MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-11485-RPQ/779

COFI – CONFORMANCE AND FAULT INJECTION – A TESTING PROCESS INCLUDING TEST AND FULT CASES DERIVATION FOR SPACE APPLICATION SOFTWARE VALIDATION

Ana Maria Ambrosio
Eliane Martins*
Nandamudi Lanklapalli Vijaykumar
Solon Venâncio de Carvalho

* UNICAMP

INPE
São José dos Campos
2004

CoFI – Conformance and Fault Injection – a testing process including test and fault cases derivation for space application software validation

Ana Maria Ambrosio (INPE)

Dra. Eliane Martins (UNICAMP)

Nandamudi L. Vijaykumar (INPE)

Solon V. de Carvalho (INPE)

REPORT - INPE

October 2004

ABSTRACT

This research report describes a new testing process named CoFI (Conformance and Fault Injection), which has been defined as part of the doctorate program in Applied Computing. The CoFI integrates two existing testing approaches: conformance test and fault injection technique. This process is focused on the validation of implementations of the software services stated in the ECSS-E-70-41^{A} standard. These services, successfully adopted in many space missions, defines the application-level communication between on-board and ground applications. The European Space Agency has been setting standards for space mission software in the work of the European Committee for Space Standardization (ECSS). The standards lead to cost reduction of space mission software development for space agencies and industries over the years. The CoFI is a conformance testing process founded on the IS-9646 standard for ISO protocol, aimed to being applied for the ECSS-E-7041^{A} standard services. The advantage of this process is the generation of a re-usable abstract test suite which improves the effectiveness of testing the services. Reliability and convergence in the test and fault cases are increased the more the tests are being applied. A test cases and a fault case in the abstract format are illustrated for the *telecommand verification* service stated in the ECSS-E_4071A. Since conformance test is related to the certification process, the subject on this report may be applied to space application certification business.

COFI – CONFORMIDADE E INJEÇÃO DE FALHAS
UM PROCESSO DE TESTE INCLUINDO DERIVAÇÃO DE CASOS DE TESTE E DE FALHAS PARA VALIDAÇÃO DE SOFTWARE DE APLICAÇÕES ESPACIAIS

RESUMO

Este relatório de pesquisa descreve um novo processo de teste denominado CoFI (Conformance and Fault Injection), o qual está sendo definido como parte do programa de doutorado em Computação Aplicada. O CoFI integra duas abordagens de teste existentes: teste de conformidade e técnica de injeção de falhas. Este processo tem como objetivo principal a validação de implementações dos serviços de software especificados no padrão ECSS-E-70-41^{A}. Esses serviços, adotados com sucesso em muitas missões espaciais, definem a comunicação, no nível das aplicações, entre aplicativos de solo e aplicativos a bordo de satélites. A Agência Espacial Européia (ESA) vem estabelecendo padrões para software de missões espaciais no trabalho do Comitê Europeu para Padronização Espacial (ECSS). Os padrões proporcionam uma redução de custos no desenvolvimento do software realizado pelas agências espaciais e pela indústria ao longo dos anos. O CoFI é um processo de teste de conformidade baseado no padrão IS-9646 para protocolos ISO, cujo objetivo é ser aplicado para validação dos serviços padronizados pela ECSS-E-7041A. A vantagem desse processo é a geração de um conjunto de testes abstratos, o qual melhora a eficácia dos testes. A confiança nos casos de testes e casos de falhas e a convergência dos mesmos são aumentadas conforme os casos vão sendo mais aplicados (re-usados). Um caso de teste e um caso de falha na sua forma abstrata são ilustrados para o serviço *verificação de telecomando* especificado em ECSS-E_4071A. Como teste de conformidade está relacionado ao processo de certificação, o assunto abordado nesse relatório pode ser aplicado em certificação de aplicações espaciais.

Palavras chaves: processo de teste, teste de conformidade, injeção de falhas por software, validação de aplicações espaciais, serviços de comunicação solo-bordo.

SUMMARY

Pág.

LIST OF FIGURES 9

LIST OF TABLES 11

LIST OF ACROMNS AND ABBREVIATIONS 13

CHAPTER 1 INTRODUCTION 15

CHAPTER 2 SPACE COMMUNICATION AND THE ROLE OF THE PACKET UTILIZATION SERVICES 19

CHAPTER 3 COFI: CONFORMANCE-AND-FAULT-INJECTION TESTING PROCESS 23

3.1 CONFORMANCE TESTING 23

3.1.1 POSITIVE POINTS FOR USING THE ISO-9646 24

3.2 SOFTWARE-IMPLEMENTED FAULT-INJECTION TECHNIQUE 24

3.2.1 POSITIVE POINTS FOR USING SWIFI 25

3.3 THE COFI ACTIVITY FLOW 25

3.4 THE ACTIVITY DESCRIPTION 28

3.4.1 DEFINE TEST PURPOSE 28

3.4.2 SPECIFY ABSTRACT TEST SUITE 29

3.4.3 SELECT AND PARAMETERIZE THE TEST CASES 30

3.4.4 EXECUTE THE TEST CAMPAIGNS 31

3.4.5 ANALYZE TEST RESULTS 31

3.4.6 GENERATE THE TEST REPORT 31

CHAPTER 4 APPROACH FOR STATE-BASED TEST AND FAULT CASES GENERATION 33

CHAPTER 5 ABSTRACT TEST AND FAULT CASES FOR THE ECSS-E-70-41 SERVICES 37

5.1 TEST CASE: 39

5.2 FAULT CASE 43

CHAPTER 6 RELATED WORK 49

CHAPTER 7 CONCLUSION 51

REFERENCES 53

APPENDIX A - OVERVIEW OF THE IS-9646 - CONFORMANCE TESTING METHODOLOGY AND GFRAMEWORK 59

APPENDIX B - TEST PUPOSES FOR THE STANDARD PUS SERVICES 67

APPENDIX C - THE EVENT/STATE MATRIX FOR THE *TELECOMMAND VERIFICATION* SERVICE 69

APPENDIX D - TEST CASES GENERATED BY THE CONDADO TOOL FOR THE *TELECOMMAND VERIFICATION* SERVICE 73

LIST OF FIGURES

2.1 - Elements of a space mission 19

2.2 - Telecommand system layers 20

3.1 - CoFI Testing Process 27

3.2 - Ferry-Injection Architecture 30

5.1 - Points of Control and Observation for the *telecommand verification* service 39

5.2 - *Normal finite state diagram* mapping the normal situations of the *telecommand verification* service 41

5.3 - *Exceptional finite state diagram* mapping the Exceptional Situations of the *telecommand verification* service 44

5.4 - Transition Tree of the mapping the Exceptional Situations of the *telecommand verification* service.....45

A.1 - Standardization Organization vs. Client vs. Testing laboratory 60

A.2 – ISO conformance testing process 61

A.3 – Local Test Method in Single-Party Context 64

LIST OF TABLES

1.1 – Incident and causes in space missions 16
3.1 – Concepts and acronyms 28
5.1 – Capability Set of the *telecommand verification* service 38
5.2 – Guard conditions for the *telecommand verification* service 40
5.3 – Dynamic Behavior Description Test Case 43
5.4 – Dynamic Behavior Description Fault Case 47
A.1 – ISO Conformance testing concepts 62
A.2 – Concepts related with testing methods 65
C.1- Event/State Matrix for the *telecommand verification* service 70

LIST OF ACROMNS AND ABBREVIATIONS

ATS	Abstract Test Suite
CCSDS	Cooperation Committee for Space Data Standardization
CLCW	Command Link Control Word
CLTU	Command Link Transfer Unit
CORBA	Common Object Request Broker Architecture
ECSS	European Cooperation for Space Standardization
FIC	Fault Injection Controller
FIM	Fault Injection Module
FSM	Finite State Machine
ICS	Implementation Conformance Statement
IXIT	Implementation extra Information for Testing
MSC	Message Sequence Chart
OBDH	On-Board Data Handling
PCO	Point of Control and Observation
PETS	Parameterized Executable Test Suite
PLUTO	Procedure Language for User in Test and Operation
PUS	Packet Utilization Service
SAMSTAG	SDL and MSC based test case generation
SDL	Specification Description Language
SLE	Space Link Extension
SWIFI	Software Implemented Fault Injection

TC	Telecommand
TM	Telemetry
UML	Unified Modeling Language
UUT	Unit Under Testing

CHAPTER 1

INTRODUCTION

The development of space application software is very expensive and one reason is that their one-time application prevents cost reduction. These kinds of software need to be reusable in order to make them more profitable for the space industry. In (Mazza., 2000) it is pointed out that space software requires standardization by the fact that nowadays more and more tasks are delegated to software. So, in order to avoid cost and schedule overruns and to enable production of high-quality safe software, the European Space Agency (ESA) together with industry have been setting standards for management, quality assurance and engineering branches since 1993.

This set of standards defined by the European Cooperation for Space Standardization (ECSS) may be found in (ECSS, 2003). In the engineering branch of the ECSS one may find a recently defined standard, the ECSS-E-70-41^{A} (ECSS_PUS, 2003) which specifies services for the application-process communication between ground and on-board systems. These services are referred to as Packet Utilization Services (PUS) and they have already been used for the following space missions: X-Ray Multi-Mirror (XMM), Meteosat Second Generation (MSG), International GammaRay Astrophysics Laboratory (INTEGRAL), Automated Transfer Vehicle (ATV), ESA's Project for On-Board Autonomy satellite (PROBA), ROSETTA, MARS EXPRESS, CryoSat, Gravity Field and Steady-State Ocean Circulation Explorer (GOCE), the Euro-Russian space collaboration GALILEO Program for global satellite navigation, etc. (Merri, 1996), (Merri, 2002).

Since the PUS services are publicly available as standards, it is essential to assure that implementations (computer programs) of these services conforms to their corresponding specifications. Moreover, it would be more cost-effective to establish a standard set of previously defined test cases rather than defining test cases for each implementation in isolation. The need of investments in space software validation is illustrated in the list of

problems with software testing and validation that caused loss of very important and expensive space missions (Mazza, 2000). TABLE 1.1 summarizes these problems.

TABLE 1.1 - Incident and causes in space missions.

Project	Incident	Root Cause
ARIANE 501, June 1996	Launch failure, loss of 4 CLUSTER spacecraft	Lack of proper software and system validation
Mars Climate Orbiter, Sept. 1998	Loss of Orbiter	Confusion between imperial and metric units. Lack of proper software/system validation
Mars Polar Lander, Dec.1998	Loss of Lander	SW error forced premature shutdown of landing engine: lack of proper software validation
Titan-4B, April 1999	Failure to put USAF Milstar spacecraft into useful orbit	Centaur upper stage failed to ignite. Decimal-point error in the guidance system. Lack of proper software validation
Delta-3, April 1999	Launch aborted	Failure of on-board SW to ignite main engine. Lack of proper software validation
PAS-9, Arch 2000	Loss of ICO Global Communication F-1 spacecraft	Error in a update to ground software. Lack of proper software validation

FONTE: Mazza, (2000).

In this report we present a testing process focussed on the validation of the PUS services, named CoFI - conformance-and-fault-injection.

A testing process is a "course of action to be taken to perform testing"[1]. It consists of several actions or activities since specifying the test cases, adjusting the test cases to the test mean, executing the test, analyzing the results and producing reports about the test execution. The CoFI covers all these test activities.

The CoFI is based in the IS-9646 standard, a conformance process for protocol certification established by the International Standard Organization (ISO). Additionally, it uses algorithms, from academic research, for space-based specification that have produced good results about automatically specifying test cases from formal notation-based specification, thereby significantly reducing time and effort with tests and assuring a greater coverage of the specification. However, getting a formal

[1] According to the process definition in IEEE Software Standards.

specification, acceptable for the formal algorithms, is hard and time-taking work, requiring many iterations. Therefore, the specification should be complete, leading to difficulty in getting it (Lai, 2002).

