for Scenario 8

a) Property 1: *A session for transmission of housekeeping data begins with the command PREPARE HOUSEKEEPING DATA. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT HOUSEKEEPING DATA (RETRANSMIT RESPONSE, when necessary) and ends with the command TRANSMIT SCIENTIFIC DATA, where SWPDC returns to acquire scientific data of EPPs.* This property can be formalized using the **Precedence Chain and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$

where,

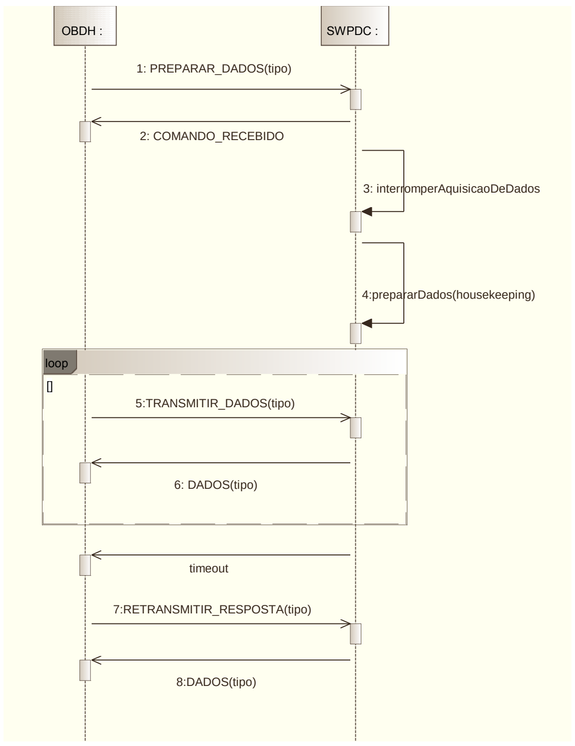


Figure 5.11 - Sequence Diagram 2 for Scenario 8

Q = PREPARE HOUSEKEEPING DATA
S = STOP DATA ACQUISITION
T = TRANSMIT HOUSEKEEPING DATA
P = TRANSMIT SCIENTIFIC DATA

Table 5.3 maps the commands and how they are represented in the TS. When running NuSMV, this property is satisfied.

b) Property 2: *The PDC can also have a situation of not receiving the command sent. After identifying delimiter of start the PDC waits for a time of 600 ms for the rest of the command. This time is equivalent to 2 times as long to transmit command with maximum size (1128 Bytes). In the case of expiry of the stipulated time, a timeout occurs and the PDC aborts the communication, the command is discarded, one event is raised to it, and the PDC expects a new command from OBDH.* This property can be formalized using the **Precedence Chain and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
where,
Q = StartCommand & Wait600ms
S = Timeout

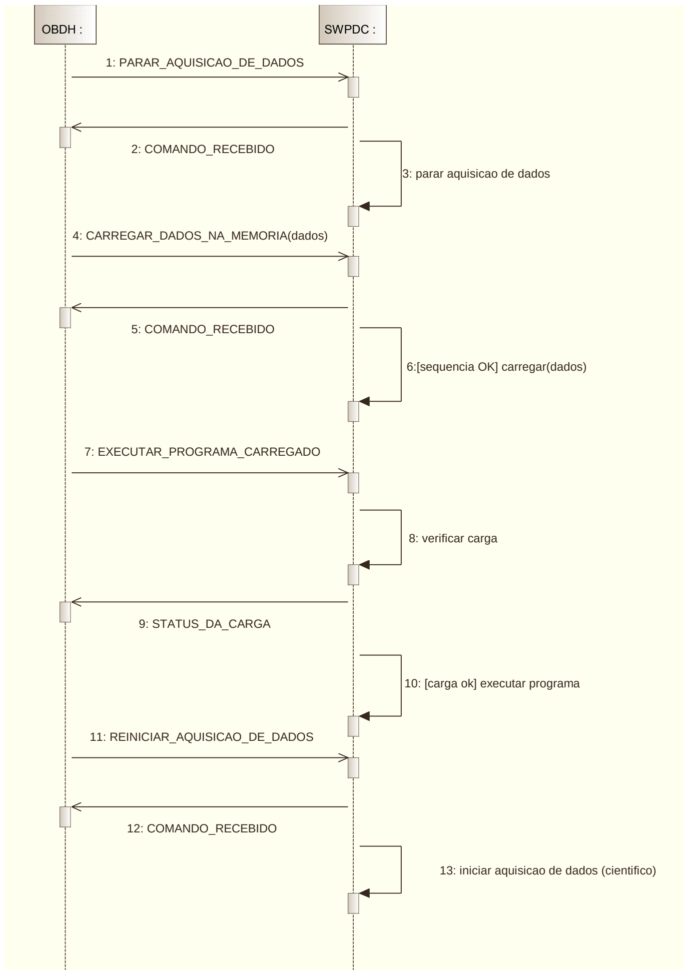

OBDH :
SWPDC :
1: PARAR_AQUISICAO_DE_DADOS
2: COMANDO_RECEBIDO
3: parar aquisicao de dados
4: CARREGAR_DADOS_NA_MEMORIA(dados)
5: COMANDO_RECEBIDO
6:[sequencia OK] carregar(dados)
7: EXECUTAR_PROGRAMA_CARREGADO
8: verificar carga
9: STATUS_DA_CARGA
10: [carga ok] executar programa
11: REINICIAR_AQUISICAO_DE_DADOS
12: COMANDO_RECEBIDO
13: iniciar aquisicao de dados (cientifico)

Figure 5.12 - Sequence Diagram 3 for Scenario 8

T = GenerateReportEvent
P = ExpectNewCommandOBDH

There is no state in the TS generated that represents the command `GenerateReportEvent`. Thus, the property is not satisfied.

c) Property 3: *The OBDH checks all fields of the replies sent by the PDC. In the case of OBDH send a command TRANSMIT DATA and if there is inconsistency in the values received in any of the fields of a data response (SCIENTIFIC DATA, HOUSEKEEPING DATA, DUMP DATA MEMORY, DIAGNOSTIC DATA, AND TEST DATA), the OBDH uses the command RETRANSMIT RESPONSE to try get a consistent data response. The command RETRANSMIT RESPONSE refers only to the last data response sent by the PDC. Therefore, the PDC must maintain, temporarily, always the latest data response that was sent to the OBDH for case of error in receipt of this response by*

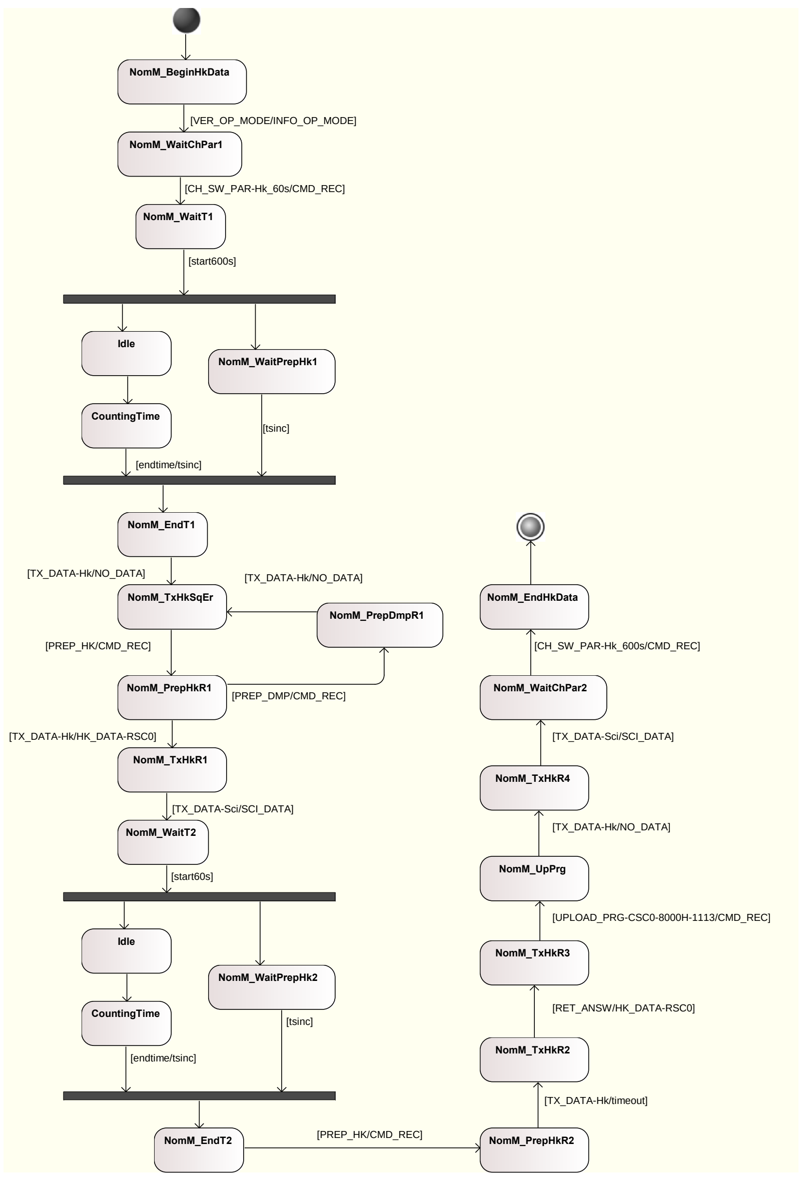


Figure 5.13 - State Machine Diagram for Scenario 8

the OBDH. The OBDH sends this same command for maximum two more times. If after these other two attempts still there is problem in the response received from the PDC, the OBDH does not transmit this command and generates a bug report which should be sent to the ground station. This property can be formalized using the **Precedence Chain and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

$\neg\exists[\neg\ Q \cup (Q \wedge\exists\ [\neg S \cup P] \wedge\exists\ [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc\ (\exists[\neg T \cup (P \wedge \neg T)]))])]$

where,

Q = TRANSMIT DATA & PDCResponseProblem

Table 5.3 - Commands and its representation on the TS

Command	TS element	Name and/or Value
PREPARE HOUSEKEEPING DATA	Variable State	_$_$1$dots$___preparar_dados$ _$housekeeping$_$$coma$___1$dots$_ _preparar_dados$_$tipo$_$$coma$_ _1$dots$___parar_aquisicao_de_dados $coma$___nomm_beginhkdata$_$
STOP DATA ACQUISITION	Variable State	_$_$3$dots$___interromperaquisicaodedados$ coma$___3$dots$___interromperaquisicaodedados $coma$___3$dots$___pararaquisicaodedados $coma$___ch_sw_par$minus$hk_60s $slash$cmd_rec___$and$___nomm_waitt1$_$
TRANSMIT HOUSEKEEPING DATA	Variable State	_$_$5$dots$transmitir_dados$_$housekeeping $_$$coma$___5$dots$transmitir_dados$_$tipo $_$$coma$___4$dots$carregar_dados_na _memoria$_$dados$_$$coma$___$minus$$_$
TRANSMIT SCIENTIFIC DATA	Variable State	_$_$7$dots$transmitir_dados$_$cientifico $_$$coma$___timeout$coma$___6$dots$# sequenciaok#carregar$_$dados$_$$coma$ ___$minus$$_$

S = RETRANSMIT RESPONSE 1 & PDCResponseProblem
T = RETRANSMIT RESPONSE 2 & PDCResponseProblem
P = Stop Sending & GenerateReportEvent

There is no state in the TS generated that represents the command `Stop Sending & GenerateReportEvent`. Thus, the property is not satisfied.

d) Property 4: *The SWPDC should allow uploading of programs in this operation mode. The first command to be sent to the SWPDC is STOP DATA ACQUISITION, for the SWPDC interrupts scientific data acquisition of the EPPs. After this command, one or more commands to LOAD DATA MEMORY should be sent to the new program to be loaded in the data memory of PDC. Later, a command RUN PROGRAM LOADED INTO MEMORY should be sent to the PDC to have the program loaded to run. At the end of the process of loading and executing of the program, a command RESTART DATA ACQUISITION causes the SWPDC return to acquire scientific data of EPPs.* This property can be formalized using the **Precedence Chain and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
where,
Q = STOP DATA ACQUISITION
S = LOAD DATA MEMORY
T = RUN PROGRAM LOADED

P = RESTART DATA ACQUISITION

Table 5.4 maps the commands and how they are represented in the TS. When running NuSMV, this property is satisfied.

Table 5.4 - Commands and its representation on the TS

Command	TS element	Name and/or Value
STOP DATA ACQUISITION	Variable State	_$_$3$dots$___interromperaquisicaodedados $coma$___3$dots$___interromperaquisicaodedados $coma$___3$dots$___pararaquisicaodedados $coma$___ch_sw_par$minus$hk_60s$slash$ cmd_rec___$and$___nomm_waitt1$_$
LOAD DATA MEMORY	Variable State	_$_$5$dots$transmitir_dados$_$housekeeping $_$$coma$___5$dots$transmitir_dados$_$tipo$_$ $coma$___4$dots$carregar_dados_na_memoria $_$dados$_$$coma$___$minus$$_$
RUN PROGRAM LOADED	Variable State	_$_$8$dots$___iniciaraquisicaodedados$coma$ ___7$dots$retransmitir_resposta$_$tipo$_$ $coma$___7$dots$___executar_programa_carregado $coma$___$minus$$_$
RESTART DATA ACQUISITION	Variable State	_$_$8$dots$___iniciaraquisicaodedados$coma$ ___8$dots$dados$_$tipo$_$$coma$___11$dots$ reiniciar_aquisicao_de_dados$coma$___$minus$$_$

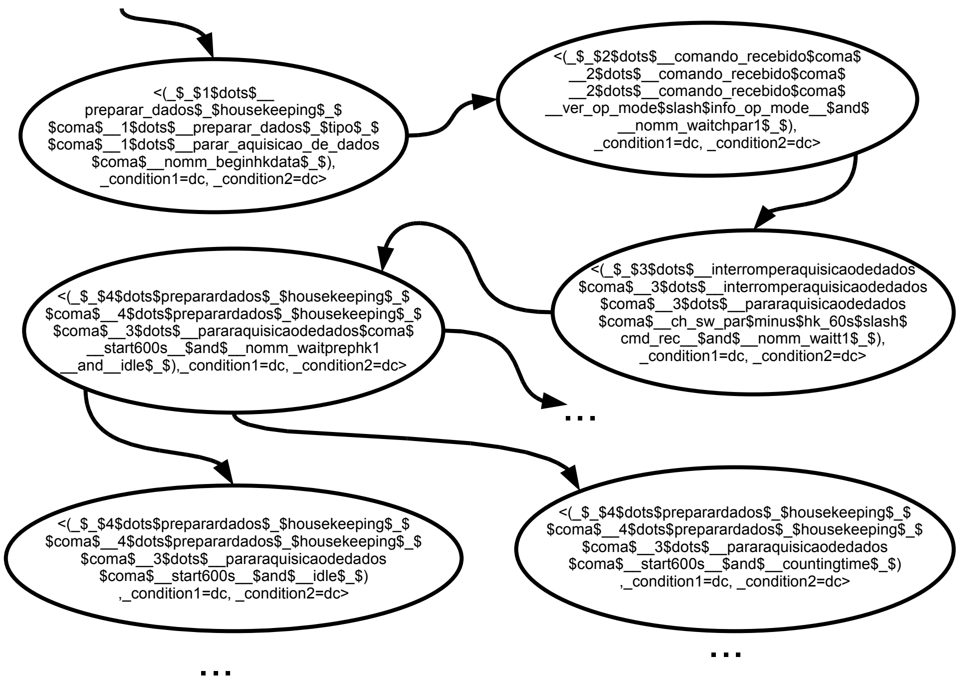
For Scenario 8, it was obtained 396 states, of which 50 are reachable states when running NuSMV model checker. Table 5.5 summarizes all information about this scenario. Figure 5.14 exhibits part of the unified TS for Scenario 8.

Table 5.5 - Statistics about Scenario 8 of SWPDC

UML diagrams			Transition System		Properties Verified	
SD	AD	SMD	Total States	Reachable States	Property	Satisfied
3	0	1	396	50	P1	Yes
					P2	No
					P3	No
					P4	Yes

5.1.2 Summary of the results for SWPDC case study

Table 5.6 summarizes the results of all scenarios of SWPDC. It is possible to view the number of UML diagrams used for each one of the scenarios. Another information is the number of states (total and reachable), as well as the number of properties



<(_$_$1$dots$__ preparar_dados$_$housekeeping$_$ $coma$__1$dots$__preparar_dados$_$tipo$_$ $coma$__1$dots$__parar_aquisicao_de_dados $coma$__nomm_beginhkdata$_$), _condition1=dc, _condition2=dc>
<(_$_$2$dots$__comando_recebido$coma$ __2$dots$__comando_recebido$coma$ __2$dots$__comando_recebido$coma$ __ver_op_mode$slash$info_op_mode__$and$ __nomm_waitchpar1$_$), _condition1=dc, _condition2=dc>
<(_$_$3$dots$__interromperaquisicaodedados $coma$__3$dots$__interromperaquisicaodedados $coma$__3$dots$__pararaquisicaodedados $coma$__ch_sw_par$minus$hk_60s$slash$ cmd_rec__$and$__nomm_waitt1$_$), _condition1=dc, _condition2=dc>
<(_$_$4$dots$preparardados$_$housekeeping$_$ $coma$__4$dots$preparardados$_$housekeeping$_$ $coma$__3$dots$__pararaquisicaodedados$coma$ __start600s__$and$__nomm_waitprephk1 __and__idle$_$),_condition1=dc, _condition2=dc>
...
<(_$_$4$dots$preparardados$_$housekeeping$_$ $coma$__4$dots$preparardados$_$housekeeping$_$ $coma$__3$dots$__pararaquisicaodedados $coma$__start600s__$and$__idle$_$) ,_condition1=dc, _condition2=dc>
<(_$_$4$dots$preparardados$_$housekeeping$_$ $coma$__4$dots$preparardados$_$housekeeping$_$ $coma$__3$dots$__pararaquisicaodedados $coma$__start600s__$and$__countingtime$_$) ,_condition1=dc, _condition2=dc>
...
...

Figure 5.14 - Part of the unified TS obtained for Scenario 8

that were satisfied and not satisfied.

Table 5.6 - Summary of the results of the eight scenarios analyzed for SWPDC

	UML			Transition System		Properties	
Scenarios	SD	AD	SMD	Total States	Reachable States	Satisfied	Not Satisfied
Scenario 1	1	1	1	243	31	4	0
Scenario 2	2	0	0	14	14	1	0
Scenario 3	1	0	1	3745	63	0	2
Scenario 7	1	1	1	3564	54	1	0
Scenario 8	3	0	1	396	50	2	2
Scenario 9	1	1	1	1377	36	1	0
Scenario 10	2	1	1	8262	99	3	0
Scenario 12	2	1	1	8262	99	3	0

As it is possible to see, some properties were not satisfied, meaning that some diagrams do not reflect all the requirements. The main reason for the properties have not been met, was the absence of states in the TSs that represent certain commands/sentences of the properties. This case study was very valuable, because by means of the various scenarios, it has enabled to exercise the whole features of

SOLIMVA 3.0, which include unifying three or more UML diagrams.