In order to avoid the problems of having a formal specification, in the CoFI process normal and exceptional behavior are independently modeled from the service description. Additionally, it extends the functional conformance (given in the IS-9646 protocol conformance testing standard) with the software implemented fault-injection (SWIFI) to accelerate the rate of exceptional events.

The CoFI process comprises some characteristics that should be useful to support space software validation/certification.

The structure of this report is as follows:

- Chapter 2- - *Space Communication and the Role of the Packet Utilization Services*: presents the context in which the ECSS-E-7041A services standard is used;

- Chapter 3 - *CoFI: Conformance-and-Fault-Injection Testing Process*: describes the conformance-and-fault-injection testing process, its activities, the adopted concepts and justifications;

- Chapter 4 - *Approach for state-based test and fault cases generation:* addresses the approach for specifying abstract test and fault cases that comprises an activity of the CoFI process;

- Chapter 5 - *Abstract test and fault cases for the ECSS-E-70-41 Services:* illustrates the application of the approach for specifying abstract tests and fault cases for the *telecommand verification* service given in the ECSS-E-7041[A] standard**;**

- Chapter 6 - *Related Work*: discusses related works;

- *Chapter 7 - Conclusion:* presents conclusions and points out future works;

- Appendix A - *Overview of the IS-9646: conformance testing methodology and framework*: summarizes the main concepts of the IS-9646 standard for ISO protocol;

- Appendix B - *Test purposes for the Standard PUS Services*: lists the test purposes defined for the PUS services;

- Appendix C – The *Event/State Matrix for the Telecommand Verification Service*: shows the complete transition matrix specifying the expected behavior of the *telecommand verification* service;

- Appendix D - *Test Cases Generated By The Condado Tool for the Telecommand Verification Service*: lists the test case suite automatically generated by the Condado tool for the normal behavior of the *telecommand verification* service.

CHAPTER 2

SPACE COMMUNICATION AND THE ROLE OF THE PACKET UTILIZATION SERVICES

Communication between the ground and on-board systems is provided by many subsystems (hardware and software) at several communication levels. FIGURE 2.1 shows the main elements of a space mission: satellite, ground station and satellite control system. One overview of the satellite systems may be found in (Larson, 1992).

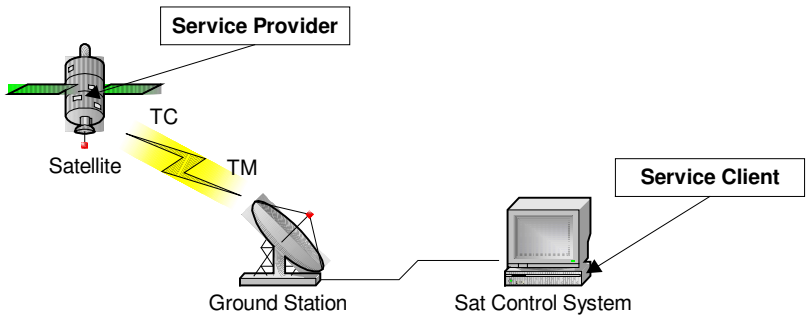

FIGURE 2.1- Elements of a space mission.

The ECSS-E-7041A specifies services to be provided by the satellite (service provider) during its communication with the Satellite Control System (service client). The packet utilization services (PUS) may also be applied in different software within a space mission. For example, they may be applied to the embedded software of the On-board Data Handling Computer (OBDH) as well as to the satellite simulator. The simulator provides the same set of services in the module simulating the OBDH behavior, when it is used to replace the real satellite during the Satellite Control System testing and operator training.

The PUS services are characterized by a set of reports and requests transferred from/to on-board application processes. For the requests and reports to be effective, ground and on-board communication uses the layered CCSDS-defined Telecommand System (CCSDS, 2000), (CCSDS, 2001a), (CCSDS, 2001b), shown in gray in FIGURE 2.2. This system comprises 5 layers, from the Physical to the Packetization layers, providing

all the necessary controls to accurately deliver the telecommand (TC) packet that has been generated on the ground to the on-board application process. The Telemetry (TM) Packets deliver the service reports that have been generated on-board to the on-ground application process. The Command Link Control Word (CLCW) and the Command Link Transfer Unit (CLTU) are intermediary formats supporting the protocol communication

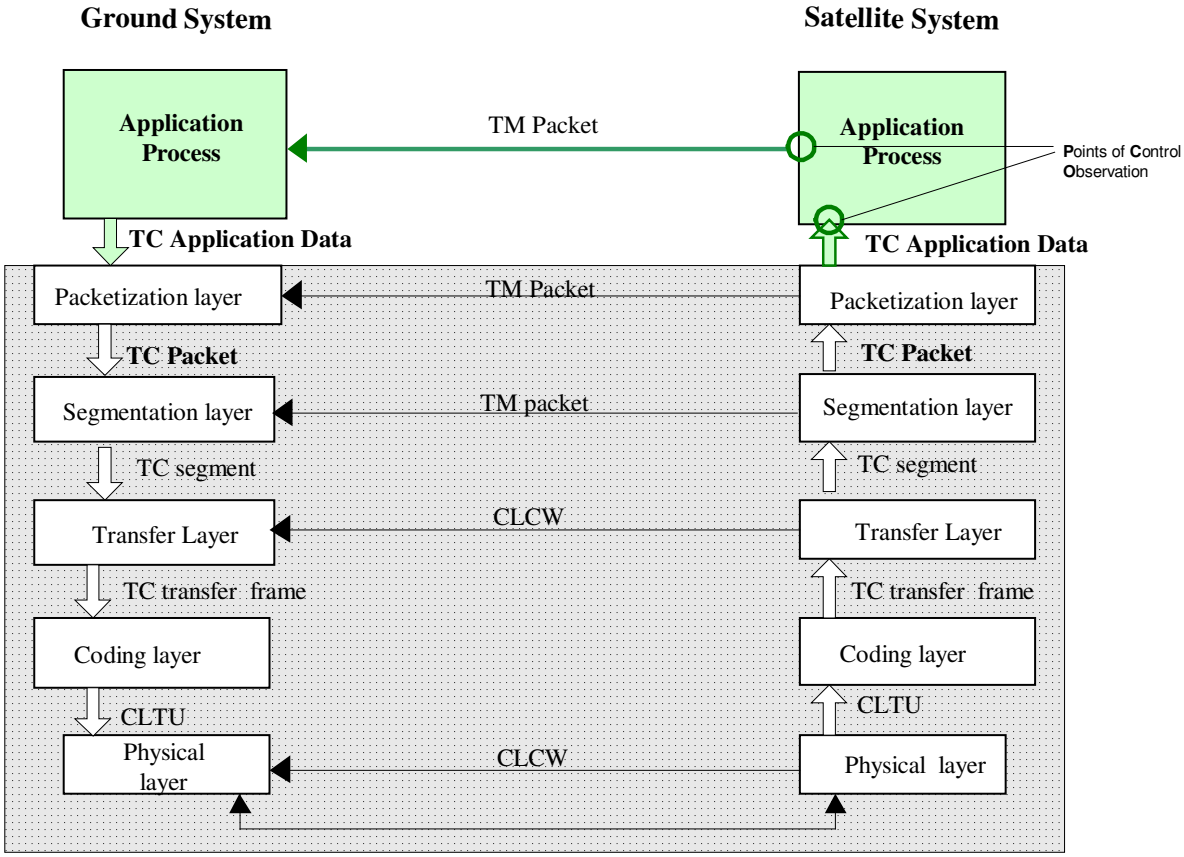

FIGURE 2.2 - Telecommand system layers.

SOURCE: ECSS-E-70-41 (2003, p. 29).

Although, space application software differ from ISO protocols, as they run under adverse environmental conditions, they are only recently being specified as standards, and many times they are uniquely implemented, the structure of the PUS services (implemented into the application processes) over the leveled CCSDS protocols is similar to the ISO-OSI basic reference model (ISO, 1994). So, the conformance testing

process defined in the IS-9646 standard seems suitable to the CoFI process, since points of control and observation (PCOs) be established. The PCOs define the system testability as they allow the tester to control and observe the occurrence of testing events.

The application process that implements the PUS services and its commonest points of control and observation are depicted in the Satellite System side in FIGURE 2.2. An implementation of such an application process is the target of the testing process presented in this report.

CHAPTER 3

COFI: CONFORMANCE-AND-FAULT-INJECTION TESTING PROCESS

The CoFI testing process integrates two existing approaches: conformance testing and software-implemented fault injection (SWIFI) technique. This process is partially supported by prototyped tools developed in previous research at the ATIFS project (ATIFS, 2002), (Martins, 2003a). A summary of the work that has been carried out in the ATIFS and that motivated us to define the CoFI process for space application software may be find in (Ambrosio, 2004).

This chapter discusses conformance testing and SWIFI techniques, presents the CoFI activities flow and describes the activities of the process.

3.1 Conformance Testing

The International Standard Organization (ISO) and the International Telecommunication Union (ITU-T) have standardized procedures, for communication protocols, with practical guides for conformance testing. These procedures, stated in the IS-9646 standard for Conformance Testing Methodology and Framework (IS 9646, 1991), (Baumgarten, 1994), have allowed testing laboratories around the world to perform testing and certification of ISO protocols.

> *Conformance testing* consists of checking whether an Implementation Under Test (IUT) satisfies all the specified *conformance requirements*. These requirements must be explicitly mentioned in the standard, as they determine what should be tested.
>
> The *conformance requirements* are checked by running a number of tests against the IUT (these tests are referred as *test cases*).

In the IS-9646, the testing process comprises activities in which abstract test suites are derived to certify a protocol implementation. The implementation to be tested is one or more OSI protocols, in a single layer or in one or more OSI layers. A set of abstract test cases is generated according to the protocol specification. Additional information about

the specific implementation is then added to the test cases only before the tests run. In this way the abstract test suite may be re-used, thereby allowing test results to be compared.

In short, the IS-9646 presents concepts and solutions for certification of protocol implementations that may eventually be applied for certification of PUS service implementations as well. Appendix A summarizes the IS-9646 standard contents.

3.1.1 Positive Points for Using the ISO-9646

The reasons for adopting the IS-9646 definitions in the CoFI process are the following. Firstly, it incorporates a great deal of practical experience from test experts. IS-9646 has been used by communication industries and by testing laboratories that perform protocol testing of commercial implementations for approximately 20 years. Secondly, some concepts of the IS-9646 have already been adopted in the validation of a Space Link Extension (SLE) (CCSDS, 2002) service implementation. Validation of the SLE implementation in the cross-supporting program carried out by ESA and Jet Propulsion Laboratory (JPL) is presented in (Mertens, 2002). The TTCN[2] and the commercially available ITEX[3] tool to edit and document the testing procedures were used for helping the validation of the referred SLE implementation.

3.2 Software-Implemented Fault-Injection Technique

Space software applications require a high degree of reliability especially when running on-board of a spacecraft. The software-implemented fault-injection technique (SWIFI) has been widely used in the dependability validation process of critical systems (Voas, 1998). SWIFI has been adopted to accelerate the rate of fault events during testing,

[2] In earlier versions, TTCN stands for Tree and Tabular Combined Notation. The new meaning is Testing and Test Control Notation apt for TTCN-3.
[3] ITEX Graphical Editor (Telelogic AB, Sweden)
http://www.telelogic.com/products/tau/ttcn/index.cfm
I supports test case specification in TTCN & ASN.1 and checks syntax and semantics of the test suite.

which enables software designers to identify and remove the faults of the implementation under testing. For fault removal, external faults are submitted to verify the system behavior under the presence of the faults, and to remove the uncovered failures in the implementation and/or in the design. The SWIFI may also be used for fault forecasting purposes, in which the faults are injected to rate the efficiency of the operational behavior of the implemented fault tolerant mechanisms (Arlat, 1990).