5.2 SWPDCpM - Software for the Payload Data Handling Computer - protoMIRAX experiment

The protoMIRAX is a scientific instrument for observations of x-ray cosmic sources, placed on a stratospheric balloon platform. The main component of protoMIRAX is a camera or, x-ray telescope (CRX). The protoMIRAX balloon gondola will house the X-ray camera and the various subsystems of the space segment, including the On-Board Data Handling Subsystem (OBDH), the Attitude Control and Pointing Subsystem (ACS), the Flight Control and Telecommunications Subsystem (FCTS), and the Power Supply Subsystem (PSS). Also, there will be a main GPS receiver that will provide universal time to all subsystems and two star cameras that will be part of the control loop (BRAGA et al., 2015).

The On-Board Data Handling Subsystem (OBDH; In Portuguese: Subsistema de Gestão de Bordo - SGB) processes the information received from the Ground station, as well as gets, generates, formats, stores, and transmits to the Ground station, via FCTS, information of the subsystems of the experiment protoMIRAX. The main hardware unit of OBDH is the Payload Data Handling Computer (PDCpM), which is the on-board computer of OBDH. It is a PC/104 ultra-low-power AMD Geode LX computer. It operates on 333 MHz, has a 128 MB SDRAM, and has interfaces in addition to analog-digital/digital-analog converters. The software embedded in the computer (SWPDCpM) has been developed in C language over the real-time operating system RTEMS. The architecture of the SWPDCpM is composed of three layers where the bottom layer is the basic software, the intermediate layer is the flight software library in which there are basic services and reusable components, and the top layer is the application software, the main part of SWPDCpM (BRAGA et al., 2015). Figure 5.15 shows a simplified physical architecture of the protoMIRAX experiment with its main subsystems.

In order to apply SOLIMVA 3.0 methodology, the document consulted was the OBDH's (SGB's) Software Requirements Specification. This is indeed the SWPD-CpM's SRS which presents definitions, system environment, software logical model, interfaces, and requirements description, and behavioral modeling by means of UML sequence diagrams.

In this SRS, one can note that SWPDCpM's logical model divides the software into the following application processes which are in fact the main actors of SWPDCpM:

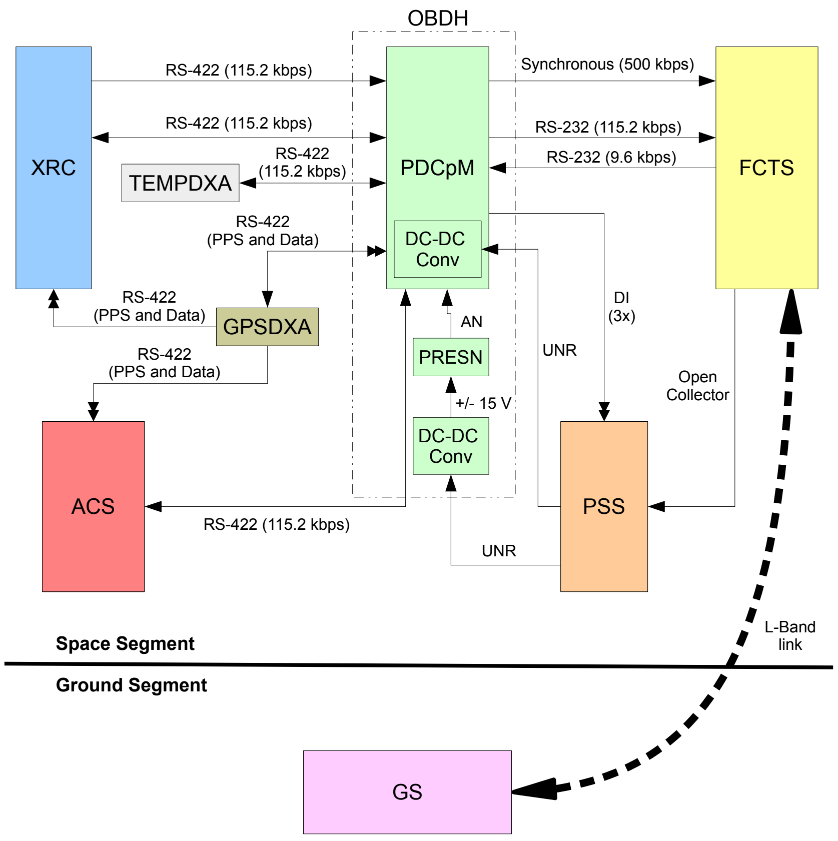


Figure 5.15 - Simplified physical architecture of the protoMIRAX balloon experiment. XRC: X-ray camera; ACS: Attitude Control System; TEMPDXA: Temperature monitoring equipment; GPSDXA: GPS unit; OBDH: On-Board Data Handling Subsystem; PDCpM: Payload Data Handling Computer; DC-DC Conv: DC-DC Converter; PRESN: Pressure Sensor; PSS: Power Supply Subsystem; FCTS: Flight Control and Telecommunications Subsystem. Braga et al. (2015)

a) **CTL:** SWPDCpM general control and transfer of packages;

b) **HK:** Internal *housekeeping* data collection of SWPCpM;

c) **SYN:** Clock synchronization and maintenance of mission time;

d) **SCA:** Interface control of data packets with SCA;

e) **CRX:** Interface control of data packets with CRX;

f) **AP:** Generic application process, both on board and in the ground;

g) **ESS:** An application process of the ground station.

5.2.1 Scenarios of SWPDCpM

In the case study, twelve scenarios of SWPDCpM were carried out.

a) Scenario 1: Receive Telecommand;

b) Scenario 2: Submit Telecommand;

c) Scenario 3: Submit Telemetry;

d) Scenario 4: Enable/Disable Telemetry Sending;

e) Scenario 5: Report Computer Operation Mode;

f) Scenario 6: Changing Computer Operation Mode;

g) Scenario 7: Distribute Commands On and Off;

h) Scenario 8: Control of Dump Memory Process;

i) Scenario 9: Report Software Current Version;

j) Scenario 10: Report Charge State of a New Version;

k) Scenario 11: Start New Version Loading;

l) Scenario 12: Control Interfaces with SCA.

As for SWPDC, the next subsections detail three of the twelve scenarios of SWPDCpM (scenarios 6, 7, and 8). Subsection 5.2.2 summarizes the results obtained from all the scenarios analyzed.

5.2.1.1 Scenario 6: Changing Computer Operation Mode

In order to control the computer operation mode, the CTL process should: control the switching between computer operation modes of SWPDCpM; and allow to consult the current computer operation mode. There is one UML sequence diagram which represents Scenario 6. It is shown in Figure 5.16.

Observe the following requirement related to this scenario: *Upon receipt of a TC from CTL.TC requesting the change of computational operation mode, the CTL process should proceed the following sequence of operations: 1) To obtain new computer operation mode from TC and to perform mode changing by issuing internal commands; 2) If the computer operation mode can not be changed, a TM_RE must be*

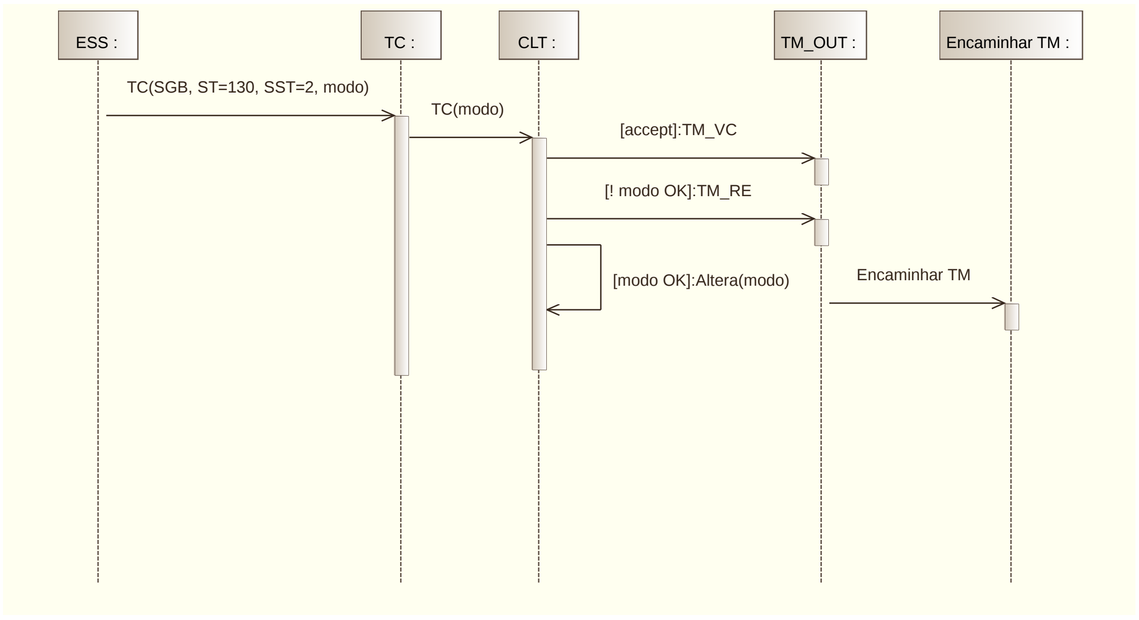


Figure 5.16 - Sequence Diagram for Scenario 6

generated and sent to CTL.TM_OUT; and 3) If a TM_RE is generated, it must contain the value of the current mode and the value of the requested mode. From this requirement, two user-defined properties for this scenario can be extracted:

a) Property 1: *Upon receipt a TC, if the command change mode is valid, the process must change the operation mode.* This property can be formalized using the **Response Pattern and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

   ¬∃ [¬ (Q ∧ ∀□ (P → ∀◇ (S))) ∪ (Q ∧ ¬ (Q ∧ ∀□ (P → ∀◇ (S))))]
   where,
   Q = ReceiveTC_CTL
   P = ValidChangeMode
   S = ChangeOperationMode

   Table 5.7 maps the commands and how they are represented in the TS. When running NuSMV, the property is not satisfied. This is because the two commands `ValidChangeMode` and `ChangeOperationMode` are represented by the same state in the TS generated.

b) Property 2: *Upon receipt a TC, if the command change mode is invalid, the process must generate _TM_RE.* This property can be formalized using the **Response Pattern and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

Table 5.7 - Commands and its representation on the TS

Command	TS element	Name and/or Value
ReceiveTC_CTL	Variable State	_tc$_$sgb$coma$___st$equals$130$coma$___ sst$equals$2$coma$___modo$_$
ValidChangeMode	Variable State	_#modo___ok#$dots$altera$_$modo$_$
ChangeOperationMode	Variable State	_#modo___ok#$dots$altera$_$modo$_$

$$\neg\exists\, [\, \neg\, (\, Q \wedge \forall\Box\, (P \rightarrow \forall\Diamond\, (S)))\, \cup\, (Q \wedge \neg\, (\, Q \wedge \forall\Box\, (P \rightarrow \forall\Diamond\, (S)))\,)]$$

where,

Q = ReceiveTC_CTL

P = InvalidChangeMode

S = Generate_TM_RE

Table 5.8 maps the commands and how they are represented in the TS. When running NuSMV, the property is not satisfied. This is because the two commands `InvalidChangeMode` and `Generate_TM_RE` are represented by the same state in the TS generated.

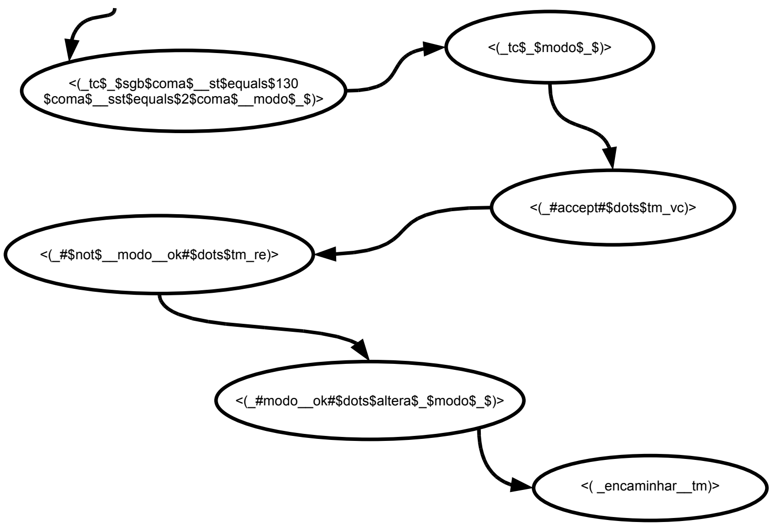
Table 5.8 - Commands and its representation on the TS

Command	TS element	Name and/or Value
ReceiveTC_CTL	Variable State	_tc$_$sgb$coma$___st$equals$130$coma$___ sst$equals$2$coma$___modo$_$
InvalidChangeMode	Variable State	_#$not$___modo___ok#$dots$tm_re
Generate_TM_RE	Variable State	_#$not$___modo___ok#$dots$tm_re

For Scenario 6, it was obtained 6 states, of which 6 are reachable states when running NuSMV model checker. Table 5.9 summarizes all information about this scenario. Figure 5.17 exhibits the unified TS for Scenario 6. The TS generated as *txt* file as well as the output containing the model checker notation (smv file) for this scenario can be found in Appendix B. Table B.1, also in Appendix B, presents all properties which were verified for all scenarios of SWPDCpM.

Table 5.9 - Statistics about Scenario 6 of SWPDCpM

UML diagrams			Transition System		Properties Verified	
SD	AD	SMD	Total States	Reachable States	Property	Satisfied
1	0	0	6	6	P1	No
					P2	No



<(_tc$_$sgb$coma$__st$equals$130 $coma$__sst$equals$2$coma$__modo$_$)>
<(_tc$_$modo$_$)>
<(_#accept#$dots$tm_vc)>
<(_#$not$__modo__ok#$dots$tm_re)>
<(_#modo__ok#$dots$altera$_$modo$_$)>
<(_encaminhar__tm)>

Figure 5.17 - The unified TS obtained for Scenario 6

5.2.1.2 Scenario 7: Distribute Commands On and Off

In this scenario the CTL process must distribute commands on and off required by TC. There is one UML sequence diagram which represents Scenario 7. It is shown in Figure 5.18.

Observe the following requirement related to this scenario: *The CTL process should distribute commands on/off requested by TC. The following sequence of operations must be performed: 1) Get command word on/off of the input TC; 2) Acting in hardware to perform the commands on/off through the CTL.ON_OFF_TC interface.* This requirement is represented by the following property:

a) Property 1: This property can be formalized using the **Precedence Pattern and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

   ¬∃ [¬ (Q ∧ (¬∃ [¬ S ∪ (P ∧ ¬ S)])) ∪ (Q ∧ ¬ (Q ∧ (¬∃ [¬ S ∪ (P ∧ ¬ S)])))]
   where,
   Q = ReceiveTC_CTL
   P = ObtainwordOnOff
   S = ActingHardware_CTL.ON_OFF_TC

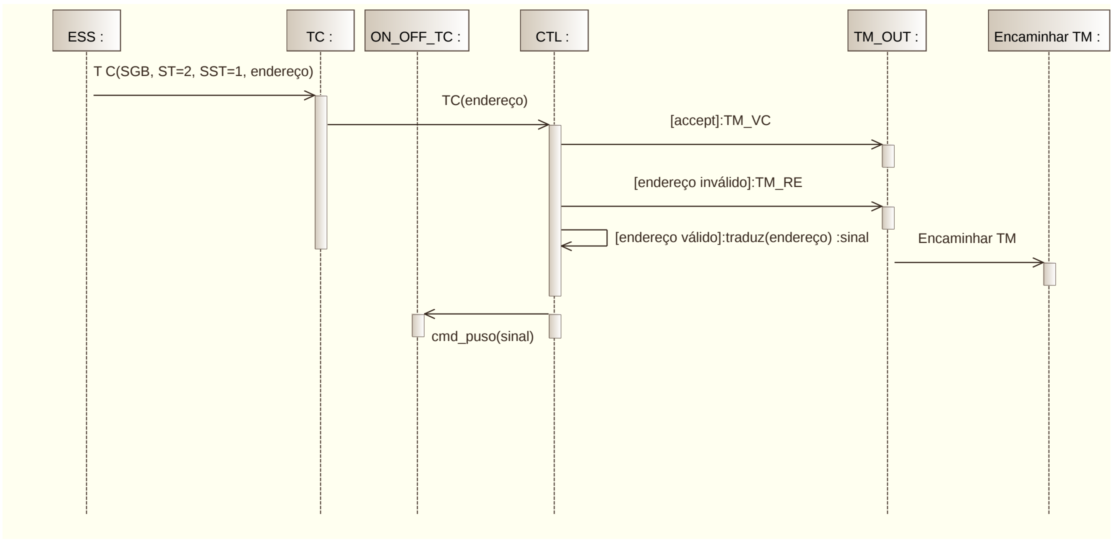


Figure 5.18 - Sequence Diagram for Scenario 7

Table 5.10 maps the commands and how they are represented in the TS. When running NuSMV, the property is satisfied.

Table 5.10 - Commands and its representation on the TS

Command	TS element	Name and/or Value
ReceiveTC__CTL	Variable State	__tc\$_\$endereco\$_\$
ObtainwordOnOff	Variable State	__#endereco___valido#\$dots \$traduz\$_\$endereco\$_\$___\$dots\$sinal
ActingHardware_CTL.ON_OFF_TC	Variable State	__cmd__pulso\$_\$sinal\$_\$

For Scenario 7, it was obtained 7 states, of which 7 are reachable states when running NuSMV model checker. Table 5.11 summarizes all information about this scenario. Figure 5.19 exhibits the unified TS for Scenario 7.