3.2.1 Positive Points for Using SWIFI

The first reason to incorporate the software-implemented fault injection (SWIFI) in the testing process was to accelerate the rate of problems in the unit under testing that would be difficult (in time) to happen. Moreover, the SWIFI allows testers to reproduce common faults encountered in the space context like memory and communication faults. Another reason is to improve the evaluation of the system not only for conformance but also for observation of its behavior under the presence of external faults. Although, the likelihood of a particular fault (or the entire faultload) can not be determined, and hence absolute conclusion about the software dependability cannot be drawn, this technique represent a step towards safer software behavior. SWIFI has also the advantage of being easily expandable for new classes of faults, and it is free of physical interference. In (Madeira, 2002) an experimental evaluation of a COTS system for space applications is presented. The fault injection was used to detected Single Event Upsets (SEU), which are transient errors caused by space radiation. The SWIFI is adopted to represent common space environment faults under which such kind of systems are submitted when in its operational phase.

3.3 The CoFI activity flow

The CoFI testing process comprises six activities: define test purposes, specify abstract test suite, select and parameterize test cases, execute test campaigns, analyze test results, generate test report. presented bellow. These activities are organized into three phases:

- preparation for testing,

- test operations and,
- test report production.

The phases and activities have been chosen to comply with the IS-9646.

The CoFI activities flow is illustrated in Figure 3.1. in parallel with the implementation activity. The testing activities are represented in rectangular boxes and the arrows show what is received and produced by the activity. The activities of the first phase, in gray boxes, differ from the IS-9646 by dealing with fault definitions and formal specifications that have been added to automate the process. The concepts and acronyms used in the activities flow description that are based on IS-9646, are summarized in Table 3.1.

According to the IS-9646, the standard specification should provide the conformance requirements for guiding both the implementation and the testing. Implementation Conformance Statement (ICS) and Implementation extract Information for Testing (IXIT) accompanying the unit under test (UUT[4]) have to be precisely informed and delivered to the tester.

[4] UUT is used instead of IUT (implementation under test) to comply with space jargon.

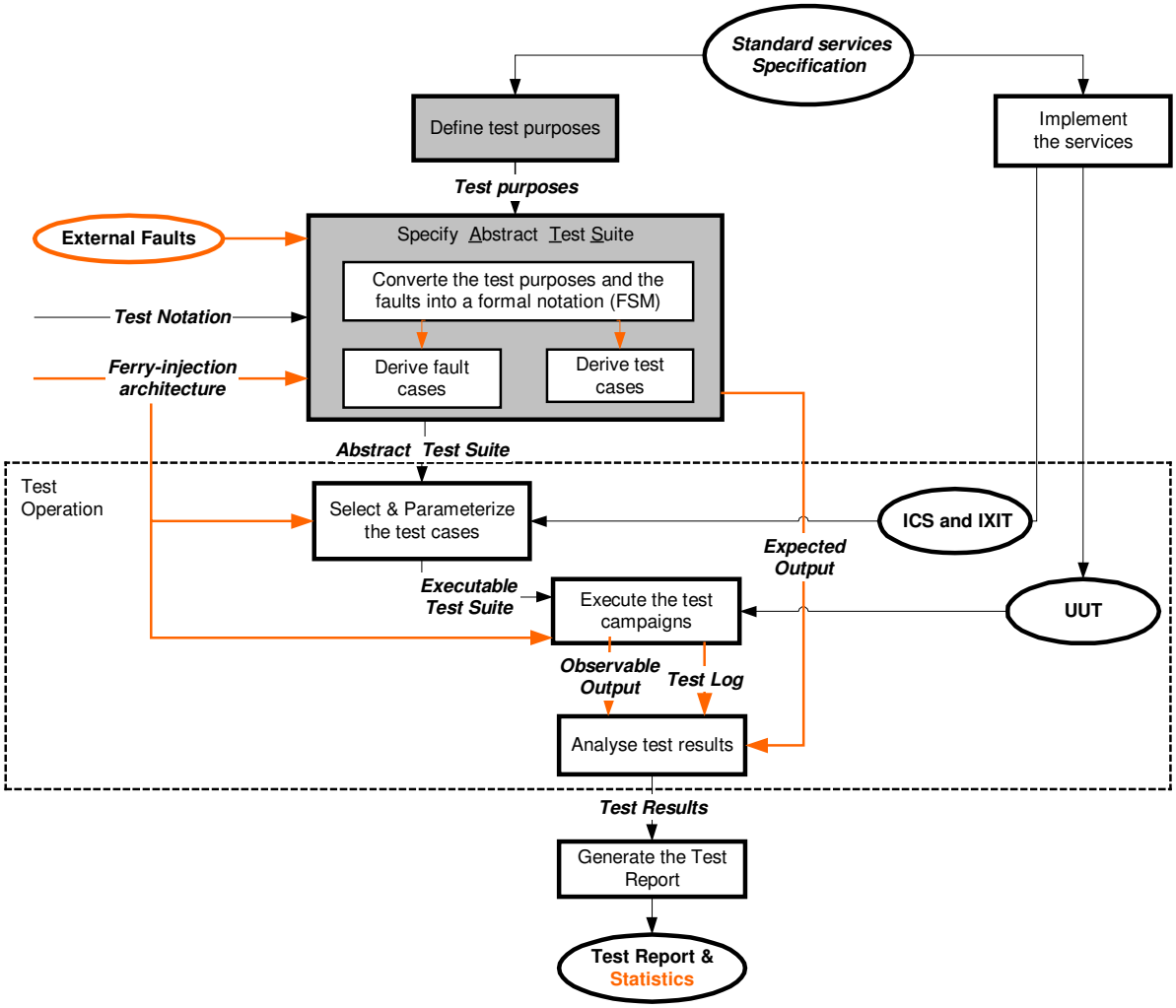

FIGURE 3.1- CoFI Testing Process.

TABLE 3.1 - Concepts and acronyms.

Test Purpose	A prose description of a well-defined objective of testing, focusing on a single conformance requirement or a set of them.
Test Notation	The notation used to describe the abstract test case - implementation independent.
Abstract Test Suite (ATS)	A set of Abstract Test Cases. An ATS refers to a particular test method.
Abstract Test Case	A complete and independent specification of the actions required to achieve a specific test purpose. It is complete in the sense that it is sufficient to enable a test verdict unambiguously for each potentially observable test outcome. It is independent in the sense that it should be possible to execute the derived executable test case in isolation from other such test cases.
Implementation Conformance Statement (ICS)	A statement made by the supplier of an implementation, listing the capabilities that have been implemented.
Implementation extra Information for Testing (IXIT)	A statement made by the supplier of an implementation which contains all of the information related to the UUT and its testing environment, necessary to execute the test suites.
Parameterized Executable Tests Suite (PETS)	A selected set of test cases, in which all test cases have been parameterized in accordance with the relevant ICS.
Test Report	A document giving details of the testing that has been carried out, using a particular ATS. It lists all the abstract test cases, identifies those for which corresponding executable test cases were run, and presents the test verdicts.
UUT	Unit under Test

3.4 The activity description

The CoFI testing process activities are described bellow.

3.4.1 Define Test Purpose

The *preparation for testing* phase starts with the definition of *test purposes* activity. This activity receives the service standard specification and produces the test purposes. According to IS-9646, a test purpose is the test objective. Here, the test purpose summarizes the functionality of one service stated in the standard. For the "*telecommand verification*" service standardized in ECSS-E-7041A, the corresponding test purpose is "*to verify the telecommand execution*", which defines the objective of this service.

3.4.2 Specify Abstract Test Suite

This activity receives the test purposes and produces an abstract test suite (ATS). An ATS is provided for each ECSS-E-70-41^{A} service.

The ATS comprises *test cases* and *fault cases* which are implementation-independent and focused on the detailed content of the specification. Scenarios are identified for one test purpose, taking into account the service capability, e.g. the service requests and reports defined in ECSS-E-7041A. There must be at least one test case created for each situation in which the requests may be received (by situations we mean, the different values carried into packet fields, counters, timers and replays that may eventually be received). Fault cases are similar to test cases, except that they represent all the exceptional situations and fault handling. The *external faults* defined within the problem domain such as memory and communication faults help in deriving the fault cases. Assumptions about the service testability (i.e., the points of control and observation) are required for the definition of the tests. Only observable inputs and outputs are mapped for the conformance requirements. In order to automate some steps in the generation of test and fault cases we propose a new approach, using a formal step-by-step method, which is presented in Chapter 4.

Besides the points of control and observation, facilities for executing the tests and injecting the faults influence the choice of cases in ATS. The testing architecture strongly affects the conformance requirements that can effectively be checked by the testing means (Cavali, 1996).

The *ferry-injection architecture,* illustrated in FIGURE 3.2, was adopted as the mean for executing the tests and injecting the faults (Araujo, 2000). In this architecture the testing system and the system under testing are as independent as possible of each other. This independence is obtained by the use of the ferry-clip (composed of the components: active ferry, passive ferry and ferry-channel) concept defined by Chanson (Chanson, 1989), (Zeng, 1985). The test engine component comprises an user interaction interface and a mechanism to run an executable test script containing the PETS. The test script controls the command flows through the PCOs. In this

architecture, memory and communication faults may be controlled by the Fault Injection Controller (FIC) and executed by the Fault Injection Module (FIM). At this moment, only communication faults involving loss, duplication, delay or alteration of messages were considered. Research has been carried out for extending the fault-injection architecture with memory faults facilities. Experiments in configuring this architecture for testing space applications are discussed in (Martins, 2002) (Martins, 2003b).

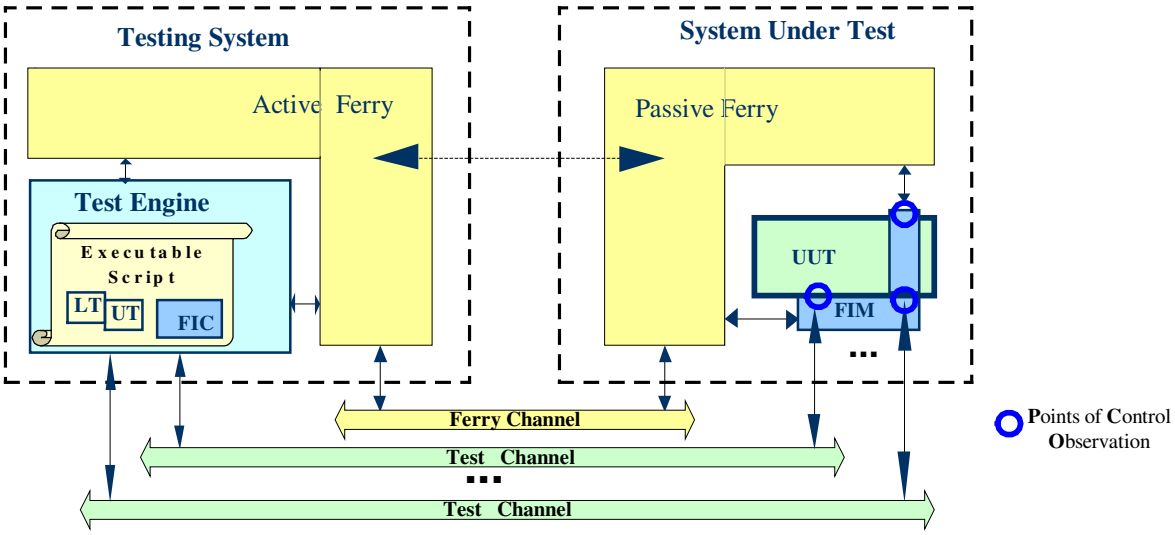

FIGURE 3.2 - Ferry-Injection Architecture.

SOURCE: Martins, (2003a).

Once the test/fault cases are defined, they may be translated into either TTCN[5] *test notation* or into a language supported by an automated testing tool. Each test/fault case is associated to an *expected output*. This information is later used in the *analyze test result* activity in the last phase.

3.4.3 Select and Parameterize the Test Cases

This activity starts the *test operation* phase. It is characterized by the choice of the test/fault cases, from the ATS, to be applied to the UUT. This choice takes into account

not only the testing architecture, but also the observability of the UUT. The selected cases are completed and are parameterized with the provided implementation details and information like values of counters and timers documented in ICS and IXIT. In this stage the test set is named *parameterized executable test suite* (PETS) and the tests are written in an executable language. In CoFI, the PETS may be given in a script-like language, which may be one used in space systems tests, as PLUTO (ECSS, 2001), or in any script-like language.

3.4.4 Execute the Test Campaigns

In this activity the PETSs are ready and the test campaigns are carried out on the UUT. The reactions of the UUT are observed for normal and exceptional situations. All inputs, outputs and other test events (i.e., the read-outs produced during the test campaigns) are logged not only for the result analysis but also for future reference. The observable output is associated to each executing test/fault case.

3.4.5 Analyze Test Results

This activity consists of comparing the *observable output,* generated during the test campaigns, with the *expected outputs* provided by the *Specify ATS* activity. The comparison may be manually or automatically-executed, once the outputs (*expected and observed)* are written in a format to generate the formal test report.

3.4.6 Generate the Test Report

This activity consists of attributing to each executed test/fault case, one of the following results: pass, fail, inconclusive, error in the abstract or executable test case, or abnormal test case termination (classification defined in IS-9646). The complete result analysis leads to a verdict about both the conformance requirements and the expected behavior when the implementation is submitted to external faults. Once the comparison is

[5] In earlier versions, TTCN stands for Tree and Tabular Combined Notation. The new meaning is Testing and Test Control Notation apt for TTCN-3.

performed, *statistics* about the test execution may be included in the *test report* for diagnostics.

Roughly, the CoFI differs from the IS-9646 testing process in the following aspects: it needs to define faults; it includes fault cases besides the test cases in the ATS, it provides an explicit comparison between the Expected and Observed outputs (automatic comparison is possible if the specification is given in a formal notation like a Finite State Machine (FSM)), it generates statistics about the test results, it includes the ferry-injection architecture, it generates test cases from a formal specification need of faults definition.

CHAPTER 4

APPROACH FOR STATE-BASED TEST AND FAULT CASES GENERATION

The approach to derive the abstract test and fault cases leads the tester to translate the service specification in a formal notation, a Finite State Machine.

On the one hand, the formal notation-based specification allows the use of formal algorithms to automatically generate test cases, thereby reducing much time and effort with tests and assuring a greater coverage of the specification. On the other hand, however, getting the specification into a notation that is acceptable for the formal algorithms is hard and time-taking work, requiring many iterations (Merri, 1996).

In order to reduce the difficulty of creating a formal and complete specification, the CoFI approach comprises a set of steps that make use of UML notation and the definition of scenarios. Scenarios of normal situations generate the test cases and the exceptional situations generate the fault cases. Thus, the number of cases generated each time is smaller than if global modeling were used. In short, the main steps of the CoFI approach are:

1) describe the service as an use case:
    - identify the requests and reports handled by the service,
    - identify the information represented by special variables or by the telecommand packet data fields whose variation of values causes a service behavior to change the service behavior. Represent this information as *operational variables* (Binder, 2000- chapter 7),
    - define a *basic scenario,* whose actors are the service provider and the service client;
2) create *normal* and *exceptional scenarios* starting from the basic scenario, taking into account situations deduced from the operational variables;

3) design Sequence Diagrams on mapping all requests, reports and operational variables exchanged between the server and client providers. Separate the normal and the exceptional handling situations and create:

   - one *normal sequence diagram* that represents the normal scenarios and ,
   - one *exceptional sequence diagram* that represents the exceptional scenarios;

4) create an *event/state matrix* where:

   - a line represents an event. The events may be valid, invalid and inopportune and they are represented by requests and by operational variables,
   - a column represents a state. States are derived from the *sequence diagrams* created in previous step. Take each interval between events in the server provider line of the sequence diagram as a state,
   - a cell has two elements: the output (or action) executed as a consequence of the event occurrence and the target state. The cell represents the same information that the transition in the state diagram. A transition is formally written as [output/target_state],
   - an output may be: (i) a normal, explicitly-specified output; (ii) an exceptional, explicitly-specified output; (iii) an abnormal termination, or simply rejection of the input (this is also named “illegal transition”); (iv) a non-specified output (in this case the cell shall be completed with implementation information). In observing the testing process these information are in the ICS and IXIT documents (see Chapter 3);

5) create a *normal finite state diagram* only with the events triggering the normal, explicitly-specified outputs. This step may be done in parallel with the step 4. In fact, the sate/event matrix is not required to be complete before the step 5 being performed;

6) generate the *test cases* submitting the *normal finite state diagram* to recheability analysis algorithms for graph search. If a depth first search with intercalation algorithm is used to generate test cases and this set of test cases is fully applied, the

tests completely cover the specification. In (Ural, 1992) the reader finds a description of six formal methods for generating test sequences from FSM-based specifications. The Condado tool (Martins, 1999) for generating the test comprises the above cited characteristics;

7) create an *exceptional finite state diagram* only mapping the events that trigger the exceptional explicitly-specified output and the illegal transitions (see step 4). This step may be done in parallel with the step 4. The step 4 helps to identify the exceptional information;

8) Generate the *fault cases*:

   - Submit the *exceptional finite state diagram* to algorithms of graph traversing (Lee, 1996) for automatically generate fault cases. A fault case is identified by both a set of transitions necessary to the UUT reach the state in which the fault has to be injected and the fault parameters. The fault must be such that is supported for the FIC/FIM components of the testing architecture. The communication faults are characterized by the following parameters:

     - *when* - time or the message identification,

     - *what* – fault type. For communication faultload, the fault types represent the most common transmission errors like extra message (data insertion or duplication), message loss (deletion), message fields modified (distortion and reordering) and delayed (Holzmann, 1990),

     - *how* - mask or byte position. The values to be attributed to *how* parameter are obtained from: (i) the operational variables, (ii) the requests (iii) other information available on the *event/state matrix* (see step 4),

     - *where* - the communication fault which is associated to a PCO.

It is worth clarifying that this approach presupposes the following definitions:

- an implementation-independent test or fault case is characterized as a sequence of inputs and corresponding expected outputs;

- a pair [input, expected output] or [fault, expected output] is represented in a transition of a finite state machine. So, a test and fault case is a path starting from the initial state and finishing in a reachable state.

- The fault to be injected is characterized by both the instant to inject (or the UUT state) and by the fault instance. The instant to inject the fault is relatively defined by the set of inputs from the initial state and subsequent inputs up to a specific (reachable) state.

The following section illustrates the CoFI approach applied for the *telecommand verification* service of the ECSS-E-70-41 for test and fault cases generation.

CHAPTER 5

ABSTRACT TEST AND FAULT CASES FOR THE ECSS-E-70-41 SERVICES

This section presents an example of an abstract test case and an abstract fault case derived according to the approach defined in chapter 4, for the *telecommand verification* service standardized in (ECSS, 2003).

The service description:

The *telecommand verification* service provides feedback about the execution of a given telecommand at whichever stages are meaningful for that telecommand. Long-term execution telecommands usually pass through the following stages: reception, execution start, execution progress and execution completion. In deep space missions these stages may take hours. So, when requiring this service, the client should indicate which reports he or she wants, by setting in the Ack four-bit (AckAccBit, AckStartBit, AckProgBit, AckCompBit) of the packet data field[6], respectively for acceptance-report, start-report, progress-report and completion-report. Table 2 summarizes the service capabilities by listing the reports for success and for failures.

The capabilities, the service type and subtypes, the report for success and for failures are summarized in TABLE 5.1. In column “Requests and Reports” it is presented minimum and additional capabilities.

[6] For more information about the telemetry and telecommand packet formats, the reader is referred to the chapter 5 of the ECSS-E-70-41A

TABLE 5.1 – Capability Set of the *telecommand verification* service.

Service Type	Service description	Capability set	
		Sub-type	Requests and Reports
1	Telecommand Verification	1 2 3 4 5 6 7 8	TC acceptance Report – Success TC acceptance Report – Failure TC execution started Report – Success TC execution started Report – Failure TC execution progress Report – Succes TC execution progress Report – Failure TC execution completed Report – Succes TC execution completed Report – Failure

FONTE: ECSS-E-70-41, (2003).

The test purpose for this service is *to check the telecommand verification behavior*. The complete set of test purposes for the ECSS-E-7041A is presented in appendix B.

The CoFI approach for a test/fault case derivation:

In step 1 (see Chapter 4), the requests and reports are directly identified in the capability set (shown in Table 5.1); the operational variables are identified in the Ack bits whose four-bit combination reflects the ground-on-board communication operational scenarios.

The next steps require assumptions about the Points of Control and Observation (PCOs), because only the observable events are taken into account. We are supposing three PCOs for this service:

- PCO_1 - observes the telecommand application data arrival,
- PCO_2 - allows us to recognize the status of the telecommand execution (although the telecommand may be performed by one or more Application Processes, this information should be passed through only one communication mechanism represented here by PCO_2) and
- PCO_3 - allows us to observe the generated telemetry reports to be sent to the service client.

FIGURE 5.1, below, illustrates the three PCOs for the *telecommand verification* service.

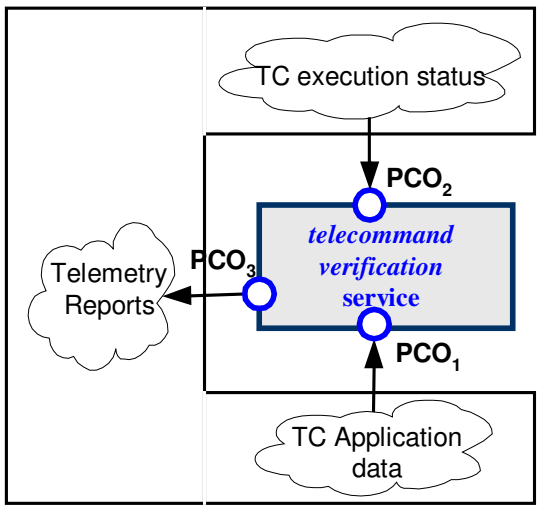

FIGURE 5.1 - Points of Control and Observation for the *telecommand verification* service.

Steps 2 and 3 are not shown here, whereas the complete event/state matrix established in step is given in appendix C.

5.1 Test Case:

The steps 5 and 6 are dedicated to the normal situations from which the test cases are derived. The four-bit Ack, identified in the first step of the approach as operational variables become *guard conditions* in the fourth step. The events observed in PCOs and the guards define partially a transition in the *normal finite state diagram*. The transition definition is, then, added with the expected output. The transitions of the normal situations compose both the event/state matrix and the *normal finite state diagram*.

A transitions is characterized by *Input[guard]/Output*.

For a *normal finite state diagram*: Input, guard and Output (for all transitions) are finite sets:

- Input = set of observable (normal) events,

- Guard = set of conditions: logical expressions, including Boolean variables,
- Output = set of observable actions or outputs;

The Input, guard and Output sets for the *telecommand verification* service are:

Input = {TCarriv, AccOK, StartOK, PrOK, CompOK};

Guard(**) = { Nstep < Max (*) , AckAccBit, AckStartBit, AckProgBit, AckCompBit};

Output = {receive TC, AccSucc_Rep, StartSucc_Rep, PrSucc_Rep, CompSucc_Rep}

(*) The guard condition [Nstep < Max] was added to this set to indicate whether the current execution-step (Nstep) has achieved its maximum allowed value (Max).

(**) All elements of the Guard set are associated to all transitions. The possible values and the meaning of these elements are shown in Table 5.2.