Table 5.11 - Statistics about Scenario 7 of SWPDCpM

UML diagrams			Transition System		Properties Verified	
SD	AD	SMD	Total States	Reachable States	Property	Satisfied
1	0	0	7	7	P1	Yes

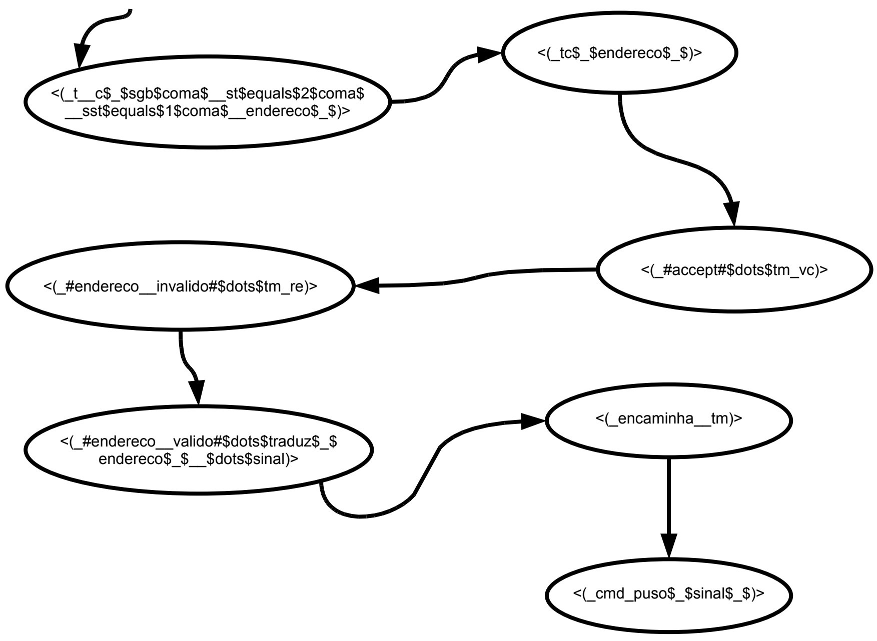

<(_t__c$_$sgb$coma$__st$equals$2$coma$ __sst$equals$1$coma$__endereco$_$)>
<(_tc$_$endereco$_$)>
<(_#accept#$dots$tm_vc)>
<(_#endereco__invalido#$dots$tm_re)>
<(_#endereco__valido#$dots$traduz$_$ endereco$_$__$dots$sinal)>
<(_encaminha__tm)>
<(_cmd_puso$_$sinal$_$)>

Figure 5.19 - The unified TS obtained for Scenario 7

5.2.1.3 Scenario 8: Control of Dump Memory Process

The CTL process should control the dump memory when requested by the ground station. There is one UML sequence diagram which represents Scenario 8. It is shown in Figure 5.20. Two user-defined properties for this scenario, related to requirements, were verified.

a) Property 1: *This function meets the telecommand of memory dump request. The following sequence of operations must be performed: 1) Get address and memory size required for dump; 2) Copy memory to dumpfile CTL.MMFS, reporting the progress of the operation through TM_VC, up to a maximum of 4MB; 3) Start transfer process of file generated in CTL.MMFS. Report TM_RE and ignore the request when: 4) The requested memory is invalid; 5) The requested size exceeds the maximum dump capacity; 6) A memory transfer process is still in progress.* This property can be formalized using the **Precedence Chain and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

$\neg\exists[\neg Q \cup (Q \wedge \exists [\neg S \cup P] \wedge \exists [\neg P \cup (S \wedge \neg P \wedge \exists \bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$

where,

Q = ReceiveTC_CTL.TC_DumpMemory

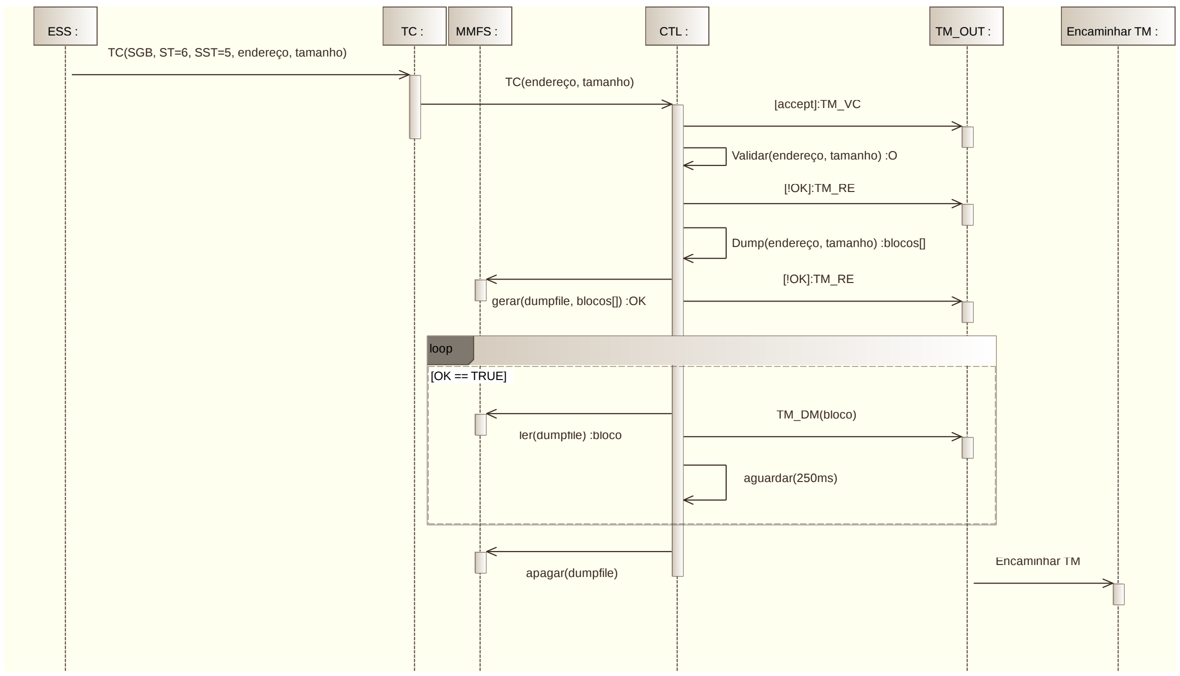


Figure 5.20 - Sequence Diagram for Scenario 8

S = GetAddressMemorySize
T = (¬(InvalidMemory ∨ InvalidSize ∨ ProcessTransferProgress) → (CopyMemory-Dumpfile → Forward_TM_VC))
P = StartTransferDumpFile

There is no state in the TS that represents the command `ProcessTransferProgress`. The property is not satisfied.

b) Property 2: *This process is initiated by CTL automatically when a dumpfile file is generated in CTL.MMFS. The following operation must be done in parallel with other software functions: 1) Generate TM_DM packages limited by the maximum size set to a TM package, considering that the last TM_DM package may have a size less than or equal to the maximum size of a packet TM; 2) Send TM_DM packages to CTL.TM_OUT at a rate not greater than four packets per second; 3) Report TM_VC indicating the completion status of the operation; 4) Delete dumpfile when finished. Report TM_RE and cancel transfer process when: 5) Occur I/O error in dumpfile; 6) A change occurs for computer mode operation that does not allow dump memory during dump process.* This property can be formalized using the **Precedence Chain and Scope After Q** proposed by (DWYER et al., 1999), in CTL, as follows:

$\neg\exists[\neg Q \cup (Q \wedge \exists[\neg S \cup P] \wedge \exists[\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc(\exists[\neg T \cup (P \wedge \neg T)]))])]$
where,

Q = DumpFileGenerated
S = (¬(ErrorIODumpFile ∨ ChangeOperationModeNDump) → GeneratePackageDM)
T = SendPackagesTM_DM_CTL.TM_OUT
P = (Report_TM_VC → DeleteDumpFile)

There is no state in the TS that represents the command `ChangeOperationModeNDump`. The property is not satisfied.

For Scenario 8, it was obtained 36 states, of which 16 are reachable states when running NuSMV model checker. Table 5.12 summarizes all information about this scenario. Figure 5.21 exhibits part of the unified TS for Scenario 8.

Table 5.12 - Statistics about Scenario 8 of SWPDCpM

UML diagrams			Transition System		Properties Verified	
SD	AD	SMD	Total States	Reachable States	Property	Satisfied
1	0	0	36	16	P1	No
					P2	No

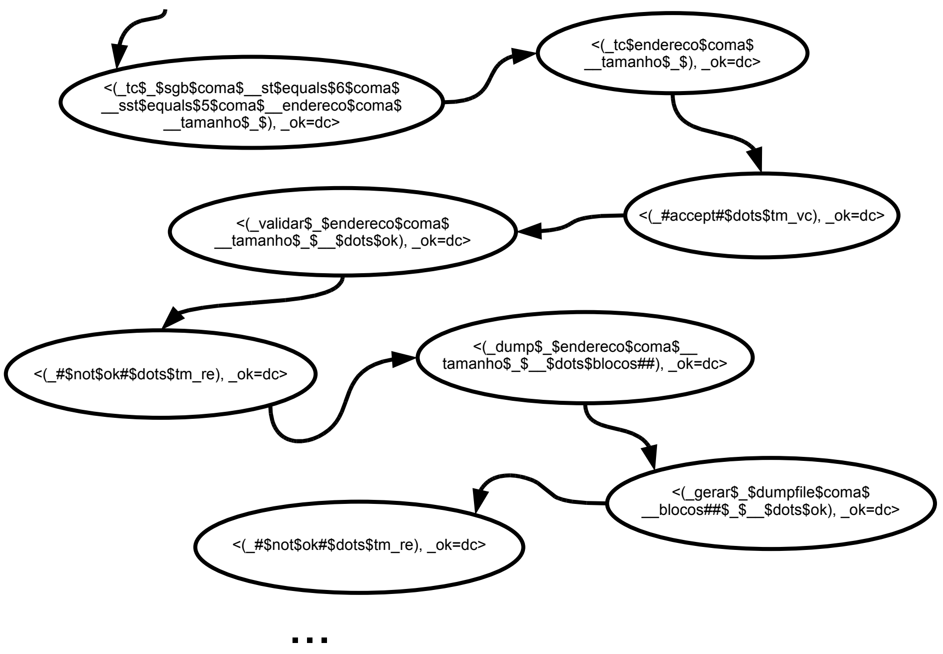

<(_tc$_$sgb$coma$__st$equals$6$coma$
__sst$equals$5$coma$__endereco$coma$
__tamanho$_$), _ok=dc>
<(_tc$endereco$coma$
__tamanho$_$), _ok=dc>
<(_#accept#$dots$tm_vc), _ok=dc>
<(_validar$_$endereco$coma$
__tamanho$_$__$dots$ok), _ok=dc>
<(_#$not$ok#$dots$tm_re), _ok=dc>
<(_dump$_$endereco$coma$__
tamanho$_$__$dots$blocos##), _ok=dc>
<(_gerar$_$dumpfile$coma$
__blocos##$_$__$dots$ok), _ok=dc>
<(_#$not$ok#$dots$tm_re), _ok=dc>
...

Figure 5.21 - Part of the unified TS obtained for Scenario 8

5.2.2 Summary of the results of SWPDCpM

Table 5.13 summarizes the results of all scenarios of SWPDCpM. It has the same information of the previous summarized table of case study SWPDC: number of UML diagrams used for each one of the scenarios, number of states (total and reachable), as well as the number of properties that were satisfied and not satisfied.

Table 5.13 - Summary of the results of the twelve scenarios analyzed for SWPDCpM

	UML			Transition System		Properties	
Scenarios	SD	AD	SMD	Total States	Reachable States	Satisfied	Not Satisfied
Scenario 1	1	0	0	15	7	2	0
Scenario 2	1	0	0	7	7	0	7
Scenario 3	1	0	0	27	11	2	0
Scenario 4	1	0	0	6	6	1	0
Scenario 5	1	0	0	5	5	1	0
Scenario 6	1	0	0	6	6	0	2
Scenario 7	1	0	0	7	7	1	0
Scenario 8	1	0	0	36	16	0	2
Scenario 9	1	0	0	6	6	1	0
Scenario 10	1	0	0	7	7	1	0
Scenario 11	1	0	0	7	7	1	0
Scenario 12	1	0	0	4	3	2	0

As it is possible to see, almost half of the properties were not satisfied, meaning that some diagrams do not meet all the requirements. Although this case study has only sequence diagrams, and thus it is not possible to exercise the whole features (which includes unifying the three UML diagrams) of SOLIMVA 3.0, this case study is very important because it is a complex space application under development at INPE. The results of using SOLIMVA 3.0 to analyze the (partial) design of SWPDCpM have already been communicated to the OBDH (SGB) development team.

The properties were not satisfied because it was not found states that could represent some commands or sentences present in the properties, and also because some commands or sentences were represented by one unique state in the TS.

5.3 Final Remarks

This chapter presented the application of SOLIMVA 3.0 to two case studies, SWPDC and SWPDCpM, related to an X-ray imaging balloon experiment under develoment at INPE. Defects were detected within the design of these software systems showing the feasibility of the methodology.

The case studies were conducted considering twenty scenarios: eight from SWPDC and twelve from SWPDCpM. For the eight scenarios of SWPDC, a total of nineteen properties were verified. Of these, fifteen were satisfied and four were not satisfied. This means that in 21.05% of the properties, the design of the SWPDC, by means of its UML diagrams, were not correct. For the twelve scenarios of SWPDCpM, a total of twenty three properties were verified. Of these, twelve were satisfied and eleven were not. Thus, for SWPDCpM, 47.83% of the properties were not satisfied. This is a significant defect detection in the (partial) design of the SWPDCpM.

It is important to know why the design (UML) did not satisfy the properties. Thus, an analysis of the causes of the defects found was performed. For SWPDC, it was always the same reason: there is no state in the TS that represents a certain command. This means that some commands are omitted or do not exist in the diagrams. For SWPDCpM, the defects occurred due to: (i) there is no state in the TS that represents a certain command; (ii) two commands are represented by the same state in the TS. So, the property can not be specified. Table 5.14 shows the causes and the number of properties which were not satisfied due to each cause, for the two case studies.

Table 5.14 - Causes for encountered defects

	(i) there is no state in the TS that represents a certain command	(ii) two commands are represented by the same state in the TS
SWPDC	4	0
SWPDCpM	5	6

Considering that, it is exposed the importance of the approach based on Formal Verification to improve the quality of embedded software development for INPE's space projects. The results show the potential for a wide acceptance of Formal Verification for the development of complex software systems.

The next chapter presents the contributions obtained in this work, as well as its limitations and future prospects of this research.

6 CONCLUSIONS AND FINAL REMARKS

INPE deals with critical embedded software in satellites and stratospheric baloons. Critical systems require high reliable software. A challenge in the systems development process is to advance defect detection at early stages of their life cycles. V&V techniques are essential to provide quality to the software systems being developed. Formal Methods, such as Model Checking, offer a large potential to provide effective verification techniques. However, Formal Methods require mathematical background and their use is restricted, as users privilege the simplicity of other notations, rather than more formal means.

In this line, many studies have emerged in order to develop approaches that allow the use of Formal Methods in a manner that is transparent to the user. One such example is the translation of industry non-formal standards such as UML to model checkers notation, considered as a great step towards a wide acceptance of Formal Methods in every day software development (SANTIAGO JÚNIOR, 2011).

Despite criticism regarding UML, it is indeed used in practice in many application domains. Modeling systems for object oriented and/or embedded software development is an approach that has been employed by researchers and practitioners, specially by means of the several UML behavioral diagrams.

In order to cover these two domains, taking advantage of the positive aspects of both, this PhD thesis proposed the SOLIMVA 3.0 methodology (SANTOS et al., 2014a),(SANTOS et al., 2014b),(ERAS et al., 2015), aiming to transform a non-formal language (UML) to a formal language (language of a model checker) in order to detect defects within the design of the software product. SOLIMVA 3.0 is an extension of a methodology initially developed to generate model-based system and acceptance test cases considering Natural Language requirements artifacts (SOLIMVA 1.0), and to detect incompleteness in software specifications by means of Model Checking (SOLIMVA 2.0). By including Formal Verification in the SOLIMVA methodology, the V&V process is enriched, addressing not only testing and inspection but also Formal Verification.

The approach proposed in SOLIMVA 3.0 considers the properties generated from UML use case models or requirements expressed in pure textual notation (Natural Language), and the Transition System translated from up to three UML behavioral diagrams: sequence, activity, and behavioral state machine. Then, Model Checking can be used to ensure that the behavior of the system satisfies the requirements, that is, whether the properties are satisfied by the Transition System that represents the behavior of the application under evaluation. Using more than one diagram provides a rich view of the system by different angles or in different states in time. It also helps to find inconsistencies and incompleteness in the models by confronting multiple views of the same system.

Additionally, a tool, XMITS, was developed to automate some steps of SOLIMVA 3.0 methodology. XMITS performs a three-step translation. First, it translates individual types of diagrams into a TS in a simple intermediate format. After that, XMITS merges all single TSs into a unified TS. Finally, the tool transforms this unified TS into the notation of the model checker (NuSMV).

Briefly, the process proposed in SOLIMVA 3.0 is as follows: first, scenarios are identified to be checked (scenarios are checked one by one). These scenarios have certain properties that must be verified. These properties are extracted from UML use case models or simply in Natural Language. SOLIMVA 3.0 suggests using specification patterns (DWYER et al., 1999) to direct the formalization of properties in Computation Tree Logic (BAIER; KATOEN, 2008). The UML diagrams that are related to the scenario that is being checked are input to XMITS. Hence, XMITS automatically generates a single, unified TS in the notation of the NuSMV model checker (KESSLER, 2015). By running NuSMV it is possible to observe if there are defects in the design of the software product. In case the TS does not satisfy a certain property, a counterexample is presented by the model checker. In the whole process, three tools are used: Modelio, to generate the UML diagrams and export them to XMI format; XMITS (which was developed to support this PhD thesis), that transforms the UML to the input language of the model checker; and NuSMV, which is the model checker used to perform the Model Checking.

SOLIMVA 3.0 was applied to two real space software products (embedded software), SWPDC and SWPDCpM in order to validade the methodology. These software systems are related to the balloon-borne high energy astrophysics experiment called protoMIRAX under development at INPE. Defects were detected within the design of these software systems, what shows that the methodology can be applied in practice. Specifically, 21.05% and almost 50% of the properties formalized in CTL were not satisfied for the SWPDC and SWPDCpM case studies, respectively. This implies that there are design flaws related to the conception of these software systems. The defects encountered were due to issues such as it was not found states that could represent some commands or sentences present in the properties, and also because some commands or sentences were represented by one unique state in the TS. The results show the potential for the use of an approach based on Formal Verification to improve the quality of embedded software development for critical INPE projects.

Regarding to the case studies, twenty scenarios were checked: eight from SWPDC and twelve from SWPDCpM. For SWPDC, nineteen properties were verified, and four of them were not satisfied, i.e., the design artifacts do not reflect the requirements related to these properties. For SWPDCpM, twenty three properties were verified. Of these, eleven were not satisfied, which is a significant defect detection in the design of SWPDCpM. These results analyzing the (partial) design of SWPDCpM have already been communicated to the OBDH (SGB) development team.

The main contribution of this PhD thesis is to help to make it easier the use of Formal Verification process for critical space software towards a greater adoption in practice of Formal Methods in software development. When compared to some other research studies, (MIKK et al., 1998),(LATELLA et al., 1999),(KONRAD; CHENG, 2006),(LAM, 2007),(ESHUIS, 2006),(ANDERSON et al., 1996),(DUBROVIN; JUNTTILA, 2008),(UCHITEL; KRAMER, 2001),(BARESI et al., 2011),(MIYAZAWA et al., 2013),(BEATO et al., 2005),(CORTELLESSA; MIRANDOLA, 2002),(MERSEGUER et al., 2002), SOLIMVA 3.0 has the following advantages:

a) it uses different behavioral diagrams, when most of the studies use only one single diagram;

b) it detects design defects considering functional requirements of the software product, when some researches focus on specific types of requirements, such as performance;

c) it demands only behavioral diagrams, that are often present in software documentation, when other works require a very large amount of artifacts, including for example, structural and behavioral diagrams.