TABLE 5.2 – Guard conditions for the *telecommand verification* service.

Guard	**Possible values**	**Description**
Nstep < Max	v / f / -	Execution-step is not the last / last execution-step / don't care
AckAccBit	1 / 0 / -	Acceptance report required / not required / don't care
AckStartBit	1 / 0 / -	Start_execution report required / not required / don't care
AckProgBit	1 / 0 / -	Progress_execution report required / not required / don't care
AckCompBit	1 / 0 / -	Completion_execution report required / not required / don't care

The *normal finite state diagram*, representing the normal operational scenarios of service, is illustrated in FIGURE 5.2.

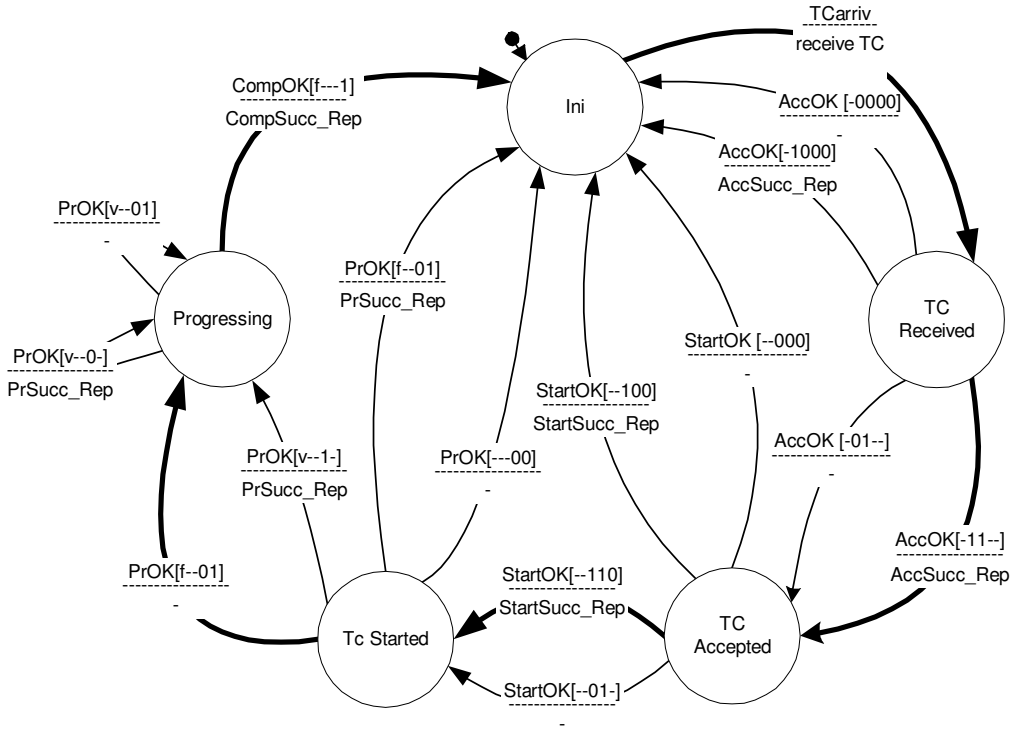

FIGURE 5.2 - *Normal finite state diagram* mapping the normal situations of the *telecommand verification* service.

A path P, composed of transitions from the *normal finite state diagram* is a test case. Since the *normal finite state diagram* maps the events triggered in PCOs, the test cases cover possible event combination occurred in the service implementation (if the implementation conforms the specification).

A possible normal scenario reflects the Ack four-bit containing 1101, in which the following reports should be generated to the Client: TC_acceptance_Success TC_execution_started_Success and TC_progress_execution_ Success. This scenario is

mapped in the path *p,* given in the set *p*, which comprises the transitions draw in bold in FIGURE 5.2.

p = {TCarriv / receive_TC , AcOk[-11--] / AccSucc_Rep , StartOk[--110] / StartSucc_Rep , PrOk[f--01] / - , CompOk[f---1] / CompSucc_Rep}

The Input and Output sets of the path *p* are given in the sets $Input_p$ and $Output_p$ respectively.

$Input_p$ = {TCarriv , AccOk[-11--], StartOk[--110], PrOk[f--01], CompOk[f---1] }

$Output_p$ = {receive TC, AccSucc_Rep, StartSucc_Rep, CompSucc_Rep}.

This path *p* represents a test case, where $Input_p$ contains the sequence of inputs to ve submitted to the service implementation and $Output_p$ contains the expected outputs generated by the UUT whenever the sequence given in $Input_p$ occurs in that order. The test case, illustrated in TABLE 5.2, was translated into the TTCN. Here, the inputs and outputs are associated to the respective PCOs. The TTCN was chosen for illustration purposes as it is an ISO standard test notation.

TABLE 5.3 – Dynamic Behavior Description Test Case.

Test Case Dynamic Behavior				
Test Case Name : telecommand acceptance, start and completion execution report with success **Purpose** : To verify the telecommand execution.				
N	**Behavior Description**	**Constraint**	**Comments**	**Expected Output**
1	PCO_1 ? TCarriv		A TC application data arrives requiring reports for acceptance, starting and completion execution steps	-
2	PCO_2 ? AccOk PCO_3 ! AccSuc_Rep	[Ackbits = 1---]	AccOk event trigged at the end of fields verification	Acceptance Success Report
3	PCO_2 ? StartOk PCO_3 ! StartSuc_Rep	[Ackbits = -1--]	StartOk event trigged at the end of the first execution step	Start-execution Success Report
4	PCO_2 ? PrOk	[Nstep =1 = Max] [Ackbits = --01]	ProgOk event trigged at the end of the second execution step. Only one progressing step exist, i.e, the TC requires 3 steps (start, 1 progress and 1 completion)	-
5	PCO_2 ? CompOk PCO_3 ! CompSuc_Rep	[Nstep =1 = Max] [Ackbits = ---1]	CompOk event trigged at the end of completion execution	Completion-execution Success Report
Detailed Comments: The "?" indicates a received event, the "!" indicates a send operation. PCO_1: is the interface with the Packetization Protocol layer in the satellite System. PCO_2: is the interface with the TC executing application at the application level in satellite System PCO_3: is the interface with the ground system TCarriv: is an event that carry out telecommand packet data for TC verification purposes.				

In the step 6 the normal scenarios represented in the state diagram of FIGURE 5.2 were submitted to the Condado tool. A set of 38 test cases were generated. These test cases are presented in appendix D.

5.2 Fault Case

The *telecommand verification* service specifies that reports indicating failure are always transmitted, so failure reports are not required in packet data fields as they are for the success reports.

In step 7 (and step 4) all the failure reports were associated to exceptional situations, mapped into the exceptional scenarios, and represented in the *exceptional finite state diagram,* shown in FIGURE 5.3.

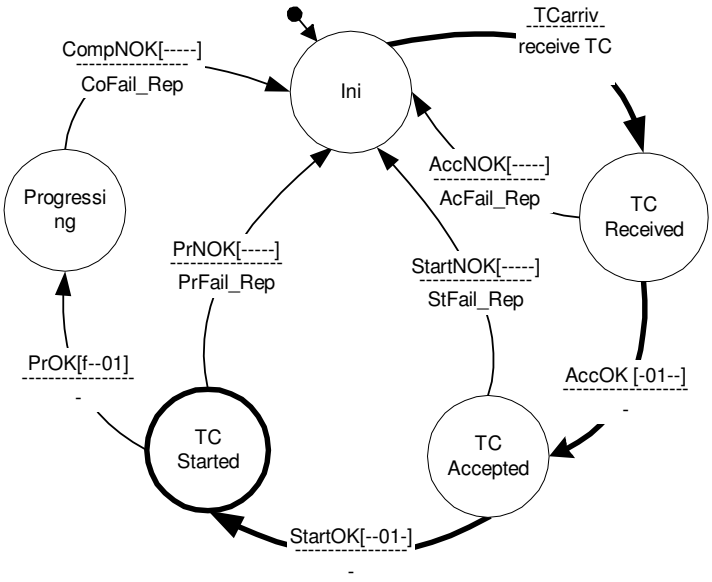

FIGURE 5.3 - *Exceptional finite state diagram* mapping the Exceptional Situations of the *telecommand verification* service.

In step 8, the suite of fault cases are derived from the exceptional finite state machine. The fault case suite includes only simple paths instead of requiring interleaving of events as in the normal case. This suite may be derived from the transition tree, shown in FIGURE 5.4, in which a fault case is a sequence of transitions that start from INI state and reach the INI again, but traversing all other states once. (It is supposed that the occurrence of a failure cause the system came back to its initial state.)

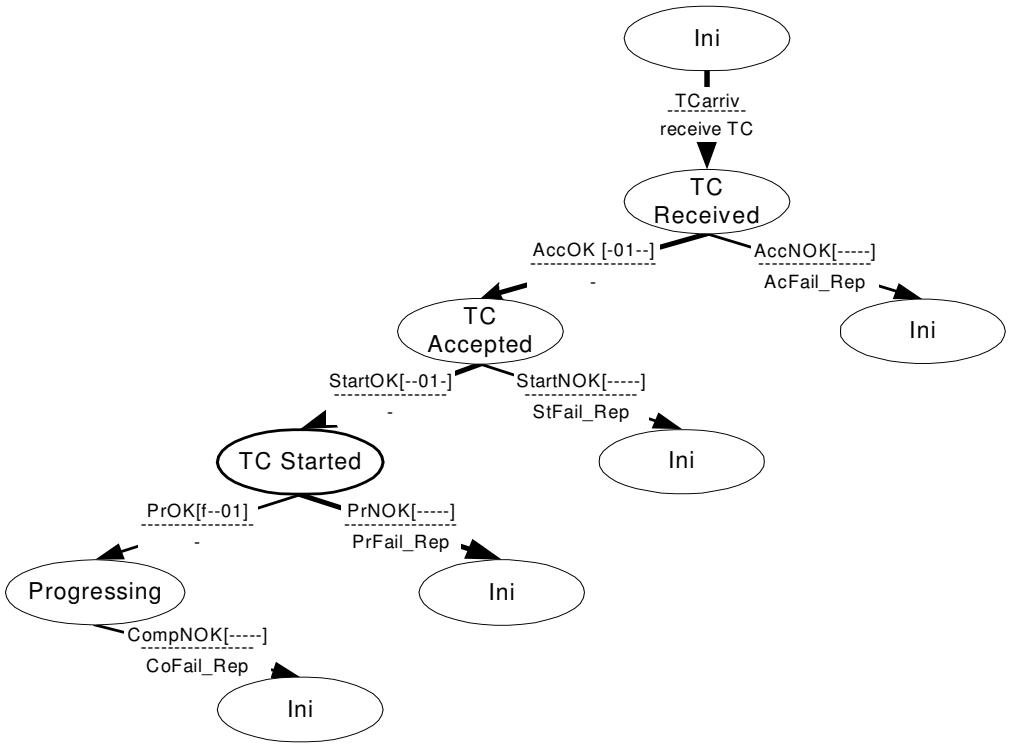

FIGURE 5.4 - Transition Tree of the mapping the Exceptional Situations of the *telecommand verification* service.

The F_{cases} set contains the four fault cases derived from the transition tree. Each element of F_{cases} is a sequence of pairs (input[guard]/output) and (fault[guard]/output).

F_{cases} = { (TCarriv/ receive TC, AccNOK[-----]/ AcFail_Rep) ;
(TCarriv/ receive TC, AccOK[-01--]/ - , StNOK[-----]/ StFail_Rep) ;
(TCarriv/ receive TC, AccOK[-01--]/ - , StOK[--01-]/ - , PrNOK[-----]/ PrFail_Rep) ;
(TCarriv/ receive TC, AccOK[-01--]/ - , StOK[--01-]/ - , PrOK[f--01]/ - , CompNOK/ CompFail_Rep) }

The F_{cases} set contains only the first part characterizing the fault cases, the relative instant to inject the fault. The fault instance (see Section 4), depends on the parameters of the exceptional scenarios.

A possible exceptional scenario reflects the Ack four-bit containing 0100 and a failure in the telecommand execution progress stage. This scenario triggers the generation of the following reports to the client: TC_execution_started_Success and TC_progress_execution_Failure. The fault case contains:

- the fault relative instant is given by the path p_f illustrated in P_{pf} set; the inputs are in I_{pf} set, and expected outputs are in O_{pf} set.

  P_{pf} = {(TCarriv/ receive TC, AccOK[-01--]/ - , StOK[--01-]/ - , PrNOK[----]/ PrFail_Rep)}

  I_{pf} = { TCarriv, AccOk[-10--], StOk[--01-] };

  O_{pf} = {receiveTC, PrFail_Rep}.

  The inputs are associated to the PCOs as follows: TCarriv in PCO_1 and AccOK, StartOK in PCO_2

- the parameters of the fault instance has the following values :

  - *when* = following the last normal event given in I_{pf};
  - *what* (fault type) = extra message;
  - *how* = data insertion of the PrNOK event;
  - *where* = in PCO_2.

This abstract fault case, written in TTCN, is shown in TABLE 5.3.

TABLE 5.4 –Dynamic Behavior Description Fault Case.

Test Case Dynamic Behavior				
Test Case Name : start execution report with success and failure in the progress **Purpose** : To verify the telecommand execution.				
N	**Behavior Description**	**Constraint**	**Comments**	**Expected Output**
1	PCO_1 ? TCarriv		A TC packet data arrives, requiring reports for starting execution step	Receive TC
2	PCO_2 ? AccOk	[Ackbits = 01--]	AccOk event trigged at the end of fields verification	-
3	PCO_2 ? StartOk PCO_3 ! StartSuc_Rep	[Ackbits = -1--]	StartOk event trigged at the end of the first execution step	Start-execution Success Report
4	PCO_2 ? PrNOk PCO_3 ! PrFail_Rep		ProgNOk event trigged at the end of the second execution step.	Progress Failure Report

CHAPTER 6

RELATED WORK

In literature one may find a great number of studies not only on formal models of the system behavior but also on test cases generation from state-based specifications mainly for protocol testing. Good overview of test case generation may be found in (Dssouli, 1999), (Lee, 1996), (Bochmann, 1994), etc.. Tretman has established a formal definition of the conformance relation between the implementation and the specification (Tretmans, 1999). An updated survey of communication protocol testing focused on test sequence generation methods, testing tools and experience reports is presented in (Lai, 2002).

The CoFI process may be compared with the SAMSTAG (Grabowski, 1994), N+ (Binder, 2000), TOP (Koné, 2004), as they all are based on formal definitions of the specification, they orients the test cases generation from a state-based specification.

The testing method named 'SDL and MSC based test case generation' (SAMSTAG) aims at developing automatic generation of abstract test cases for telecommunication protocols. It formally defines both the specifications and the test purposes. The method assumes that the specification is written in Specification Description Language (SDL) and the test purposes are defined by a Message Sequence Chart (MSC). Besides formalizing the test purposes, the method defines the relation between test purposes, protocol specifications, and the test cases.

The TOP method handles interoperability test automation for communication systems. The test case generation uses an on-the-fly basis, i.e., the global behavior graph of the system under test is not previously computed; instead, test paths are extracted from an analysis of sub-specifications of the communicating entities. The tests are deployed over the CORBA platform.

In (Binder,2000; chapter 7) the N+ state-based testing strategy, focusing on UML state models developed for object-oriented implementations, is presented. In this strategy, the

test cases aim at covering round-trip and sneak paths of a state-based specification. The sneak paths test illegal or non explicitly-defined transitions, while the round-trip covers the explicitly-modeled transitions. A response matrix is created with the complete information about the implementation under test and the sneak paths are developed from the response matrix. Although the N+ is neither for protocol testing nor based on the IS-9646, the idea of distinguishing the normal from exceptional situations when dealing with test case generation was incorporated into the CoFI process.

CHAPTER 7

CONCLUSION

This report presented a testing process named CoFI (Conformance and-Fault-Injection) which is based on the conformance testing process standardized in IS-9646 for protocol certification. The CoFI testing activities comply with software-implemented fault injection (SWIFI) and test automation were added over this basis.

The CoFI process roughly instance the IS-9646 methodology and framework for space software services testing.

The CoFI process includes an approach for generating test and fault cases with which a reusable abstract test suite is created.

The *telecommand verification* service, defined in the ECSS-E-70-41[A] standard and prepared by the European Space Agency (ESA), was used to illustrate the generation of an abstract test case and an abstract fault case.

The CoFI pproach leads to improvement not only in the testing activities efficiency, but also in the reliability of space application implementing the PUS services. Additionally, it offers a real opportunity to reduce the testing costs in a mission, since it guides the design of the test cases that are still frequently performed by testers by interpreting specifications written in natural language.

The CoFI process requires yet further researches. Although the test cases were presented here in TTCN, the notation recommended by ISO, the CoFI process does not determine a test and fault case description language. Studies must be carried out on the use of the language like PLUTO to write the Parameterized Executable Test Suite (PETS). Other points that have being studied are related to the fault formal characterization and the ferry-injection testing architectures supporting multi-point communication.

Acknowledgment

We would like to thank researchers from the European Space Agency that motivated us to continue research on certification-related space software procedures. We are grateful to our colleague M. Fatima Mattiello-Francisco for encouraging this work.

REFERENCES

Ambrosio, A.M.; Martins E.; Mattiello-Francisco M.F.; Silva C. S. ; Vijaykumar N. L. On the use of test standardization in communication space applications. In: International Conference on Space Operations, SpaceOPs 8., 2004. 17-21 May, Montreal, Canadá. **Proceedings**... Montreal:AIAA, 2004. Available in the digital library URLib: <www.spaceops2004.org>. Access in: 2 sep. 2004.

Araújo, M.R.R. **Fsofist -** uma ferramenta para teste de protocolos tolerantes a falhas. 2000. 52p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Computação Universidade de Campinas, Campinas. 2000.

Arlat, J.; Aguera, M.; Amat, L.; Crouzrt,Y.; Fabre, J.-C.; Lapri, J.-C.; Martins, E.; Powell,D Fault Injection for Dependability Validation: A Methodology and Some Applications. **IEEE Transactions on Software Engineering**, v. 16, n.2, p. 166-182, 1990. (ISS 0098-5589).

Instituto Nacional de Pesquisas Espaciais (INPE)/ Universidade Estadual de Campinas (UNICAMP). **ATIFS project**. Available in the digital library URLib: In: <http://www.inpe.br/atifs>. Access in: 5 sep. 2002.

Baumgarten, B.; Giessler, A. **OSI conformance testing methodology and TTCN**. Amsterdan:Elsevier, 1994. 328p.

Binder, R. **Testing object-oriented systems** - models, patterns and tools**.** Boston: Addison-Wesley, 2000. 1191p.(ISBN 0-201-80938-9).

Bochmann, G.; Petrenko, A . Protocol Testing: review of Methods and Relevance for Software Testing. In: International Symposium on Software Testing and Analysis, 1994. 17-19 August 1994, Seatle, Washington, USA. **Proceedings**... Seatle: ACM Press. p.109-124

Chanson, S.T.; Lee, B.P.; Parakh, N.J.; Zeng H.X. Design and implementation of a ferry clip test system. In: Symposium on Protocol Specification Testing & Verification, IFIP 9.,1989, Enscchede, The Netherlands. **Proceedings**... The Netherlands: North-Holland, p.101-118, 1990.

Cavali, AR.; Favreau, J.P., Phalippou, M. Standardization of formal methods in conformance testing of communication protocols. **Computer Networks and ISDN Systems,** n. 29, p.3-14, 1996.

Consultative Committee for Space Data Systems (CCSDS). **CCSDS 910.0-Y-1** - Space link extension services – executive summary. Yellow Book. Issue 1. April, 2002. Available in the digital library URLib: <http://www.ccsds.org/ccsds/recommandreports.html>. Access in: 2 jun. 2004.

Consultative Committee for Space Data Systems (CCSDS). **CCSDS-201.0-B-3 -** Telecommand part 1—channel service. Blue Book. Issue 3. June 2000. Available in the digital library URLib: <http://www.ccsds.org/ccsds/recommandreports.html>. Access in: 2 jun. 2004.

------. **CCSDS-202.0-B-3 -**Telecommand part 2 - data routing service. Blue Book. Washington DC: NASA. . Issue 3. June 2001a.

------. **CCSDS-203.0-B-3 -**Telecommand part 3 - data management service. Blue Book. Washington DC: NASA. Issue 2. June 2001b.

Dssouli, H.; Salek, K.; Aboulhamid, E; En-Nouaary, A ; Bourhfir, C. Test Development for Communication Protocols: Towards Automation. **Computer Networks**, n. 31, p. 1835-1872, 1999.

European Cooperation for Space Standardization (ECSS). **Standards**. Noordwijk: ESA publication Division. Available in the digital library URLib: <http://www.ecss.nl/>. Access in: 2 jun. 2003.

------. **ECSS-E-70-32A** - Space engineering Procedure Language for User in Test and Operations: PLUTO. Issue Draft 5, 2001. Noordwijk: ESA publication Division. Available in the digital library URLib: <http://www.ecss.nl/>. Access in: 6 dec. 2003.

------. **ECSS-E-70-41** [A] – Ground systems and operations: telemetry and telecommand Packet utilization. January, 2003. Noordwijk: ESA publication Division. Available in the digital library URLib: <http://www.ecss.nl/>. Access in: 2 jun. 2003.

Grabowski, J. **SDL and MSC Based Test Case Generation:** An Overall View of the SAMSTAG Method. Berne: University of Berne, 1994. 23p. (IAM-94-0005).

Holzmann, G. J. **Design and validation of computer protocols.** Englewood Cliffs: Prentice Hall, 1990. 512p. (ISBN 0-13-539834-7). Available in the digital library URLib: <http://spinroot.com/spin/doc/book91.html>. Access in:02 sep. 2002.

International Organization for Standardization.(ISO). **IS 7498-1** - Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. Geneve, 1994.

International Organization for Standardization (ISO/IEC). **IS 9646** - International standard conformance testing methodology and framework. Geneve, 1991.

International Telecommunication Union - Telecommunication Standardization Sector of ITU (ITU-T). **Recomendation X.290** – OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications – general concepts. Geneve, April, 1995.

Koné, O. An Interoperability Testing Approach to Wireless Application Protocols. **Journal of Universal Computer Science**, v. 9, n.10, p. 1220-1243, 2003.

Lai, R. A survey of communication protocol testing. **The Journal of Systems and Software**, n. 62, p. 21-46, 2002.

Larson, W.; Werts, J. **Space mission analysis and design.** Dordrecht: Kluwer Academic Publisher, 1992. 865p.

Lee,D.; Yannakakis,M. Principles and methods of testing finite state machines: a survey. **Procceedings of IEEE**, v.84, n. 8, p. 1090-1123, 1996.

Madeira, H.; Some, R. R.; Moreira, F.; Costa, D.; Rennels, D. Experimental evaluation of a COTS system for space applications. In: International Conference on Dependable Systems and Networks (DSN'02), June 23 - 26, 2002, Washington, D.C., USA. **Proceedings**... Washington, D.C.: IEEE Computer Society, 2003. P.325-330. Available in the digital library URLib: <http://computer.org/proceedings/dsn/1597/15970325abs.htm>.

Martins, E. ; Sabião, S.B.; Ambrosio, A. M. ConData: a Tool for Automating Specification-based Test Case Generation for Communication Systems. **Software Quality Journal**, v.8, n. 4, p. 303-319, 1999.