A qualitative comparison between the SOLIMVA 3.0 methodology and other several approaches was presented in Section 2.5. Although a quantitative comparison between SOLIMVA 3.0 and these other approaches has not been made, it is very important to remark:

a) SOLIMVA 3.0 methodology was applied to two real and complex case studies of the space domain. In both case studies, there was involvement of the software national industry. The academic community has long been charged to reverse their knowledge to the industry and research institutes (such as INPE) so that these organizations can use such knowledge to the development of their systems. By using real and complex case studies, SOLIMVA 3.0, supported by the XMITS tool, is in agreement with this perspective allowing that an informal language (UML), still quite popular, can continue to be used for creating the design of software systems. The complexity for the use of formal methods (Model Checking, in this case) is almost completely hidden from the practitioner, and thus SOLIMVA 3.0 methodology has a high potential to be applied to the development of other highly complex software systems.

b) The software design developed for both case studies had already been assessed by experienced professionals in the context of their respective system/software development lifecycle and formal technical reviews. Finding additional defects within these design solutions, even if practitioners in the field have evaluated them, demonstrates the real usefulness of this PhD thesis.

6.1 Limitations

There are some limitations in SOLIMVA 3.0, of which stand out:

a) One of the main difficulties when working with Model Checking is the formalization of properties. SOLIMVA 3.0 has proposed a solution for automating the model generation. However, the properties still have to be manually formalized. A study about automated property formalization, specially if requirements are expressed in Natural Language, is important to be carried out.

b) Another restriction is related to counterexample. How to find the diagram and the exact point in this diagram where the property was not satisfied? The automated translation of the model checker counterexample back to the UML diagrams is important to identify in which diagram, or diagrams, the detected problem is related.

6.2 Suggestions for Future Research

To solve the limitations presented above, the following list of future work is proposed:

a) Directives to improve the automation of property formalization or facilitate this formalization is important to be addressed.

b) Another important future work is the development of a new module of XMITS to catch the feedback from NuSMV and show to the user the results automatically.

In addition, the following other suggestions can be listed:

a) It is important to define a more detailed formal semantics for the translation from UML to TS proposed by SOLIMVA 3.0.

b) One other interesting new feature could be the implementation of new UML diagrams compatibility. The Converter module of XMITS is ready to accept new UML diagrams by adding a new *collector* class.

c) A subset of the UML diagrams syntax is implemented in XMITS. There are several other features not supported by XMITS in the current version. To increase this subset also increases the possibility of the tool use.

d) NuSMV is the only model checker used in XMITS. New grammars can be defined so that other model checkers, such as SPIN, may be used.

e) The current ouput of TUTS module is a txt file, showing all states. It is not possible to see the TS tree, following the flow of transitions. Adding an output in graph format, showing states and transitions would help and facilitate the validation of the TS.

f) Regarding to parallelism, when a process is divided due to parallel messages/activities/states, each process goes on alone. In many cases, there are states that are created in each one of these processes that are the same, that is, repeated states. Sometimes this led to not run some scenarios, especially those with many cases of parallelism. A method in XMITS that seeks for repeated states during the process of creation of the TS is missing. Thus, it is relevant to improve the tool with such a method to deal with state space explosion.

REFERENCES

AMJAD, H. **Combining model checking and theorem proving**. Cambridge, United Kingdom, 2004. 131p. Technical Report. 18, 19

ANDERSON, R. J.; BEAME, P.; BURNS, S.; CHAN, W.; MODUGNO, F.; NOTKIN, D.; REESE, J. D. Model checking large software specifications. **ACM SIGSOFT Software Engineering Notes**, v. 21, p. 156–166, 1996. 4, 6, 32, 36, 39, 123

BAIER, C.; KATOEN, J.-P. **Principles of model checking**. Cambridge, MA, USA: MIT Press, 2008. 975 p. Available from: <http://mitpress.mit.edu/books/principles-model-checking>. xv, 1, 3, 5, 18, 19, 20, 24, 27, 45, 64, 71, 122

BARESI, L.; MORZENTI, A.; MOTTA, A.; ROSSI, M. Towards the uml-based formal verification of timed systems. **Formal Methods for Components and Objects**, v. 6957, p. 267–286, 2011. 6, 34, 36, 39, 123

BEATO, M. E.; BARRIO-SOLÓRZANO, M.; CUESTA, C. E.; FUENTE, P. de la. Uml automatic verification tool with formal methods. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 127, n. 4, p. 3–16, 2005. 6, 35, 36, 123

BEHRMANN, G.; DAVID, A.; LARSEN, K. A tutorial on uppaal. **Formal methods for the design of real-time systems**, Springer, v. 3185, p. 200–236, 2004. 26

BJORK, R. C. **The simulation of an automated teller machine.** 2012. Available from: <http://www.math-cs.gordon.edu/courses/cs211/ATMExample/>. Access in: 11 mai. 2014. 43

BOEHM, B. **Software engineering economics**. 1. ed. US: Prentice Hall, 1981. 9

BRAGA, J.; D'AMICO, F.; AVILA, M. A. C.; PENACCHIONI, A. V.; SACAHUI, J. R.; SANTIAGO, V. A. de; MATTIELLO-FRANCISCO, F.; STRAUSS, C.; FIALHO, M. A. A. The protomirax hard x-ray imaging balloon experiment. **A&A**, v. 580, p. A108, 2015. Available from: <http://dx.doi.org/10.1051/0004-6361/201526343>. xvi, 5, 109, 110

BRIAND, L.; LABICHE, Y. A uml-based approach to system testing. **Software and Systems Modeling**, Springer, v. 1, n. 1, p. 10–42, 2002. 31, 36

BRITO, P. H. S.; LEMOS, R. D.; RUBIRA, C. M. F.; MARTINS, E. Architecting fault tolerance with exception handling: verification and validation. **J. Comput. Sci. Technol.**, Institute of Computing Technology, Beijing, China, v. 24, n. 2, p. 212–237, mar. 2009. ISSN 1000-9000. Available from: <http://dx.doi.org/10.1007/s11390-009-9219-2>. 31, 36, 39

CAVADA, R.; CIMATTI, A.; JOCHIM, C. A.; KEIGHREN, G.; OLIVETTI, E.; PISTORE, M.; ROVERI, M.; TCHALTSEV, A. **NuSMV 2.5 user manual**. 2005. Available from: <http://nusmv.fbk.eu/>. Access in: 12 mai. 2013. 66

CHEN, M.; MISHRA, P.; KALITA, D. Coverage-driven automatic test generation for uml activity diagrams. In: ACM GREAT LAKES SYMPOSIUM ON VLSI, 18., 2008, Orlando, Florida, USA. **Proceedings...** New York, NY, USA: ACM, 2008. p. 139–142. ISBN 978-1-59593-999-9. 477088. 31, 36

CLARKE, E. The birth of model checking. **25 Years of Model Checking**, Springer, v. 5000, p. 1–26, 2008. 22

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). 3, 19

CLARKE, E. M.; GRUMBERG, O.; PELED, D. **Model checking**. Cambridge - MA: MIT press, 1999. ISBN 978-0262032704. 1

COCKBURN, A. **Writing effective use cases**. US: Addison-Wesley Professional, 2000. 304 p. 11, 43

CORTELLESSA, V.; MIRANDOLA, R. Prima-uml: a performance validation incremental methodology on early uml diagrams. **SC Programming**, Elsevier, v. 44, n. 1, p. 101–129, 2002. 6, 34, 36, 123

DEBBABI, M.; HASSAÏNE, F.; JARRAYA, Y.; SOEANU, A.; ALAWNEH, L. **Verification and validation in systems engineering**. Berlin, Heidelberg - Germany: Springer, 2010. 270 p. xv, 4, 63, 74

DELAMARO, E.; MALDONADO, J. C.; M., J. **Introdução ao teste de software**. Rio de Janeiro: Ed. Elsevier, 2007. 1

DUBROVIN, J.; JUNTTILA, T. Symbolic model checking of hierarchical uml state machines. In: INTERNATIONAL CONFERENCE ON APPLICATION OF CONCURRENCY TO SYSTEM DESIGN, 8., 2008, Xi'An - China. **Proceedings...** Xi'An - China: IEEE, 2008. p. 108–117. ISBN 978-1-4244-1838-1. 4, 6, 32, 36, 39, 123

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Patterns in property specifications for finite-state verification. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 21., 1999, Los Angeles, CA, USA. **Proceedings...** New York, NY, USA: ACM, 1999. p. 411–420. ISBN 1-58113-074-0. 5, 24, 31, 44, 64, 72, 73, 78, 94, 96, 97, 99, 100, 102, 103, 105, 106, 112, 114, 116, 117, 122

ECLIPSE.ORG. **Papyrus**. 2014. Available from: <https://www.eclipse.org/papyrus/>. Access in: 19 jan. 2014. 161

ERAS, E. R.; SANTOS, L. B. R. dos; JÚNIOR, V. A. de S.; VIJAYKUMAR, N. L. Towards a wide acceptance of formal methods to the design of safety critical software: an approach based on uml and model checking. **Lecture Notes in Computer Science**, v. 9158, p. 612–627, 2015. 4, 5, 77, 121

ESHUIS, R. Symbolic model checking of uml activity diagrams. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 15, n. 1, p. 1–38, 2006. 4, 6, 32, 36, 123

FRASER, G.; WOTAWA, F.; AMMANN, P. Testing with model checkers: a survey. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 19, n. 3, p. 215–261, 2009. 22, 25

GANAI, M.; GUPTA, A. **SAT-based scalable formal verification solutions**. Princeton - US: Springer, 2007. 2, 17

GILB, T.; GRAHAM, D.; FINZI, S. **Software inspection**. London - UK: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN 078-5342631814. 1

GODBOLE, N. S. **Software quality assurance**: principles and practice. Oxford, UK: Alpha Science International, 2006. 1

HAREL, D. Statecharts: A visual formalism for complex systems. **Science of computer programming**, Elsevier, v. 8, n. 3, p. 231–274, 1987. 3

HARTMANN, J.; VIEIRA, M.; FOSTER, H.; RUDER, A. A uml-based approach to system testing. **Innovations in Systems and Software Engineering**, Springer, v. 1, n. 1, p. 12–24, 2005. 31, 36

HOLZMANN, G. **The SPIN model checker**: primer and reference manual. Boston - MA: Addison-Wesley, 2004. 25, 63

IEEE. Institute of electric and electronic engineers. **Standard glossary of software engineering terminology**, Standard 610.12, 1990. 1, 2, 9, 10

JOHNSON, K.; CALINESCU, R.; KIKUCHI, S. An incremental verification framework for component-based software systems. In: INTERNATIONAL ACM SIGSOFT SYMPOSIUM ON COMPONENT-BASED SOFTWARE ENGINEERING, 16., 2013. **Proceedings...** New York, NY, USA: ACM, 2013. p. 33–42. ISBN 978-1-4503-2122-8. 33

JÚNIOR, V. A. d. S.; CRISTIÁ, M.; VIJAYKUMAR, N. L. **Model-based test case generation using statecharts and z: a comparison and a combined approach**. São José dos Campos, 2010. 70 p. Available from: <`http://urlib.net/sid.inpe.br/mtc-m19@80/2010/02.26.14.05`>. Access in: 30 nov. 2015. 90

KESSLER, F. B. **NuSMV Home Page**. 2015. Available from: <`http://nusmv.fbk.eu/`>. Access in: 12 mai. 2015. 5, 25, 45, 63, 122

KIM, Y.; HONG, H.; BAE, D.; CHA, S. Test cases generation from uml state diagrams. In: IET. **Software, IEE Proceedings-**. UK, 1999. v. 146, n. 4, p. 187–192. 31, 36

KNAPP, A.; MERZ, S. Model checking and code generation for uml state machines and collaborations. In: WORKSHOP ON TOOLS FOR SYSTEM DESIGN AND VERIFICATION, 5., 2002, Reisensburg, Germany. **Proceedings...** Reisensburg, Germany: Institut für Informatik, Universität Augsburg, 2002. v. 11, p. 59–64. 3, 33, 36, 39

KONRAD, S.; CHENG, B. Real-time specification patterns. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 27., 2005, St. Louis, MO, USA. **Proceedings...** New York, US: ACM, 2005. p. 372–381. ISBN 1-58113-963-2. 33, 36

KONRAD, S.; CHENG, B. H. Automated analysis of natural language properties for uml models. **Lecture Notes in Computer Science**, v. 3844, p. 48–57, 2006. 4, 6, 33, 36, 123

LAM, V. S. W. A formalism for reasoning about uml activity diagrams. **Nordic J. of Computing**, Publishing Association Nordic Journal of Computing, Finland, v. 14, n. 1, p. 43–64, jan. 2007. ISSN 1236-6064. Available from: <http://dl.acm.org/citation.cfm?id=1515784.1515786>. 4, 6, 32, 36, 123

LATELLA, D.; MAJZIK, I.; MASSINK, M. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. **Formal Aspects of Computing**, Springer, v. 11, n. 6, p. 637–664, 1999. 4, 6, 32, 36, 123

MAKINEN, M. A. **Model based approach to software testing**. Master Thesis — Helsinki University of Technology, Helsinki, 2007. 10

MARTHUR, A. P. **Foundations of software testing.** India: Dorling Kindersley: Pearson Education in South Asia, 2008. 1, 2, 6, 40

MATOS, A. V. de. **Unified modeling language UML prático e descomplicado**. São Paulo, Brazil: Érica, 2002. 16

MERSEGUER, J.; CAMPOS, J.; BERNARDI, S.; DONATELLI, S. A compositional semantics for uml state machines aimed at performance evaluation. In: INTERNATIONAL WORKSHOP ON DISCRETE EVENT SYSTEMS, 6., 2002, Zaragoza, Spain. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2002. p. 295–302. ISBN 0-7695-1683-1. 6, 34, 36, 123

MERZ, S. Model checking: A tutorial overview. **Modeling and verification of parallel processes**, Springer, v. 2067, p. 3–38, 2001. 22

MIKK, E.; LAKHNECH, Y.; SIEGEL, M.; HOLZMANN, G. Implementing statecharts in promela/spin. In: WORKSHOP ON INDUSTRIAL STRENGTH FORMAL SPECIFICATION TECHNIQUES, 2., 1998. **Proceedings...** Washington, DC, USA: IEEE, 1998. p. 90–101. ISBN 0-7695-0081-1. 4, 6, 32, 36, 39, 123

MIYAZAWA, A.; ALBERTINS, L.; IYODA, J.; CORNÉLIO, M.; PAYNE, R.; CAVALCANTI, A. **Final report on combining SysML and CML**. Seventh Framework Programme, 2013. 219p. Technical Report. 6, 34, 36, 123

MODELIOSOFT. **Modelio open source community**. 2011. Available from: <https://www.modelio.org>. Access in: Feb. 4, 2015. 79

NASA. **NASA software assurance**: software assurance definitions. 2009. Available from: <http://www.hq.nasa.gov/office/codeq/software/umbrella_defs.htm>. Access in: 19 mai. 2014. 1

NASA, A. R. C. **Java Pathfinder Home Page**. 2015. Available from: <http://babelfish.arc.nasa.gov/trac/jpf>. Access in: 18 abr. 2015. 26

NAUR, P.; BACKUS, J. W.; BAUER, F. L.; GREEN, J.; KATZ, C.; MCCARTHY, J.; PERLIS, A. J. Revised report on the algorithmic language algol 60. **Communications of the ACM**, v. 6, n. 1, p. 1–17, 1963. 66

OMG, T. O. M. G. **OMG - Unified Modeling Language (OMG UML)**. 2015. Available from: <http://www.uml.org/>. Access in: 07 aug. 2014. 2, 10, 78

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. 78

PARNAS, D. L.; CLEMENTS, P. C. A rational design process: How and why to fake it. **IEEE Trans. Software Eng.**, v. 12, n. 2, p. 251–257, 1986. Available from: <http://doi.ieeecomputersociety.org/10.1109/TSE.1986.6312940>. 1

PETRE, M. Uml in practice. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 35., 2013, San Francisco, CA, USA. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2013. p. 722–731. ISBN 978-1-4673-3076-3. 2

PRESS, O. U. **The Oxford Dictionary**. 2015. Available from: <http://www.oxforddictionaries.com>. Access in: 15 out. 2015. 4

PRESSMAN, R. S. **Software engineering**: a practitioner's approach - 7th ed. New York, US: McGrawHill, 2010. ISBN 0073375977. 11, 14

QUEILLE, J.-P.; SIFAKIS, J. Specification and verification of concurrent systems in cesar. **Lecture Notes in Computer Science**, v. 137, p. 337–351, 1982. 3, 19

RIEBISCH, M.; PHILIPPOW, I.; GÖTZE, M. Uml-based statistical test case generation. **Objects, Components, Architectures, Services, and Applications for a Networked World**, Springer, v. 2591, p. 394–411, 2003. 31, 36

SANTIAGO JÚNIOR, V.; VIJAYKUMAR, N. L.; FERREIRA, E.; GUIMARãES, D.; COSTA, R. C. Gtsc: Automated model-based test case generation from statecharts and finite state machines. In: SESSÃO DE FERRAMENTAS DO III CONGRESSO BRASILEIRO DE SOFTWARE: TEORIA E PRáTICA (CBSOFT), 3., 2012, Natal, RN. **Anais...** Porto Alegre - RS: SBC, 2012. p. 25–30. 28

SANTIAGO JÚNIOR, V. A. **SOLIMVA**: a methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications. 2011. 264 p. (sid.inpe.br/mtc-m19/2011/11.07.23.30-TDI). Thesis (PhD in Applied Computing) — Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil, 2011. xv, xvi, 1, 3, 4, 5, 25, 28, 29, 30, 31, 40, 89, 90, 91, 121

SANTIAGO JÚNIOR, V. A.; VIJAYKUMAR, N. L. Generating model-based test cases from natural language requirements for space application software. **Software Quality Control**, Kluwer Academic Publishers, Hingham, MA, USA, v. 20, n. 1, p. 77–143, mar. 2012. ISSN 0963-9314. Available from: <http://dx.doi.org/10.1007/s11219-011-9155-6>. 28

SANTIAGO, V.; MATTIELLO-FRANCISCO, M.; COSTA, R.; SILVA, W.; AMBROSIO, A. Qsee project: an experience in outsourcing software development for space applications. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING & KNOWLEDGE ENGINEERING, 19., 2007, Boston, MA, USA. **Proceedings...** Boston - USA: Citeseer, 2007. p. 51–56. 5, 89

SANTIAGO, V.; VIJAYKUMAR, N.; GUIMARÃES, D.; AMARAL, A.; FERREIRA, E. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING VERIFICATION AND VALIDATION WORKSHOP, 2008, Lillehammer,Norway. **Proceedings...** Washington, DC, USA: IEEE, 2008. p. 63–72. ISBN 978-0-7695-3388-9. 28