Martins, E.; Mattiello-Francisco M.F.; Morais, A. N.P. Uso da ferramenta de testes FSOFIST na validação de uma aplicação espacial. In: Jornada Ibero-Americana de Engenharia de Software e Engenharia de Conhecimento, 2., 2002. Salvador, BA, Brasil. **Anais**... 2002. 1 CD-ROM..

Martins, E.; Ambrosio, A.M; Mattiello-Francisco M.F. ATIFS: a testing toolset with software fault injection. In: Workshop UK Testing Research, 2. (SoftTest), 4-5 September 2003,, York, England. **Proceedings**... York: Department of Computer Science/University of York, 2003a.

Martins, E.; Mattiello-Francisco, M.F. A Tool for Fault Injection and Conformance Testing of Distributed Systems. In: Latin-American Dependable Computing Symposim, 1., São Paulo, Brasil, october 2003. **Lecture Notes in Computer Science**. Berlin: Springer Verlag, p. 282-302, 2003b.

Mazza, C. **Standards:** the foundations for space IT. Darmstadt, Germany, 2000. Lecture given during the Workshop on Space Information Technology in the 21st Century, 27 sep. 2000. Available in the digital library URLib: <www.esoc.esa.de/pr/documents/workshops/it_2000/it_in_future/esa_c_mazza.ppt >. Access in:02 sep. 2002.

Merri, M ; Rüting, J.; Schurman, P. Validation of the ESA Packet Utilization Standard by object-Oriented Analysis. In: International Conference on Space Operations, 4. (SpaceOPs 96), 16-20 September 1996, Munich, Germany. **Proceedings**.... Gilching: CAM GmbH. 1 CD-ROM.

Merri, M.; Melton, B.; Valera, S.; Parkes, A. The ECSS Packet Utilization Standard And Its Support Tool. In: International Conference on Space Operations, 7. (SpaceOPs02), 9-12 October 2002, Houston, USA. **Proceedings**... Houston: AIAA, 2002. (ISBN 1-56347-585-5). Available in the digital library URLib: <www.spaceops2002.org/papers/spaceops02-p-t5-06.pdf >Access in:02 fev. 2004.

Mertens, M. Advanced Protocol Testing Methods and Tools. In: International Conference on Space Operations, 7. (SpaceOPs02), 9-12 October 2002, Houston, USA. **Proceedings**... Houston: AIAA, 2002. (ISBN 1-56347-585-5). Available in the digital library URLib: <www.spaceops2002.org/papers/spaceops02-a-t1-31.pdf>. Access in:02 fev. 2004.

Tretmans, J.; Belinfante, A. Automatic Testing with Formal Methods. In: Conference on Software Testing Analysis and Review, 7. (EuroSTAR´99), 8-12 November,1999, Barcelona, Spain. **Proceedings**... Galway: EuroStar Conferences, 1999.

Ural, H. Formal methods for test sequence generation. **Computer Communication**, v.15, n.5, p.311-325, june 1992.

Voas, J. M.; McGraw, G. **Software fault injection**: inoculating programs against erros. New York: John Wiley & Sons, INC. 1998. 353p. (ISBN 0-471-183810-4).

Zeng, H. X.; Rayner, D. The impact of the Ferry concept on protocol testing. In: International Conference on Protocol Specification, Testing and Verification, 5., June 10-13, 1985, Toulouse-Moissac, France. **Proceedings**... The Netherlands: North-Holland - IFIP WG6.1. 1985. p. 519–531. (ISBN:0-444-87881-5).

APPENDIX A
OVERVIEW OF THE IS-9646 - CONFORMANCE TESTING METHODOLOGY AND FRAMEWORK

A-1 -Protocol certification issues

The IS-9646 standard is oriented towards practical needs. It incorporates a great deal of practical experience from test experts which have been involved in concrete test issues.

In guiding the test of a standardized protocol implementation, ISO allows a wide acceptance of test results produced by different laboratories (testers). The definition of effective and well-accepted schemes of testing and certification avoids costly repetitions or variations of conformance tests of the same implementation.

Three entities are involved in the certification process: the Standardization Organization, the Client and the Testing Laboratory, as illustrated in figure A.1.

The Standardization Organization defines de product specifications and proformas (questionnaires) to be filled in by the Client. These questions aim at making clearer and unambiguous the conformance requirements. The Client, who implemented the product to be tested, fills the proformas (PICS and PIXIT). Finally, the Testing Laboratory conducts the conformance assessment of the Implementation Under Test (IUT) and generates the Protocol Conformance Test Report (PCTR).

The IS-9646 standard defines the testing laboratory and clients role and the documents to be generated for all test phases. Sub-section A.2 discusses the ISO conformance testing process, its phases and activities and concepts.

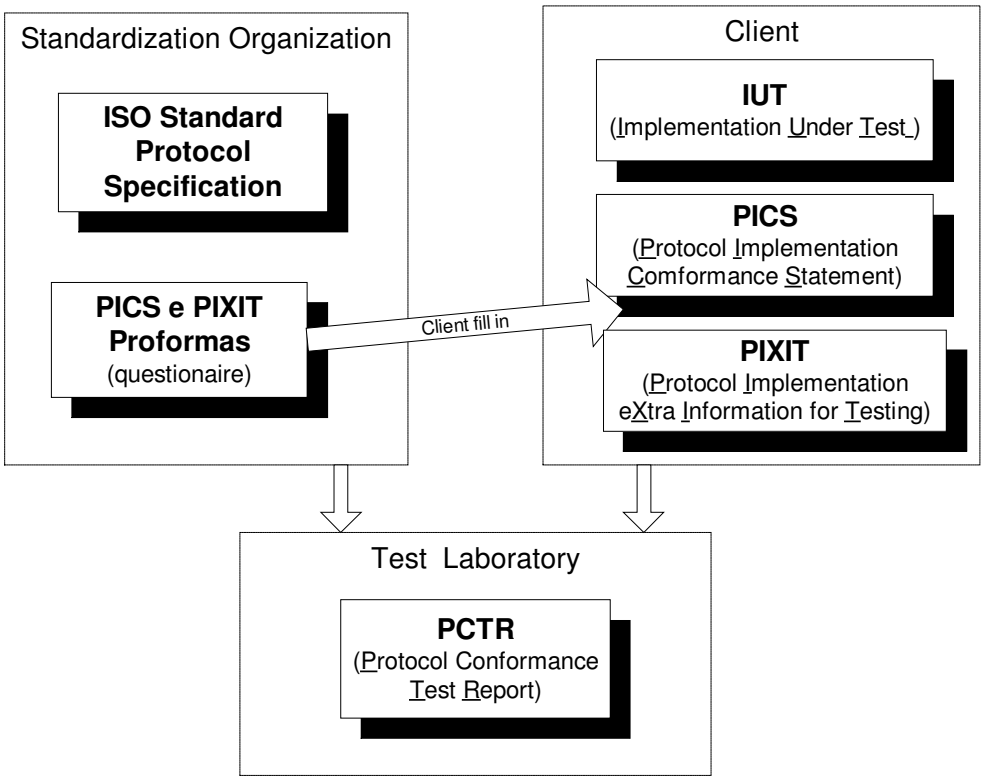

FIGURE A.1: Standardization Organization vs. Client vs. Testing laboratory

A-2 Conformance Testing Process:

The IS-9646 standard defines the conformance testing as testing the extent to which an implementation under test is a conforming implementation. According to this standard the activities for performing the conformance evaluation of an implementation against its specification is divided into three phases: Preparation for testing, Test operations and Test report production. The ISO conformance testing process, illustrated in Figure A.1, starts from the ISO standard protocol specification.

The protocol specification includes dynamic and static conformance requirements and both the Protocol Implementation Conformance Statement (PICS) and the Protocol Implementation extra Information for Testing (PIXIT) proformas[7]. The conformance requirements guide both the implementation (in right side of figure) and the testing (in

[7] A questionnaire to be fulfilled by the implementation responsible

left side of figure). The PICS proforma must be precisely filled in with details of the implementation under test (IUT) to be delivered to the tester.

The standard comprises many concepts and acronyms. Table 1 summarizes only the concepts used in this description. The testing process is described phase-by-phase in the following.

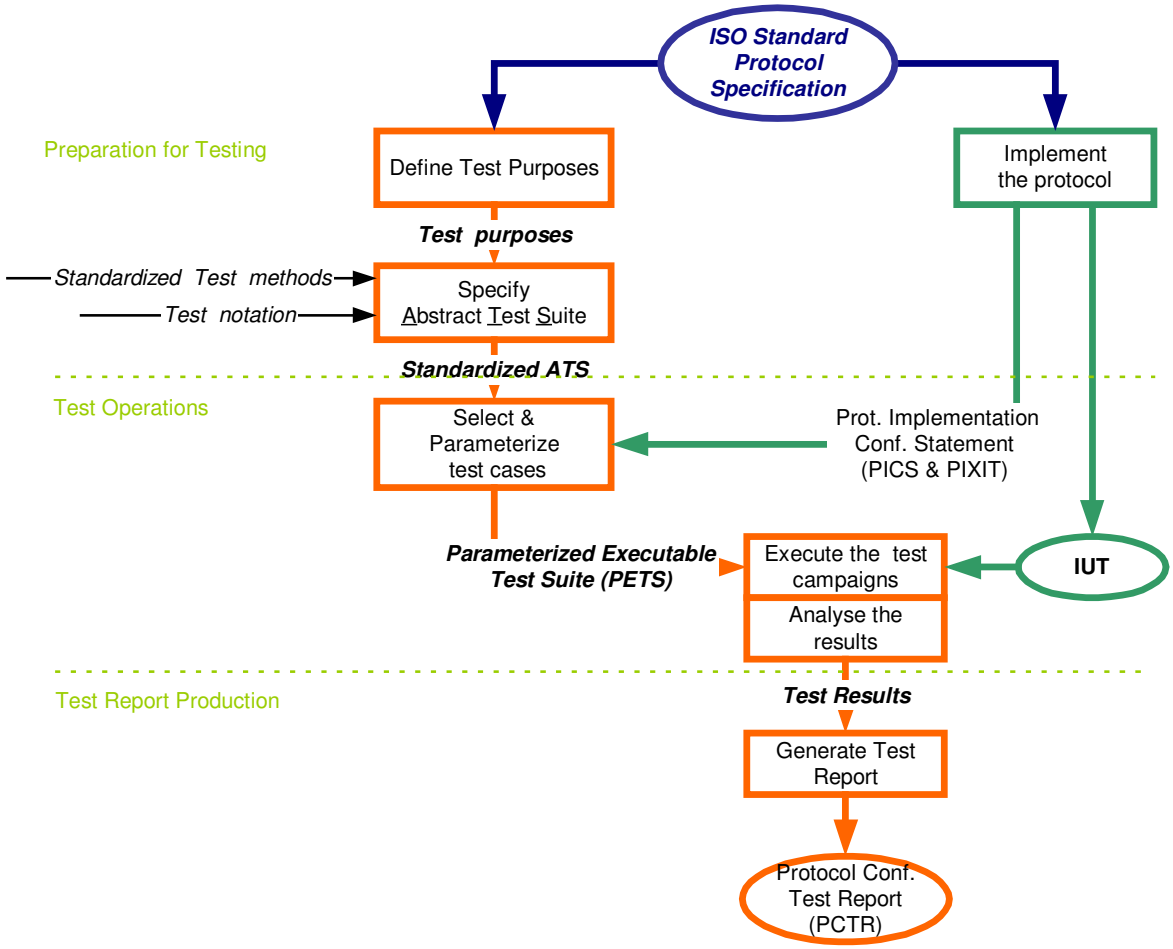

FIGURE A.2 – ISO conformance testing process

TABLE A.1 – ISO-conformance testing concepts

Test Purpose	A prose description of a well defined objective of testing, focusing on a single conformance requirement or a set of them.
Test Notation	The notation used to describe the abstract test case - implementation independent.
Abstract Test Suite (ATS)	A set of Abstract Test Cases . An ATS refers to a particular testing method.
Abstract Test Case	A complete and independent specification of the actions required to achieve a specific test purpose. It is complete in the sense that it is sufficient to enable a test verdict unambiguously to each potentially observable test outcome. It is independent in the sense that it should be possible to execute the derived executable test case in isolation from other such test cases.
Prot. Impl. Conformance Statement (PICS)	A statement made by the supplier of an implementation listing the capabilities have been implemented.
Protocol Implementation extra Information for Testing (PIXIT)	These information are about the SUT (e.g., addressing, realization of the Upper Tester within the IUT,etc.); they may precise some parameters or timer; help determine testable and untestable capabilities; and may be administrative information.
Parameterized Executable Tests Suite (PETS)	A selected set of test cases, in which all test cases have been parameterized in accordance with the relevant PICS.
Protocol Conformance Test Report (PCTR)	A document giving details of the testing carried out using a particular ATS. It lists all the abstract test cases and identifies those for which corresponding executable test cases were run, and the test verdicts.