SANTOS, L. B. R. d.; JÚNIOR, V. A. d. S.; VIJAYKUMAR, N. L. Transformation of uml behavioral diagrams to support software model checking. **Electronic Proceedings in Theoretical Computer Science**, v. 147, p. 133–142, 2014. 4, 39, 40, 121

SANTOS, L. B. R. dos; ERAS, E. R.; JR., V. A. de S.; VIJAYKUMAR, N. L. A formal verification tool for UML behavioral diagrams. **Lecture Notes in Computer Science**, v. 8579, p. 696–711, 2014. 4, 5, 77, 121

SARMA, M.; MALL, R. Automatic generation of test specifications for coverage of system state transitions. **Information and Software Technology**, Elsevier, v. 51, n. 2, p. 418–432, 2009. 10, 31, 36, 39, 46, 47

SCHÄFER, T.; KNAPP, A.; MERZ, S. Model checking uml state machines and collaborations. **Electronic Notes in Theoretical Computer Science**, Elsevier, Reisensburg, Germany, v. 55, n. 3, p. 357–369, 2001. 3, 33, 36, 39

SETZER, A. **Interactive theorem proving**. 2008. Available from: <`http://www.cs.swan.ac.uk/~csetzer/lectures/intertheo/07/interactiveTheoremProvingForAgdaUsers.html`>. Access in: 02 Jul. 2015. 18

SIAU, K.; HALPIN, T. A. (Ed.). **Unified modeling language**: systems analysis, design and development issues. US: Idea Group, 2001. ISBN 1-930708-05-X. 40

SPIVEY, J. M.; ABRIAL, J. **The Z notation**. UK: Prentice Hall Hemel Hempstead, 1992. 91

The Eclipse Foundation. **Eclipse. Available from: http://eclipse.org. Access in: Feb. 4, 2015**. 2015. 78, 161

THE OBJECT MANAGEMENT GROUP (OMG). **OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1**. Needham, MA, USA, 2011. 722 p. 12, 15, 39

UBM TECH. 2013 embedded market study. In: DESIGNWEST CONFERENCE AND EXHIBITION, 2013, McEnergy Convention Center. **Proceedings...** San Jose - CA: USA Conference Series, 2013. 2

UCHITEL, S.; KRAMER, J. A workbench for synthesising behaviour models from scenarios. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 23., 2001, Toronto, Ontario, Canada. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2001. p. 188–197. ISBN 0-7695-1050-7. 4, 6, 32, 36, 123

WIKI. **Method or Methodology?** 2015. Available from: <`http://c2.com/cgi/wiki?MethodOrMethodology`>. Access in: 15 out. 2015. 4

WOODCOCK, J.; CAVALCANTI, A.; FITZGERALD, J.; LARSEN, P. G.; MIYAZAWA, A.; PERRY, S. Features of cml: A formal modelling language for systems of systems. In: INTERNATIONAL CONFERENCE ON SYSTEM OF SYSTEMS ENGINEERING, 7., 2012, Genoa, Italy. **Proceedings...** Washington, DC, USA: IEEE, 2012. ISBN 978-1-4673-2974-3. 34

APPENDIX A - ADDITIONAL INFORMATION ABOUT SWPDC CASE STUDY

A.1 Verified Properties

This section presents all the properties which have been verified for all scenarios of SWPDC. Table A.1 shows the properties descriptions and their respective CTL formalization for each one of the eight scenarios analyzed.

Table A.1 - Verified Properties of SWPDC case study

Scenario 1	
Property Description	CTL formalization
1) The POST (Power-On Self Test) shall comprise: (i) Power status of the PDC itself; (ii) Power satus of the two EPP-HXI sets; (iii) Current internal temperature of the PDC; (iv) Coherent information of the PDC Program Memory; (v) Reading of SRAM (Data Memory) and; (vi) Correct operation of the watchdog timer circuit	**Existence Pattern and Globally Scope :** ∀◊(PowerstatusPDC) ∀◊(PowerstatusEPP-HXI) ∀◊(temperaturePDC) ∀◊(PDCProgramMemory) ∀◊(ReadDataMemory) ∀◊(OperationWatchdog)
2) The SWPDC should know how to distinguish between a power-on process and a reset process	**Existence Pattern and Scope After Q :** ¬∃ [¬ (Q ∧ ∀◊ (P)) ∪ (Q ∧ ¬ (Q ∧ ∀◊ (P)))] where P = modoIniciacaoPowerOn=true Q = DeterminarMododeIniciacao
3) The SWPDC should report processing of the POST through reports of events	**Existence Pattern and Globally Scope :** ∀◊(RecordEventReport)
4)In the case of any unrecoverable problem not being identified in the PDC after the initiation process, the PDC shall automatically enter into the safety operation mode	**Response Pattern and Globally Scope :** ∀□ ((State=_–Avaliarprocessodeiniciacao–_ ∧ bemSucedido=true) → ∀◊ (State= _–MudarparamododeoperacaoSEGURANCA–_))
Scenario2	
Property Description	CTL formalization
1)The SWPDC/PDC can switching on and off the set EPP-HXI-1 and EPP-HXI-2 through command execution from the OBDH	**Response Pattern and Globally Scope :** ∀□ (CommandSwitchOnEH1 → ∀◊ (SwitchOnEH1)) ∀□ (CommandSwitchOffEH1 → ∀◊ (SwitchOffEH1)) ∀□ (CommandSwitchOnEH2 → ∀◊ (SwitchOnEH2)) ∀□ (CommandSwitchOffEH2 → ∀◊ (SwitchOffEH2))
Scenario3	
Property Description	CTL formalization
1) The time for generation of housekeeping data can be changed by means of command sent by OBDH. The minimum value of time to generate housekeeping is 60s, and and the maximum value is 1000s	**Universality Pattern and Globally Scope :** ∀□(ModifyParameters(HK,60s)) ∀□(ModifyParameters(HK,1000s))
2) A session for transmission of housekeeping data begins with the command PREPARE HOUSEKEEPING DATA. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT HOUSEKEEPING DATA (RETRANSMIT ANSWER, when necessary) and ends with the command TRANSMIT SCIENTIFIC DATA, where SWPDC returns to acquire scientific data of EPPs	**Precedence Chain and Scope After Q :** ¬∃[¬ Q ∪ (Q ∧∃ [¬S ∪ P] ∧∃ [¬P ∪ (S ∧ ¬P ∧ ∃◯ (∃[¬T ∪ (P ∧ ¬T)]))])] where, Q = PREPARE HOUSEKEEPING DATA S = STOP DATA ACQUISITION T = TRANSMIT HOUSEKEEPING DATA P = TRANSMIT SCIENTIFIC DATA
Scenario7	
Property Description	CTL formalization
1) A session for transmission of housekeeping data begins with the command PREPARE HOUSEKEEPING DATA. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT HOUSEKEEPING DATA (RETRANSMIT ANSWER, when necessary) and ends with the command	**Precedence Chain and Scope After Q :** ¬∃[¬ Q ∪ (Q ∧∃ [¬S ∪ P] ∧∃ [¬P ∪ (S ∧ ¬P ∧ ∃◯ (∃[¬T ∪ (P ∧ ¬T)]))])] where, Q = PREPARE HOUSEKEEPING DATA S = STOP DATA ACQUISITION

Continued on next page

TRANSMIT SCIENTIFIC DATA, where SWPDC returns to acquire scientific data of EPPs	T = TRANSMIT HOUSEKEEPING DATA
	P = TRANSMIT SCIENTIFIC DATA

Scenario8	
Property Description	CTL formalization

1) A session for transmission of housekeeping data begins with the command PREPARE HOUSEKEEPING DATA. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT HOUSEKEEPING DATA (RETRANSMIT ANSWER, when necessary) and ends with the command TRANSMIT SCIENTIFIC DATA, where SWPDC returns to acquire scientific data of EPPs	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = PREPARE HOUSEKEEPING DATA
	S = STOP DATA ACQUISITION
	T = TRANSMIT HOUSEKEEPING DATA
	P = TRANSMIT SCIENTIFIC DATA

2)The PDC can also have a situation of not receiving the command sent. After identifying delimiter of start the PDC waits for a time of 600 ms for the rest of the command. This time is equivalent to 2 times as long to transmit command with maximum size (1128 Bytes). In the case of expiry of the stipulated time, a timeout occurs and the PDC aborts the communication, the command is discarded, one event is raised to it, and the PDC expects a new command from OBDH	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = StartCommand & Wait600ms
	S = Timeout
	T = GenerateReportEvent
	P = ExpectNewCommandOBDH

3)The OBDH checks all fields of the replies sent by the PDC. In the case of OBDH send a command TRANSMIT DATA and if there is inconsistency in the values received in any of the fields of a data response (SCIENTIFIC DATA, HOUSEKEEPING DATA, DUMP DATA MEMORY, DIAGNOSTIC DATA, AND TEST DATA), the OBDH uses the command RETRANSMIT RESPONSE to try get a consistent data response. The command RETRANSMIT RESPONSE refers only to the last data response sent by the PDC. Therefore, the PDC must maintain, temporarily, always the latest data response that was sent to the OBDH for case of error in receipt of this response by the OBDH. The OBDH sends this same command for maximum two more times. If after these other two attempts still there is problem in the response received from the PDC, the OBDH does not transmit this command and generates a bug report which should be sent to the ground station	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = TRANSMIT DATA & PDCResponseProblem
	S = RETRANSMIT RESPONSE 1 & PDCResponseProblem
	T = RETRANSMIT RESPONSE 2 & PDCResponseProblem
	P = Stop Sending & GenerateReportEvent

4)The SWPDC should allow uploading of programs in this operation mode. The first command to be sent to the SWPDC is STOP DATA ACQUISITION, for the SWPDC interrupts scientific data acquisition of the EPPs. After this command, one or more commands to LOAD DATA MEMORY should be sent to the new program to be loaded in the data memory of PDC. Later, a command RUN PROGRAM LOADED INTO MEMORY should be sent to the PDC to have the program loaded to run. At the end of the process of loading and executing of the program, a command RESTART DATA ACQUISITION causes the SWPDC return to acquire scientific data of EPPs	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = STOP DATA ACQUISITION
	S = LOAD DATA MEMORY
	T = RUN PROGRAM LOADED
	P = RESTART DATA ACQUISITION

Scenario9	
Property Description	CTL formalization

1)A session for transmission of data dump begins with the command PREPARE DUMP DATA MEMORY. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT DUMP DATA MEMORY and ends with the command TRANSMIT SCIENTIFIC DATA, where the SWPDC returns to acquire the scientific data of EPPs	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = PREPARE DUMP DATA MEMORY
	S = STOP DATA ACQUISITION
	T = TRANSMIT DUMP DATA MEMORY
	P = TRANSMIT SCIENTIFIC DATA

Scenario10	
Property Description	CTL formalization

1)A session for transmission of data dump begins with the command PREPARE DUMP DATA MEMORY. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT DUMP DATA MEMORY and ends with the command TRANSMIT SCIENTIFIC DATA, where the SWPDC returns to acquire the scientific data of EPPs	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = PREPARE DUMP DATA MEMORY
	S = STOP DATA ACQUISITION
	T = TRANSMIT DUMP DATA MEMORY

Continued on next page

	P = TRANSMIT SCIENTIFIC DATA

2) The PDC checks all fields of commands sent by the OBDH. In case of inconsistency in the values received in any of the fields of the OBDH command, the PDC aborts the communication, the command is dropped, an event is raised, the PDC awaiting a new OBDH command	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup$
	$(S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = SyntaticErrorCommand
	S = Timeout
	T = GenerateReportEvent
	P = ExpectNewCommandOBDH

3)The PDC can also have a situation of not receiving the command sent. After identifying delimiter of start the PDC waits for a time of 600 ms for the rest of the command. This time is equivalent to 2 times as long to transmit command with maximum size (1128 Bytes). In the case of expiry of the stipulated time, a timeout occurs and the PDC aborts the communication, the command is discarded, one event is raised to it, and the PDC expects a new command from OBDH	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup$
	$(S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = StartCommand & Wait600ms
	S = Timeout
	T = GenerateReportEvent
	P = ExpectNewCommandOBDH

Scenario12	
Property Description	CTL formalization

1)A session for transmission of data dump begins with the command PREPARE DUMP DATA MEMORY. In this case, the SWPDC interrupts the acquisition of scientific data of EPPs. The session continues with one or more commands TRANSMIT DUMP DATA MEMORY and ends with the command TRANSMIT SCIENTIFIC DATA, where the SWPDC returns to acquire the scientific data of EPPs	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup$
	$(S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = PREPARE DUMP DATA MEMORY
	S = STOP DATA ACQUISITION
	T = TRANSMIT DUMP DATA MEMORY
	P = TRANSMIT SCIENTIFIC DATA

2) The PDC checks all fields of commands sent by the OBDH. In case of inconsistency in the values received in any of the fields of the OBDH command, the PDC aborts the communication, the command is dropped, an event is raised, the PDC awaiting a new OBDH command	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup$
	$(S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = SyntaticErrorCommand
	S = Timeout
	T = GenerateReportEvent
	P = ExpectNewCommandOBDH

3)The PDC can also have a situation of not receiving the command sent. After identifying delimiter of start the PDC waits for a time of 600 ms for the rest of the command. This time is equivalent to 2 times as long to transmit command with maximum size (1128 Bytes). In the case of expiry of the stipulated time, a timeout occurs and the PDC aborts the communication, the command is discarded, one event is raised to it, and the PDC expects a new command from OBDH	**Precedence Chain and Scope After Q :**
	$\neg\exists[\neg Q \cup (Q \wedge\exists [\neg S \cup P] \wedge\exists [\neg P \cup$
	$(S \wedge \neg P \wedge \exists\bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))])]$
	where,
	Q = StartCommand & Wait600ms
	S = Timeout
	T = GenerateReportEvent
	P = ExpectNewCommandOBDH

A.2 Tansition System for Scenario 1

```
Node Children
---------------------------------------------------------
{1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 1, 1, 3, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 2, 1, 1, 2, 1, 2,
1, 2, 1, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 0, 1, 2, 1, 1, 1, 2,
1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 0, 1, 2, 1, 1, 0, 1, 2, 1, 1, 0, 2, 0, 1, 1, 0, 1, 2, 1, 1, 0, 1, 2, 1, 1, 0, 1, 2, 1, 1, 0, 2, 0, 1,
1, 0, 2, 0, 1, 1, 0, 2, 0, 1, 0, 2, 0, 1, 1, 0, 2, 0, 1, 1, 0, 2, 0, 1, 1, 0, 2, 0, 1, 0, 2, 0, 1, 0, 2, 0, 1, 0, 0, 1, 0, 2, 0, 1,
0, 2, 0, 1, 0, 2, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0}
---------------------------------------------------------
Transition System Output
---------------------------------------------------------
Message: (Verificar memoria de programa, 1: powerOn(), PDCOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Verificar memoria de dados, 2:iniciar(), switchPDCOn/start60s & Idle and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Verificar status da alimentacao do PDC, 3:verificarHardware(), switchPDCOn/start60s & Idle and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Verificar status da alimentacao EPP HXi, 4: obterStatusAlimentacao(HX1,HX2), switchPDCOn/start60s & Idle and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Verificar temperatura atual do PDC, 5:gerarRelatoPOST(), switchPDCOn/start60s & Idle and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Verificar circuito de Cao-de-Guarda, 6:reconfigurar(), switchPDCOn/start60s & Idle and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, 7:mudarModoOperacao(), switchPDCOn/start60s & Idle and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, 8:ativarModuloPrincipal(), switchPDCOn/start60s & Idle and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & CountingTime and IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & CountingTime and SafeM_Entered)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & SafeM_Entered and Idle)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & IniM_POST)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & SafeM_Entered)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & SafeM_Entered and CountingTime)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & SafeM_Entered)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
```

```
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & CountingTime)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & Idle)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & CountingTime)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & CountingTime and SafeM_Entered)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, POSTOk & SafeM_Entered)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & CountingTime)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & SafeM_Entered)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, CountingTime)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & SafeM_Entered)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, switchPDCOn/start60s & CountingTime)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
```

```
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, tsinc & SafeM_VerOp)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
```

```
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Determinar Modo de Iniciacao, -, VER_OP_MODE/INFO_OP_MODE & SafeM_EPPsOff)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = DC
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
```

```
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Limpar memoria flash, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
```

```
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
```

```
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Iniciar memoria de configuracao com valores padrao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
```

```
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
```

```
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Gravar relato de eventos, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
```

```
--------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
--------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
--------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Avaliar processo de iniciacao, -, -)
Guard: bemsucedido = DC
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = False
--------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
--------------------------------------------------------
```

```
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Re-configurar, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Voltar ao Inicio, -, -)
Guard: bemsucedido = False
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
Message: (Mudar para modo de operacao SEGURANCA, -, -)
Guard: bemsucedido = True
Guard: modoiniciacao-poweron = True
---------------------------------------------------------
NUMBER OF STATES: 243
```

A.3 NuSMV File for Scenario 1

```
MODULE main

VAR

State: {_1powerOn$$--Verificarmemoriadeprograma--PDCOff,
 _2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST,
 _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDIniM_POST,
 _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered,
 _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--IniM_POST,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_VerOp,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--Idle,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime,
 _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered,
 _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp,
 _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--CountingTime,
 _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_EPPsOff,
 _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_VerOp,
 _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_EPPsOff,
 _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--_,
 _7mudarModoOperacao$$--DeterminarMododeIniciacao--SafeM_EPPsOff,
 _7mudarModoOperacao$$--DeterminarMododeIniciacao--_,
 _8ativarModuloPrincipal$$--DeterminarMododeIniciacao--_,
 _--Limparmemoriaflash--_,
 _--Gravarrelatodeeventos--_,
 _--Iniciarmemoriadeconfiguracaocomvalorespadrao--_,
 _--Avaliarprocessodeiniciacao--_,
 _--Re-configurar--_,
   _--VoltaraoInicio--_,
 _--MudarparamododeoperacaoSEGURANCA--_ };

bemSucedido: {dc,false,true};

modoIniciacaoPowerOn: {dc,false,true};

ASSIGN

init(State):= _1powerOn$$--Verificarmemoriadeprograma--PDCOff;

next(State):=
  case

State = _1powerOn$$--Verificarmemoriadeprograma--PDCOff & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST;

State = _2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST & bemSucedido=dc & modoIniciacaoPowerOn=dc :
{_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDIniM_POST,
_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered,
 _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle};

--State = _2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered;

--State = _2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle;

State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDIniM_POST & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : {_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--IniM_POST,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered};

--State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDIniM_POST & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime;
--State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDIniM_POST & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered;

State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : {_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_VerOp,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime};
```

```
--State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered;

--State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime;

State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 {_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--Idle,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime,
 _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime};

--State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle & bemSucedido=dc & modoIniciacaoPowerOn=dc
 : _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime;

--State = _3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle & bemSucedido=dc & modoIniciacaoPowerOn=dc
 : _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime;

State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--IniM_POST & bemSucedido=dc & modoIniciacaoPowerOn=dc
 : _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered;

State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : {_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered,
 _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp,
 _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--CountingTime};

--State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp;

--State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--CountingTime;

State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp;

State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_VerOp & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_EPPsOff;

State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime & bemSucedido=dc &
 modoIniciacaoPowerOn=dc : _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp;

State = _4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--Idle & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--CountingTime;


State = _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_VerOp;

State = _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_EPPsOff;

State = _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--CountingTime & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_VerOp;

State = _5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_EPPsOff & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--_;

State = _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_EPPsOff & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _7mudarModoOperacao$$--DeterminarMododeIniciacao--_;

State = _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_VerOp & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _7mudarModoOperacao$$--DeterminarMododeIniciacao--SafeM_EPPsOff;

State = _6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--_ & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _7mudarModoOperacao$$--DeterminarMododeIniciacao--_;

State = _7mudarModoOperacao$$--DeterminarMododeIniciacao--SafeM_EPPsOff & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _8ativarModuloPrincipal$$--DeterminarMododeIniciacao--_;

State = _7mudarModoOperacao$$--DeterminarMododeIniciacao--_ & bemSucedido=dc & modoIniciacaoPowerOn=dc :
 _8ativarModuloPrincipal$$--DeterminarMododeIniciacao--_;

State = _8ativarModuloPrincipal$$--DeterminarMododeIniciacao--_ & bemSucedido=dc & modoIniciacaoPowerOn=true :
 _--Limparmemoriaflash--_;

State = _8ativarModuloPrincipal$$--DeterminarMododeIniciacao--_ & bemSucedido=dc & modoIniciacaoPowerOn=false :
 _--Gravarrelatodeeventos--_;
```

```
State = _--Limparmemoriaflash--_ & bemSucedido=dc & modoIniciacaoPowerOn=true : _--Iniciarmemoriadeconfiguracaocomvalorespadrao--_;

State = _--Gravarrelatodeeventos--_ & bemSucedido=dc & modoIniciacaoPowerOn=false : _--Avaliarprocessodeiniciacao--_;


State = _--Iniciarmemoriadeconfiguracaocomvalorespadrao--_ & bemSucedido=dc & modoIniciacaoPowerOn=true
 _--Gravarrelatodeeventos--_;

State = _--Avaliarprocessodeiniciacao--_ & bemSucedido=true & modoIniciacaoPowerOn=false : _--Re-configurar--_;

State = _--Avaliarprocessodeiniciacao--_ & bemSucedido=false & modoIniciacaoPowerOn=false : _--VoltaraoInicio--_;


State = _--Gravarrelatodeeventos--_ & bemSucedido=dc & modoIniciacaoPowerOn=true : _--Avaliarprocessodeiniciacao--_;

State = _--Re-configurar--_ & bemSucedido=true & modoIniciacaoPowerOn=false : _--MudarparamododeoperacaoSEGURANCA--_;

State = _--Avaliarprocessodeiniciacao--_ & bemSucedido=true & modoIniciacaoPowerOn=true : _--Re-configurar--_;

State = _--Avaliarprocessodeiniciacao--_ & bemSucedido=false & modoIniciacaoPowerOn=true : _--VoltaraoInicio--_;


State = _--Re-configurar--_ & bemSucedido=true & modoIniciacaoPowerOn=true : _--MudarparamododeoperacaoSEGURANCA--_;


TRUE: State;

esac;


modoIniciacaoPowerOn:=case

(State=_1powerOn$$--Verificarmemoriadeprograma--PDCOff | State=_2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST |
 State=_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDIniM_POST |
 State=_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered |
 State=_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--IniM_POST |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_VerOp |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--Idle |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--CountingTime |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_EPPsOff |
 State=_6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_VerOp |
 State=_6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_EPPsOff |
 State=_6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--_ | State=_7mudarModoOperacao$$--DeterminarMododeIniciacao--SafeM_EPPsOff
 | State=_7mudarModoOperacao$$--DeterminarMododeIniciacao--_ ) : dc;

(State=_--Limparmemoriaflash--_ | State=_--Iniciarmemoriadeconfiguracaocomvalorespadrao--_ ) : true;

(State=_--Gravarrelatodeeventos--_ | State=_--Avaliarprocessodeiniciacao--_ | State=_--Re-configurar--_ |
 State=_--VoltaraoInicio--_ | State=_--MudarparamododeoperacaoSEGURANCA--_):{false,true};

TRUE: {dc,true,false};
esac;

bemSucedido:=case

(State=_1powerOn$$--Verificarmemoriadeprograma--PDCOff | State=_2iniciar$$--Verificarmemoriadedados--IdleANDIniM_POST |
 State=_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDIniM_POST |
 State=_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--CountingTimeANDSafeM_Entered |
 State=_3verificarHardware$$--VerificarstatusdaalimentacaodoPDC--SafeM_EnteredANDIdle |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--IniM_POST |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_Entered |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_EnteredANDCountingTime |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--SafeM_VerOp |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--Idle |
 State=_4obterStatusAlimentacao$HX1-HX2$--VerificarstatusdaalimentacaoEPPHXi--CountingTime |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_Entered |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_VerOp |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--CountingTime |
 State=_5gerarRelatoPOST$$--VerificartemperaturaatualdoPDC--SafeM_EPPsOff |
 State=_6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_VerOp |
 State=_6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--SafeM_EPPsOff |
 State=_6reconfigurar$$--VerificarcircuitodeCao-de-Guarda--_ | State=_7mudarModoOperacao$$--DeterminarMododeIniciacao--SafeM_EPPsOff
 | State=_7mudarModoOperacao$$--DeterminarMododeIniciacao--_ | State=_8ativarModuloPrincipal$$--DeterminarMododeIniciacao--_ |
 State=_--Limparmemoriaflash--_ | State=_--Gravarrelatodeeventos--_ | State=_--Iniciarmemoriadeconfiguracaocomvalorespadrao--_) :
 dc;
(State=_--Re-configurar--_ | State=_--MudarparamododeoperacaoSEGURANCA--_) : true;


State=_--VoltaraoInicio--_ : false;
TRUE: {dc,true,false};
esac;
```

APPENDIX B - ADDITIONAL INFORMATION ABOUT SWPDCpM CASE STUDY

B.1 Verified Properties

This section presents all the properties which have been verified for all scenarios of SWPDCpM. Table B.1 shows the properties descriptions and their respective CTL formalization for each one of the twelve scenarios analyzed.

Table B.1 - Verified Properties of SWPDCpM case study

Scenario 1	
Property Description	CTL formalization
1)The CTL process should receive the package of TC and validate its syntax, making it available to forwarding to the target AP. If the syntax of the remote control is invalid, an event report must be generated and sent to the CTL.TM_OUT. Otherwise, the remote control must be available for forwarding to the target application process. This function should be carried out in all computer software operation modes, except in Inactive and Automatic Shutdown modes	**Precedence Pattern and Scope After Q :** $\neg\exists[\neg (Q \wedge \neg\exists [\neg S \cup (\neg P \wedge \neg S)]) \cup (\neg Q \wedge \neg (Q \wedge \neg\exists [\neg S \cup (\neg P \wedge \neg S)]))]$ where, Q = ReceiveTC S = SyntaxErrorinTC P = GenerateReportEvent
2)The CTL process should receive the package of TC and validate its syntax, making it available to forwarding to the target AP. If the syntax of the remote control is invalid, an event report must be generated and sent to the CTL.TM_OUT. Otherwise, the remote control must be available for forwarding to the target application process. This function should be carried out in all computer software operation modes, except in Inactive and Automatic Shutdown modes	**Precedence Pattern and Scope After Q :** $\neg\exists[\neg (Q \wedge \neg\exists [\neg S \cup (\neg P \wedge \neg S)]) \cup (\neg Q \wedge \neg (Q \wedge \neg\exists [\neg S \cup (\neg P \wedge \neg S)]))]$ where, Q = ReceiveTC S = NotSyntaxErrorinTC P = SubmitTelecommand
Scenario2	
Property Description	CTL formalization
1)If the target AP is SCA, the TC should be sent to CTL. SCA_TC	**Response Pattern and Globally Scope :** $\forall\Box\, (\text{AP\_SCA} \rightarrow \forall\Diamond\, (\text{TC\_CTL.SCA\_TC}))$
2)If the target AP is CRX, the TC should be sent to CTL. CRX_TC	**Response Pattern and Globally Scope :** $\forall\Box\, (\text{AP\_CRX} \rightarrow \forall\Diamond\, (\text{TC\_CTL.CRX\_TC}))$
3)If the target AP is SYN, the TC should be sent to CTL. SYN_TC	**Response Pattern and Globally Scope :** $\forall\Box\, (\text{AP\_SYN} \rightarrow \forall\Diamond\, (\text{TC\_CTL.SYN\_TC}))$
4)If the target AP is HK, the TC should be sent to CTL. HK_TC	**Response Pattern and Globally Scope :** $\forall\Box\, (\text{AP\_HK} \rightarrow \forall\Diamond\, (\text{TC\_CTL.HK\_TC}))$
5)If AP is intended for own CTL, the TC should be executed immediately.	**Response Pattern and Globally Scope :** $\forall\Box\, (\text{AP\_CTL} \rightarrow \forall\Diamond\, (\text{TCExecutado}))$
6)If AP indicated in TC is unknown, a report of event should be generated and sent to CTL.TM_OUT and TC should be thrown	**Response Pattern and Globally Scope :** $\forall\Box\, (\text{Unknown\_AP} \rightarrow \forall\Diamond\, (\text{GenerateReportEvent}))$
7)If it is detected that the current operation mode does not enable the TC routing (or application), an event report should be generated and sent to CTL.TM_OUT	**Response Pattern and Globally Scope :** $\forall\Box\, ((\text{InactiveOperationMode} \vee \text{StartOperationMode} \vee \text{AutomaticShutDownOperationMode}) \rightarrow \forall\Diamond\, (\text{GenerateReportEvent}))$
Scenario3	
Property Description	CTL formalization
1)The package of TM should be forwarded as specified below: (i) A copy of the TM package, regardless of the source, should be forwarded to CTL.MMFS, where will be stored in a log file, except if TM is a TM_DM. The other copy continues as follows: (ii) If the package of TM is the scientific type, this must be forwarded to CTL.TC_CI_OUT. (iii) All other types of TM should be forwarded to CTL.TM_OP	**Precedence Pattern and Scope After Q :** $\neg\exists[\neg (Q \wedge \neg\exists [\neg S \cup (\neg P \wedge \neg S)]) \cup (\neg Q \wedge \neg (Q \wedge \neg\exists [\neg S \cup (\neg P \wedge \neg S)]))]$ where, Q = ForwardTM_CTL.MMFS S = PackageTMscientific

Continued on next page

	P = Forward_CTL.TM_CI
2)The package of TM should be forwarded as specified below: (i) A copy of the TM package, regardless of the source, should be forwarded to CTL.MMFS, where will be stored in a log file, except if TM is a TM_DM. The other copy continues as follows: (ii) If the package of TM is the scientific type, this must be forwarded to CTL.TC_CI_OUT. (iii) All other types of TM should be forwarded to CTL.TM_OP	**Precedence Pattern and Scope After Q :** $\neg\exists[\neg (Q \land \neg\exists [\neg S \cup (\neg P \land \neg S)]) \cup (\neg Q \land \neg (Q \land \neg\exists [\neg S \cup (\neg P \land \neg S)]))]$ where, Q = ForwardTM_CTL.MMFS S = OtherPackages P = Forward_CTL.TM_OP
Scenario4	
Property Description	CTL formalization
1) The CTL process must allow the enabling and disabling of the TM forwarding, through specific TC obtained from CTL.TC, intended for the CTL process itself	**Response Pattern and Globally Scope :** $\forall\Box\ (\text{ReceiveTC\_CTL.TC} \rightarrow \forall\Diamond\ (\text{EnableDisableForwardTM}))$
Scenario5	
Property Description	CTL formalization
1) When receiving a TC of CTL.TC requesting a consult of current operation mode, the value corresponding to the current operation mode must be submitted in the form of package TM for CTL.TM_OUT. Get current computational operation mode and generate package TM_RM in response	**Response Pattern and Globally Scope :** $\forall\Box\ (\text{ReceiveTC\_CTL.TC\_ConsultOperationMode} \rightarrow \forall\Diamond\ (\text{Generate\_TM\_RM}))$
Scenario6	
Property Description	CTL formalization
1) Upon receipt a TC, if the command change mode is valid, the process must change the operation mode	**Response Pattern and Scope After Q :** $\neg\exists\ [\ \neg\ (\ Q \land \forall\Box\ (P \rightarrow \forall\Diamond\ (S))) \cup (Q \land \neg\ (\ Q \land \forall\Box\ (P \rightarrow \forall\Diamond\ (S)))\ )]$ where, Q = ReceiveTC_CTL P = ValidChangeMode S = ChangeOperationMode
2) Upon receipt a TC, if the command change mode is invalid, the process must generate _TM_RE	**Response Pattern and Scope After Q :** $\neg\exists\ [\ \neg\ (\ Q \land \forall\Box\ (P \rightarrow \forall\Diamond\ (S))) \cup (Q \land \neg\ (\ Q \land \forall\Box\ (P \rightarrow \forall\Diamond\ (S)))\ )]$ where, Q = ReceiveTC_CTL P = InvalidChangeMode S = Generate_TM_RE
Scenario7	
Property Description	CTL formalization
1)The CTL process should distribute commands on/off requested by TC. The following sequence of operations must be performed: 1) Get command word on/off of the input TC; 2) Acting in hardware to perform the commands on/off through the CTL.ON_OFF_TC interface	**Precedence Pattern and Scope After Q :** $\neg\exists\ [\ \neg\ (Q \land (\neg\exists\ [\ \neg\ S \cup (P \land \neg\ S)])\ ) \cup (\ Q \land \neg\ (Q \land (\neg\exists\ [\ \neg\ S \cup (P \land \neg\ S)])\ )\ )]$ where, Q = ReceiveTC_CTL P = ObtainwordOnOff S = ActingHardware_CTL.ON_OFF_TC
Scenario8	
Property Description	CTL formalization
1)This function meets the telecommand of memory dump request. The following sequence of operations must be performed: 1) Get address and memory size required for dump; 2) Copy memory to dumpfile CTL.MMFS, reporting the progress of the operation through TM_VC, up to a maximum of 4MB; 3) Start transfer process of file generated in CTL.MMFS. Report TM_RE and ignore the request when: 4) The requested memory is invalid; 5) The requested size exceeds the maximum dump capacity; 6) A memory transfer process is still in progress	**Precedence Chain and Scope After Q :** $\neg\exists[\neg\ Q \cup (Q \land\exists\ [\neg S \cup P] \land\exists\ [\neg P \cup (S \land \neg P \land \exists\bigcirc\ (\exists[\neg T \cup (P \land \neg T)]))])]$ where, Q = ReceiveTC_CTL.TC_DumpMemory S = GetAddressMemorySize T = (¬(InvalidMemory ∨ InvalidSize ∨ ProcessTransferProgress) → (CopyMemoryDumpfile → Forward_TM_VC)) P = StartTransferDumpFile
2)This process is initiated by CTL automatically when a dumpfile is generated in CTL.MMFS. The following operation must be done in parallel with other software functions: 1) Generate TM_DM packages limited by the maximum size set to a TM package, considering that the last TM_DM package may have a size less than or equal to the maximum size of a packet TM; 2) Send TM_DM packages to CTL.TM_OUT at a rate not greater than four packets per second; 3) Report TM_VC indicating the completion status of the operation; 4) Delete dumpfile when finished. Report TM_RE and cancel transfer process when: 5) Occur I/O error in dumpfile; 6) A change occurs for computer mode operation that does not allow dump	**Precedence Chain and Scope After Q :** $\neg\exists[\neg\ Q \cup (Q \land\exists\ [\neg S \cup P] \land\exists\ [\neg P \cup (S \land \neg P \land \exists\bigcirc\ (\exists[\neg T \cup (P \land \neg T)]))])]$ where, Q = DumpFileGenerated S = (¬(ErrorIODumpFile ∨ ChangeOperationModeNDump) → GeneratePackageDM) T = SendPackagesTM_DM_CTL.TM_OUT P = (Report_TM_VC → DeleteDumpFile)

Continued on next page

memory during dump process	
Scenario9	
Property Description	CTL formalization
1)This function must meet the TC of request to report the current software version. The following sequence of operations must be performed: (i) Get the current version code of the software. (ii) Report the code version in the form of a TM__RV package	**Precedence Pattern and Scope After Q :** $\neg\exists\ [\ \neg\ (Q \wedge (\neg\exists\ [\ \neg\ S \cup (P \wedge \neg\ S)])\ )\ \cup\ (\ Q \wedge \neg\ (Q \wedge (\neg\exists\ [\ \neg\ S \cup (P \wedge \neg\ S)])\ )\ )]$ where, Q = ReceiveTC P = ObtainCurrentVersionCode S = ReportCode__TM__RV
Scenario10	
Property Description	CTL formalization
1)This function attempts to TC request of state of charge of a new version. A TM__RC report must be sent	**Response Pattern and Globally Scope :** $\forall\Box$ (ReceiveTC__ChargeState $\rightarrow \forall\Diamond$ (Report__TM__RC))
Scenario11	
Property Description	CTL formalization
1)This function attempts to TC request of charge initiation of new version of the software. When receiving TC, extract the data and validate them. After validate such data, the following operations must be performed: (i) Persist the data into CTL.MMFS (ii) Report TM__RE informing success or anomaly	**Precedence Chain and Scope After Q :** $\neg\exists[\neg\ Q \cup (Q \wedge\exists\ [\neg S \cup P] \wedge\exists\ [\neg P \cup (S \wedge \neg P \wedge \exists\bigcirc\ (\exists[\neg T \cup (P \wedge \neg T)]))])]$ where, Q = ReceiveTC__ChargeInitiation S = ValidateData T = PersistDataCTL.MMFS P = Report__TM__RE
Scenario12	
Property Description	CTL formalization
1) The SCA process should pass TC packets intended for the SCA as soon as they arrive in SCA.TC. Forward TC for SCA__TC__HK	**Response Pattern and Globally Scope :** $\forall\Box$ (ReceivePackagesTC__SCA.TC $\rightarrow \forall\Diamond$ (ForwardTC__SCA__TC__HK))
2) The SCA process should pass TM packets originated in SCA as soon as they arrive in SCA__TC__HK to be forwarded by the CTL process. Forward TM for SCA__TM	**Response Pattern and Globally Scope :** $\forall\Box$ (ReceivePackagesTM__SCA__TC__HK $\rightarrow \forall\Diamond$ (ForwardTM__SCA.TM))

B.2 Tansition System for Scenario 6

```
Node Children
---------------------------------------------------------
{1, 1, 1, 1, 1, 0}
---------------------------------------------------------
Transition System Output
---------------------------------------------------------
Message: (_tc$_$sgb$coma$__st$equals$130$coma$__sst$equals$2$coma$__modo$_$)
---------------------------------------------------------
Message: (_tc$_$modo$_$)
---------------------------------------------------------
Message: (_#accept#$dots$tm_vc)
---------------------------------------------------------
Message: (_#$not$__modo__ok#$dots$tm_re)
---------------------------------------------------------
Message: (_#modo__ok#$dots$altera$_$modo$_$)
---------------------------------------------------------
Message: (_encaminhar__tm)
---------------------------------------------------------
NUMBER OF STATES: 6
```

B.3 NuSMV File for Scenario 6

```
MODULE main

VAR

State: {
_tc$_$sgb$coma$__st$equals$130$coma$__sst$equals$2$coma$__modo$_$,
_tc$_$modo$_$,
_#accept#$dots$tm_vc,
_#$not$__modo__ok#$dots$tm_re,
_#modo__ok#$dots$altera$_$modo$_$,
_encaminhar__tm};


ASSIGN

init (State):= _tc$_$sgb$coma$__st$equals$130$coma$__sst$equals$2$coma$__modo$_$;

next (State):= case
State = _tc$_$sgb$coma$__st$equals$130$coma$__sst$equals$2$coma$__modo$_$: _tc$_$modo$_$;
State = _tc$_$modo$_$: _#accept#$dots$tm_vc;
State = _#accept#$dots$tm_vc: _#$not$__modo__ok#$dots$tm_re;
State = _#$not$__modo__ok#$dots$tm_re: _#modo__ok#$dots$altera$_$modo$_$;
State = _#modo__ok#$dots$altera$_$modo$_$: _encaminhar__tm;

TRUE: State;
esac;
```

APPENDIX C - XMITS USABILITY ASPECTS

To present the application, Eclipse (The Eclipse Foundation, 2015) was considered, as it is one of the most used IDE for Java programming. However, XMITS is compatible with any other Java development environment.

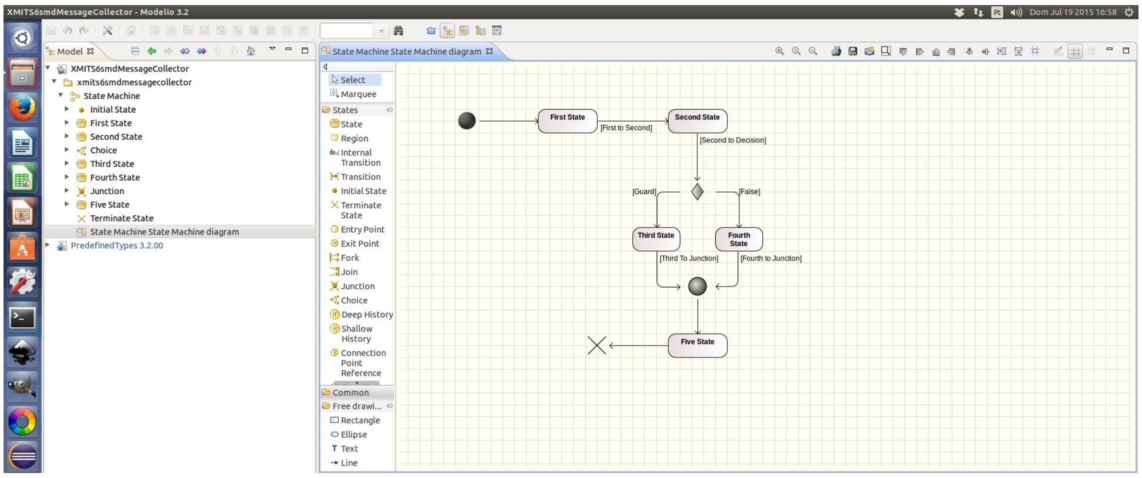
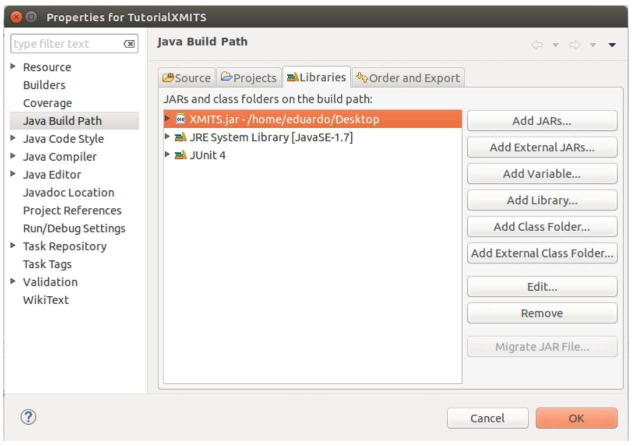


Figure C.1 - Screen of Modelio

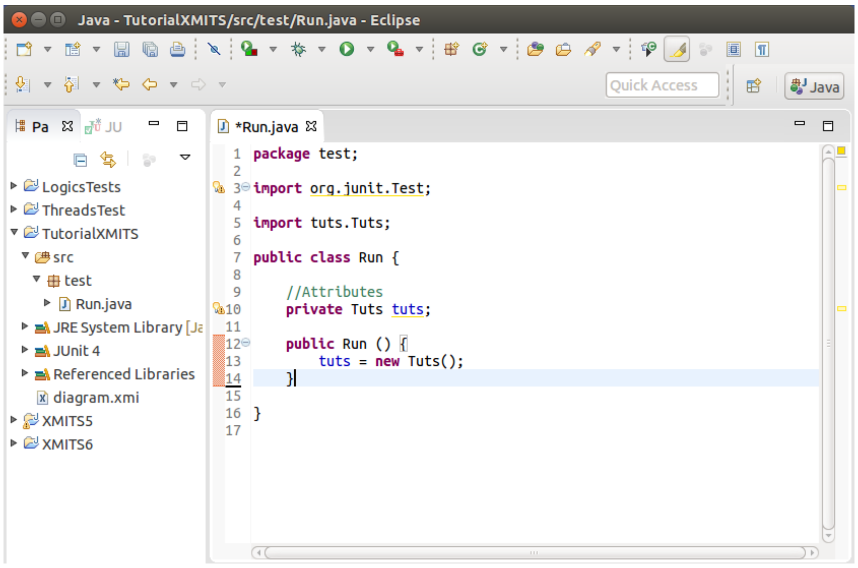
The first step is to create a UML sequence, activity or state machine diagram using Modelio 3.2, exactly as shown in Figure C.1. In the first version of XMITS, Papyrus (ECLIPSE.ORG, 2014) was used for modeling the UML diagrams. However, it leads to several inconsistencies in the XMI files. Then, after a few searches and tests, it was decided to change for the Modelio tool. Once the diagrams are finished, it is necessary to generate an XMI file for each one of them.



JARs and class folders on the build path:

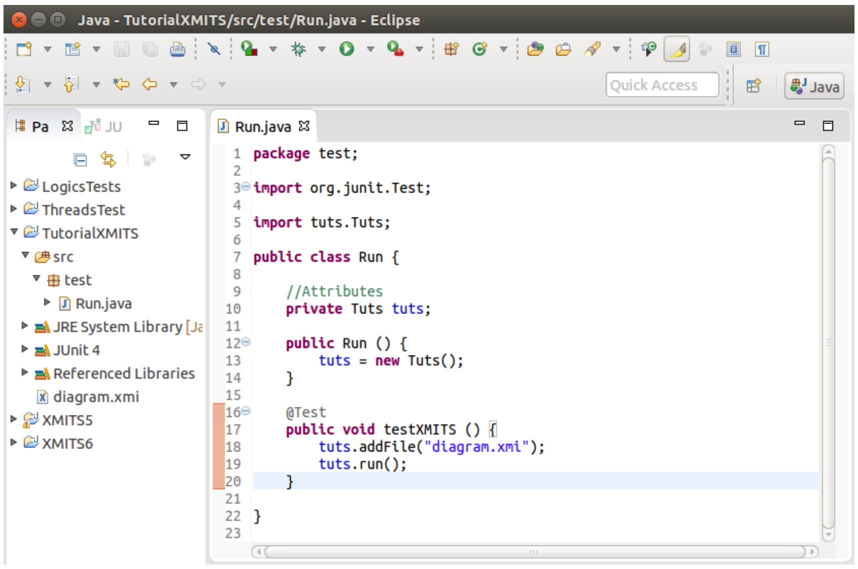
- XMITS.jar - /home/eduardo/Desktop
- JRE System Library [JavaSE-1.7]
- JUnit 4

Figure C.2 - Adding the build path in Eclipse



```java
package test;

import org.junit.Test;

import tuts.Tuts;

public class Run {

    //Attributes
    private Tuts tuts;

    public Run () {
        tuts = new Tuts();
    }

}
```

Figure C.3 - Preparing the class to run XMITS



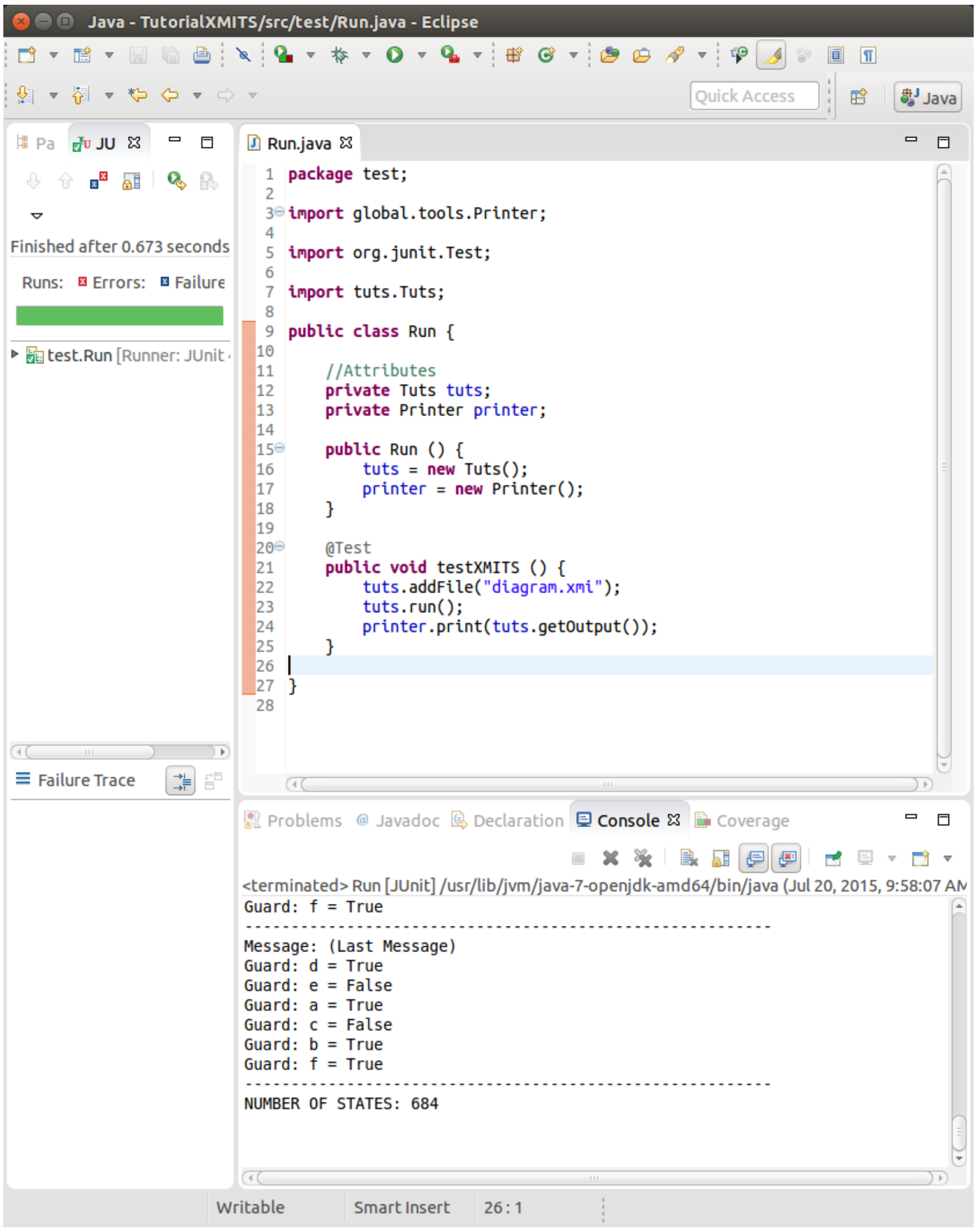
```java
package test;

import org.junit.Test;

import tuts.Tuts;

public class Run {

    //Attributes
    private Tuts tuts;

    public Run () {
        tuts = new Tuts();
    }

    @Test
    public void testXMITS () {
        tuts.addFile("diagram.xmi");
        tuts.run();
    }

}
```

Figure C.4 - Java class ready to run XMITS

XMITS is distributed in a Java package. This package should be added to the *build path* of the project that is being created, as can be seen in Figure C.2.

In the Java class where one wants to include the XMITS, a new object of `TUTS` type must be created, as in Figure C.3.

To run the XMI file exported by Modelio, it is necessary to call a function *add* and pass the name of the file as parameter of the function (Figure C.4).

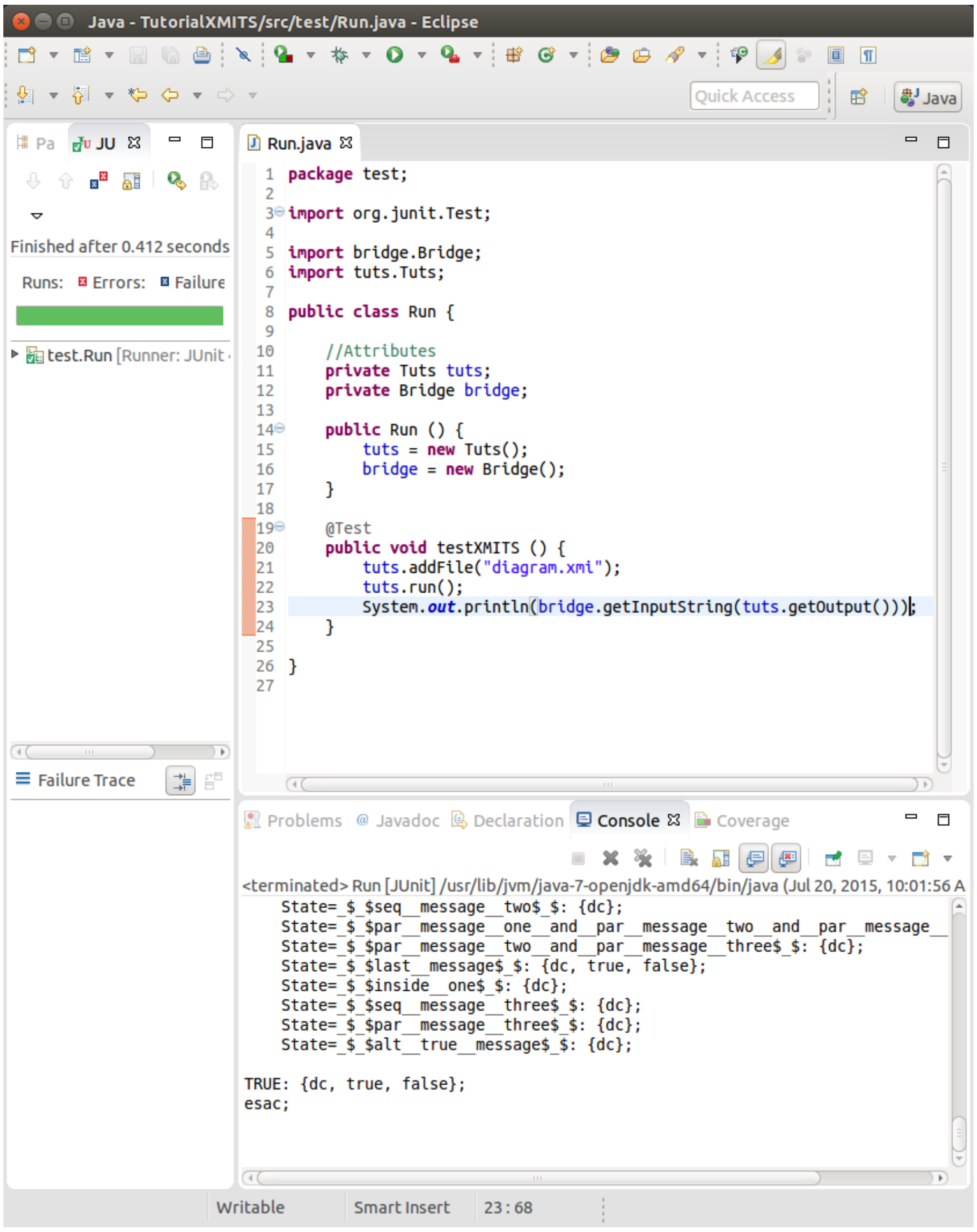


```
package test;

import global.tools.Printer;

import org.junit.Test;

import tuts.Tuts;

public class Run {

    //Attributes
    private Tuts tuts;
    private Printer printer;

    public Run () {
        tuts = new Tuts();
        printer = new Printer();
    }

    @Test
    public void testXMITS () {
        tuts.addFile("diagram.xmi");
        tuts.run();
        printer.print(tuts.getOutput());
    }

}
```

```
<terminated> Run [JUnit] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (Jul 20, 2015, 9:58:07 AM
Guard: f = True
---------------------------------------------------------
Message: (Last Message)
Guard: d = True
Guard: e = False
Guard: a = True
Guard: c = False
Guard: b = True
Guard: f = True
---------------------------------------------------------
NUMBER OF STATES: 684
```

Figure C.5 - Output as a Transition System displayed on the console

There are three modes to visualize the XMITS output: the first one is as a Transition System, using the TUTS module (displayed on the console), as explained in Subsection 4.1.5. The second one is as an input of NuSMV, using the Bridge module (displayed on the console). And the third one is as an input of NuSMV, but as a file. Each one of these outputs are displayed in Figures C.5, C.6, and C.7, respectively. Note that the changes to address each one of the three different outputs are implemented on the last line of the code.



Java - TutorialXMITS/src/test/Run.java - Eclipse

Finished after 0.412 seconds

Runs: Errors: Failure

test.Run [Runner: JUnit

Failure Trace

Run.java

```java
package test;

import org.junit.Test;

import bridge.Bridge;
import tuts.Tuts;

public class Run {

    //Attributes
    private Tuts tuts;
    private Bridge bridge;

    public Run () {
        tuts = new Tuts();
        bridge = new Bridge();
    }

    @Test
    public void testXMITS () {
        tuts.addFile("diagram.xmi");
        tuts.run();
        System.out.println(bridge.getInputString(tuts.getOutput()));
    }

}
```

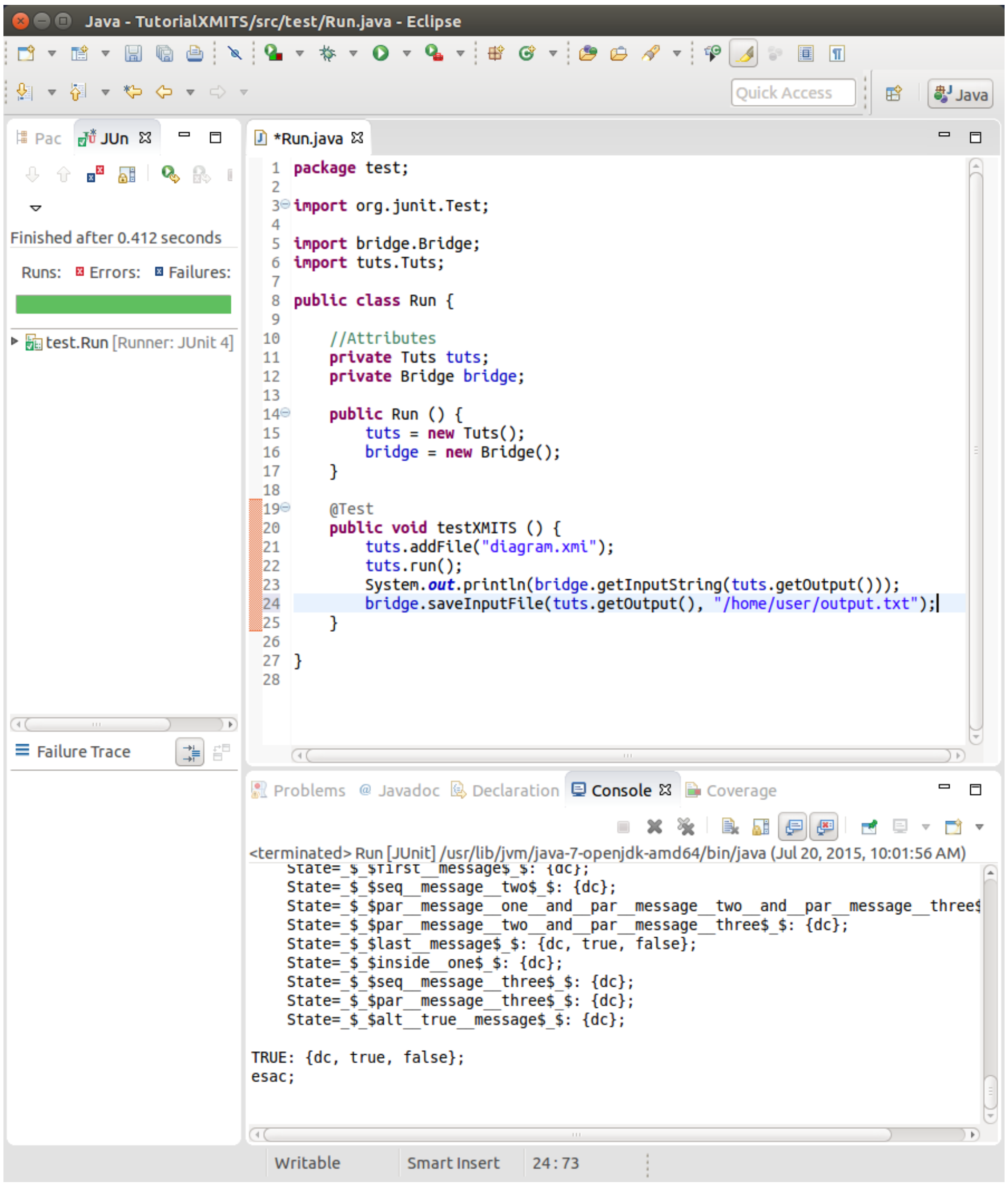
```
<terminated> Run [JUnit] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (Jul 20, 2015, 10:01:56 A
    State=_$_$seq__message__two$_$: {dc};
    State=_$_$par__message__one__and__par__message__two__and__par__message__
    State=_$_$par__message__two__and__par__message__three$_$: {dc};
    State=_$_$last__message$_$: {dc, true, false};
    State=_$_$inside__one$_$: {dc};
    State=_$_$seq__message__three$_$: {dc};
    State=_$_$par__message__three$_$: {dc};
    State=_$_$alt__true__message$_$: {dc};

TRUE: {dc, true, false};
esac;
```

Writable   Smart Insert   23 : 68

Figure C.6 - Output as an input of NuSMV displayed on the console

C.1 XMITS Class Diagrams

This section presents the class diagrams of Reader, Bridge, and Global modules of XMITS. The Converter and TUTS modules have a very large number of classes, which makes it impractical to show their class diagrams. The package diagrams of these two modules were shown in Chapter 4.



```java
package test;

import org.junit.Test;

import bridge.Bridge;
import tuts.Tuts;

public class Run {

    //Attributes
    private Tuts tuts;
    private Bridge bridge;

    public Run () {
        tuts = new Tuts();
        bridge = new Bridge();
    }

    @Test
    public void testXMITS () {
        tuts.addFile("diagram.xmi");
        tuts.run();
        System.out.println(bridge.getInputString(tuts.getOutput()));
        bridge.saveInputFile(tuts.getOutput(), "/home/user/output.txt");
    }

}
```

```
<terminated> Run [JUnit] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (Jul 20, 2015, 10:01:56 AM)
    State=_$_$first__message$_$: {dc};
    State=_$_$seq__message__two$_$: {dc};
    State=_$_$par__message__one__and__par__message__two__and__par__message__three$
    State=_$_$par__message__two__and__par__message__three$_$: {dc};
    State=_$_$last__message$_$: {dc, true, false};
    State=_$_$inside__one$_$: {dc};
    State=_$_$seq__message__three$_$: {dc};
    State=_$_$par__message__three$_$: {dc};
    State=_$_$alt__true__message$_$: {dc};

TRUE: {dc, true, false};
esac;
```

Figure C.7 - Output as an input file of NuSMV

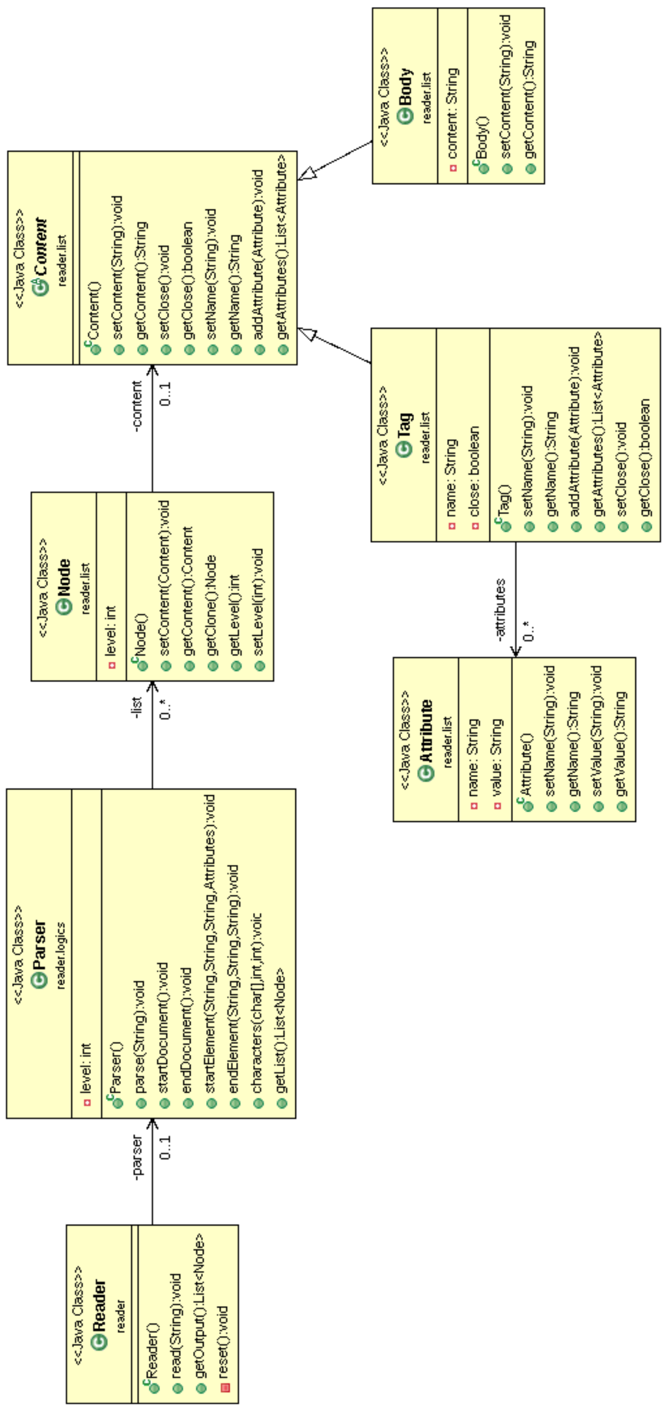


Figure C.8 - Class Diagram of Reader Module

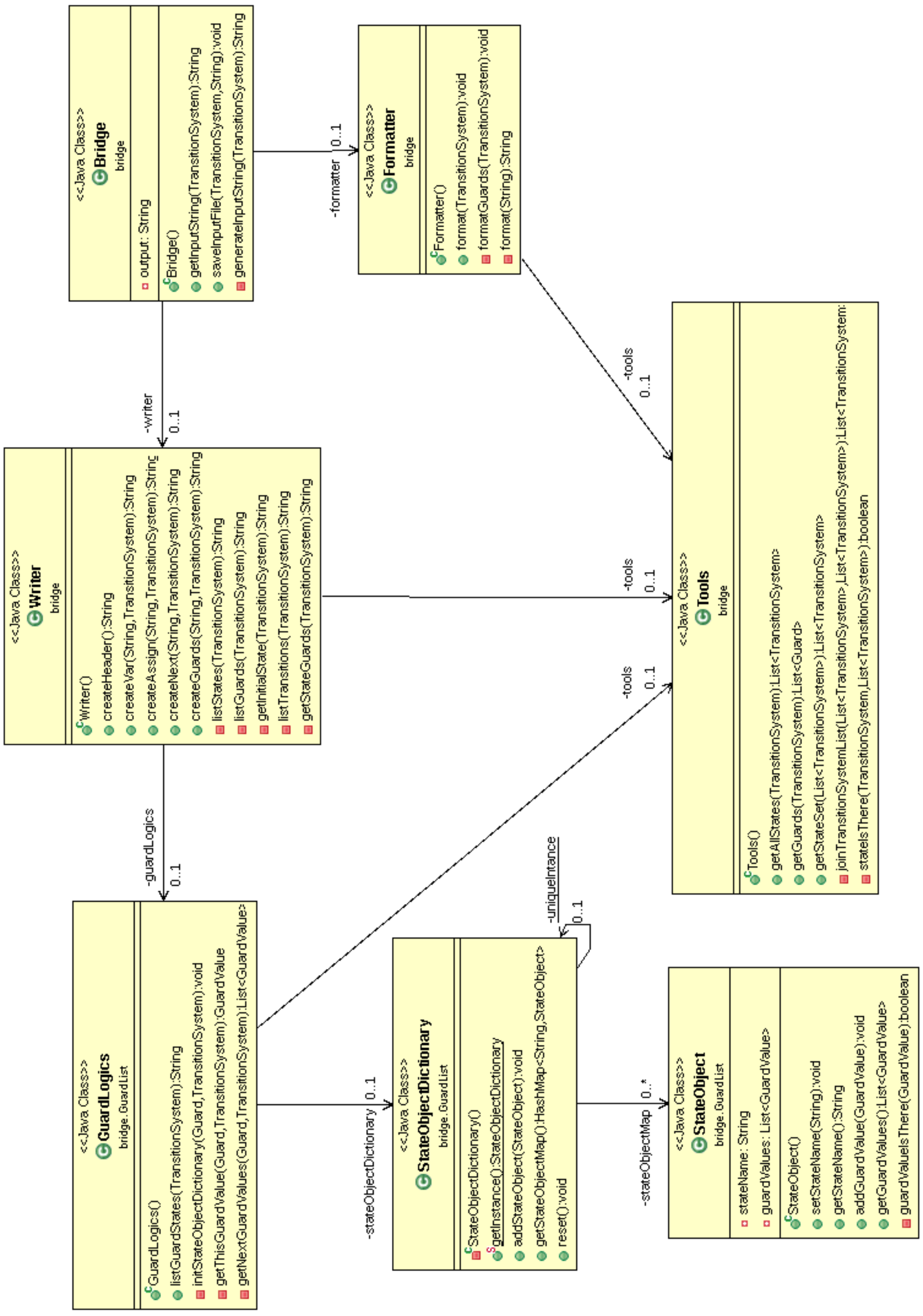

<<Java Class>>
Bridge
bridge
output: String
Bridge()
getInputString(TransitionSystem):String
saveInputFile(TransitionSystem,String):void
generateInputString(TransitionSystem):String
-formatter
0..1
<<Java Class>>
Formatter
bridge
Formatter()
format(TransitionSystem):void
formatGuards(TransitionSystem):void
format(String):String
-writer
0..1
<<Java Class>>
Writer
bridge
Writer()
createHeader():String
createVar(String,TransitionSystem):String
createAssign(String,TransitionSystem):String
createNext(String,TransitionSystem):String
createGuards(String,TransitionSystem):String
listStates(TransitionSystem):String
listGuards(TransitionSystem):String
getInitialState(TransitionSystem):String
listTransitions(TransitionSystem):String
getStateGuards(TransitionSystem):String
-tools
0..1
-tools
0..1
-tools
0..1
<<Java Class>>
Tools
bridge
Tools()
getAllStates(TransitionSystem):List<TransitionSystem>
getGuards(TransitionSystem):List<Guard>
getStateSet(List<TransitionSystem>):List<TransitionSystem>
joinTransitionSystemList(List<TransitionSystem>,List<TransitionSystem>):List<TransitionSystem>
stateIsThere(TransitionSystem,List<TransitionSystem>):boolean
-guardLogics
0..1
<<Java Class>>
GuardLogics
bridge.GuardList
GuardLogics()
listGuardStates(TransitionSystem):String
initStateObjectDictionary(Guard,TransitionSystem):void
getThisGuardValue(Guard,TransitionSystem):GuardValue
getNextGuardValues(Guard,TransitionSystem):List<GuardValue>
-stateObjectDictionary
0..1
<<Java Class>>
StateObjectDictionary
bridge.GuardList
StateObjectDictionary()
getInstance():StateObjectDictionary
addStateObject(StateObject):void
getStateObjectMap():HashMap<String,StateObject>
reset():void
-uniqueIntance
0..1
-stateObjectMap
0..*
<<Java Class>>
StateObject
bridge.GuardList
stateName: String
guardValues: List<GuardValue>
StateObject()
setStateName(String):void
getStateName():String
addGuardValue(GuardValue):void
getGuardValues():List<GuardValue>
guardValueIsThere(GuardValue):boolean

Figure C.9 - Class Diagram of Bridge Module

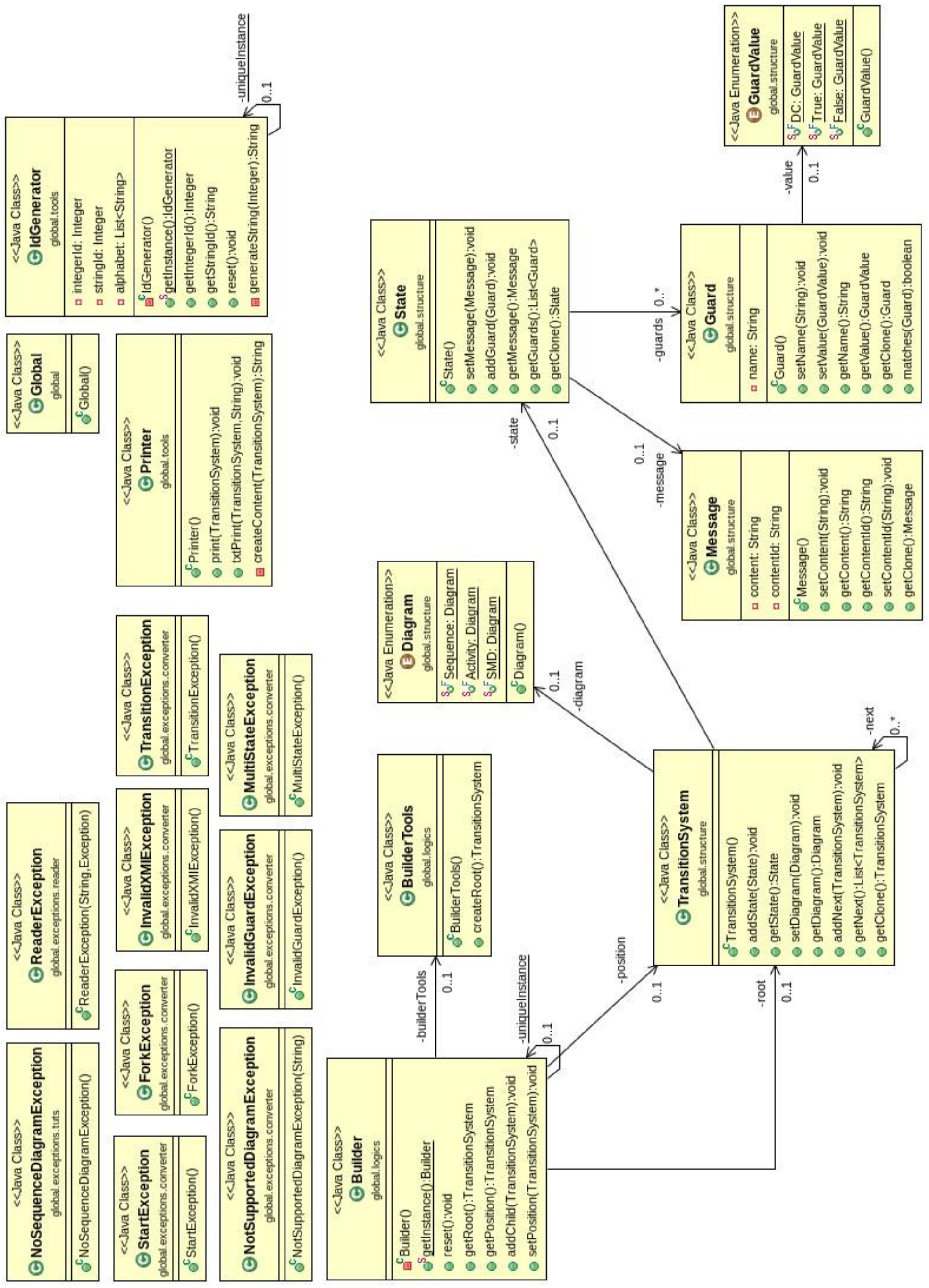


Figure C.10 - Class Diagram of Global Module