A-2.1 Test Phases Description

The first phase, ***Preparation for testing***, starts with the definition of the *test purposes* based on the conformance requirement stated in the ISO Standard Protocol Specification. The specification is given in a textual notation and the test purposes are defined based on the experience of good testers. For the next step it is recommended to derive generic test cases, which should describe the necessary actions to achieve the test purpose without to consider the environment in which the actual testing will be performed. (ISO do not rule neither how to define the test purposes nor how to derive the test cases). Each generic test case is converted the abstract test case. In this step a particular testing method and the restrictions implied by the environment are taken into consideration. The complete set of test cases comprises the *Abstract Test Suite* (ATS). The test cases are described in a *test notation* defined by ISO, the Tree and Tabular Test Notation (TTCN), an implementation-independent language. Details of the implementation like timer values, communication port addresses are later added to the test cases definition. The ISO protocol specification may have different profiles. There

will be one ATS to each profile. The ATS is implementation-independent, but it is associated to one of the four *standardized test methods* specified in the standard.

A test method is a kind of architecture of the testing system. It defines a model for the accessibility of the implementation to the tester, in terms of the points of control and observation (PCO). Furthermore, if the IUT runs in the same testing system or is remotely accessible, the test method is local or distributed, respectively. The reader is referred to (Baumgarten, 1994) (ISO, 1991), (X.290, 1995), (Tretman, 1999), for more details about test methods.

Once the ATS is ready the ***Test operation*** phase may begin. In this phase, test cases are selected from the standardized ATS according to the protocol implementation conformance declaration (PICS) related to the implementation to be tested. The selected test cases are parameterized with the provided implementation information, i.e., values available in the PICS (and the PIXIT) like network addresses, counters and timers. The abstract test cases are then transformed into the *parameterized executable test suite* (PETS) taking into account the environment where the IUT is in. Once the PETS are complete the test campaigns are carried out and the results are analyzed.

The last phase, ***Test report production***, consists of the preparation of the Protocol Conformance Test Reports (PCTR). It may begin before the test operations are complete. During the test campaign the behavior of the IUT is observed and logged. For each test case one of the following results is possible:

- pass,
- fail,
- inconclusive,
- error in the abstract or executable test case or,
- abnormal test case termination.

All results are analyzed leading to a verdict about the conformance of the IUT with respect to the standard protocol specification, the conformance requirements.

Conformance testing does not include performance, robustness and reliability.

A-3 The Testing Methods overview

A *testing method* is a description of how the IUT is to be tested. It corresponds to points of control and observation available in SUT. A *testing method* describes an abstract testing architecture consisting of a configuration of functions (LT, UT, LTCF and TCP) as applicable to a context[8]. The relationships of these functions to the Testing System and the System under Test (SUT) is also described in a testing method. The IS-9646 defines four kinds of testing methods. Just for illustration, figure 3 shows the Local Testing Method in Single-Party context.

Each testing method determines the Points of Control and Observation (PCOs) and the test events (i.e., Abstract Services Primitives (ASPs) and Protocol Data Units (PDUs)) which will be contemplated in an ATS. The PCOs and the events are the key elements for the conformance test, because the IUT is a black-box component from the tester point of view.

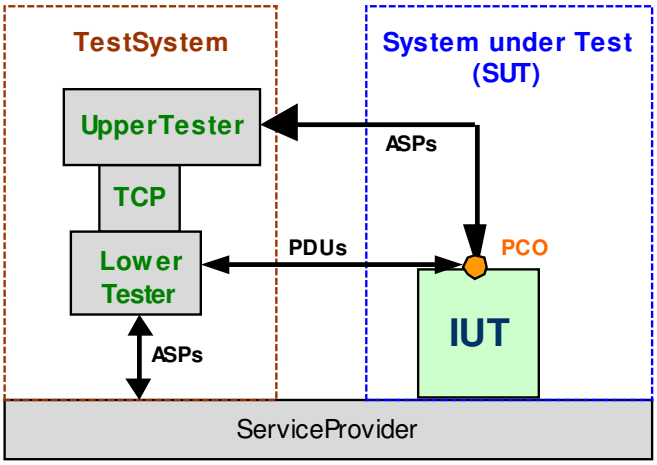

FIGURE A.3: Local test method in Single-Party context

[8] The context correspond to the number of communication channels required by IUT. The context may be: Single-party (one comm. channel); Multi-party (several comm. channels)

TABLE A.2: Concepts related with test methods

LT	Lower Tester
UT	Upper Tester
LTCF	Lower Tester Control Function
TCP	Test Coordination Procedure
PCO	Point of Control and Observation. A point within a testing environment where the occurrence of test events is to be controlled and observed. (Characterized by a set of ASPs and/or PDUs).
ASP	Abstract Service Primitive. An implementation-independent description of an interaction between a service-user and a service-provider.
PDU	Protocol Data Unit

APPENDIX B

TEST PUPOSES FOR THE STANDARD PUS SERVICES

In short, the *test purposes* to the services of this standard are the following:

(i) *to verify the telecommand execution*,

(ii) *to distribute device commands*,

(iii) *to report housekeeping & diagnostic data*,

(iv) *to report parameter statistics*,

(v) *to report events*,

(vi) *to manage memory*,

(vii) *to manage function*,

(viii) *to manage time*,

(ix) *to schedule on-board operations*,

(x) *to execute on-board monitoring*,

(xi) *to transfer large amount of data*,

(xii) *to control packet forwarding*,

(xiii) *to execute on-board storage and retrieval*,

(xiv) *to execute on-board pre-defined test*,

(xv) *to execute on-board operations procedure*,

(xvi) *to execute event-action*.

APPENDIX C

THE EVENT/STATE MATRIX FOR THE *TELECOMMAND VERIFICATION* SERVICE

This appendix presents the event/state matrix elaborated with information about the telecommand verification service. This matrix aims at helping the tester to provide the completeness assumption to the service specification for testing purposes. The completeness assumption means that each state of the UUT, can accept and respond to all possible inputs that trigger the system.

The matrix contains a sequential numeration of the transitions in the lines. To each transition, in a line, one may find the correspondent event (or input), the guard values and, in each cell of the states column, the expected action (or output) and the target state. (The target state is the reached state when the transition is fired).

Table C.1- Event/State Matrix for the *telecommand verification* service

Transitions	Events	Guards					States				
							Ini	**Tc Received**	**Tc Accepted**	**Tc Started**	**Progressing**
		Nstep < Max	**Ack Acc Bit**	**AckStartBit**	**Ack Prog Bit**	**AckCompBit**					
0	**TCarriv**	-	-	-	-	-	Tc Received/ Receive TC				
1	**AccOK**	-	off	off	off	off		Ini/ -	-		
2		-	on	off	off	off		Ini/ **AccSucc_Rep**			
3		-	off	on	-	-		Tc Accepted/ **-**			
4		**-**	on	on	-	-		Tc Accepted/ **AccSucc_Rep**			
5	**AccNOK**	**-**	-	-	-	-		Ini/ **AcFail_Rep**			
6	**StartOK**	-	-	off	off	off			Ini/ **-**		
7		-	-	on	off	off			Ini/ **StartSucc_Rep**		
8		-	-	on	on	off			Tc started/ **StartSucc_Rep**		

Table C.1- Event/State Matrix for the *telecommand verification* service (continuing).

9		-	-	off	on	-			Tc started/ **-**		
10	**StartNOK**	-	-	-	-	-			Ini/ **StFail_Rep**		
11	**ProgOK**	-	-	-	off	off				Ini/ **-**	
12		v	-	-	on	-				Progressing/ **ProgSucc_Rep**	
13		f	-	-	off	on				Progressing/ **-**	
14		f			on	off				Ini/ **ProgSucc_Rep**	
15	**PrNOK**	-	-	-	-	-				Ini/ **PrFail_Rep**	
16	**PrOK**	v	-	-	on	-					Progressing/ **ProgSucc_Rep**
17		v	-	-	off	on					Progressing/ **-**
18	**CompOK**	f	-	-	-	on					Ini/ **CompSucc_Rep**
19	**CompNOK**										Ini/ **CoFail_Rep**

APPENDIX D

TEST CASES GENERATED BY THE CONDADO TOOL FOR THE *TELECOMMAND VERIFICATION* SERVICE

The Condado is a tool developed in previous research (Martins, 1999) (Martins, 2003a). This tool is able to automatically generate test cases based on an extended finite state machine. It implements an algorithm of search depth graph, making an interleaving among transitions. The test cases provides an exhaustive coverage of the specification as the tool implements a state-based test method that exhaustively combines all transitions to derive the test cases. So, all possible invalid states and all event/action faults may be revealed (Binder,2000). Condado was developed on the protocol testing concepts basis. Only two PCOs, one for the lower tester (L) and one for the Upper tester (U), are taken into account, for the moment, for the test case generation.

Three PCOs were conceived for the *telecommand verification* service. So, in order to use Condado, we considered PCO_1 and PCO_3 in L, and made mentally a change in the abstract test suite, on substituting L for PCO_1 whenever the next message is TCarriv or receiveTC, or for PCO_3 in the other cases. The U should be substituted by PCO_2.

Only the normal finite state diagram, illustrated in FIGURE 5.2., was submitted to Condado. The 38 abstract test cases generated by Condado are listed bellow. Each case is separated by a white line. The word *senddata* and *recdata* means respectively send and receive the information specified in the second parameter (data, event or a primitive). The first parameter represents the PCO. For Condado, the letters L and T state for lower and upper testers.

```
senddata(L,TCarriv)              recdata(L,receiveTC)
senddata(U,AccOK[-0000])   recdata(L,)

senddata(L,TCarriv)              recdata(L,receiveTC)
senddata(U,AccOK[-1000])   recdata(L,AccSucc_Report)

senddata(L,TCarriv)        recdata(L,receiveTC)
senddata(U,AccOK[-01--])   recdata(L,)
senddata(U,StOK[--000])    recdata(L,)

senddata(L,TCarriv)        recdata(L,receiveTC)
senddata(U,AccOK[-01--])   recdata(L,)
senddata(U,StOK[--100])    recdata(L,StartSucc_Report)

senddata(L,TCarriv)        recdata(L,receiveTC)
```

senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[---00]) recdata(L,)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,ProgSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[---00]) recdata(L,)

senddata(L,TCarriv) recdata(L,receiveTC)

senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[f--01]) recdata(L,ProgSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-01--]) recdata(L,)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--000]) recdata(L,)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--100]) recdata(L,StartSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)

senddata(U,PrOK[---00]) recdata(L,)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,ProgSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--110]) recdata(L,StartSucc_Report)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[---00]) recdata(L,)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)

senddata(U,PrOK[f--01]) recdata(L,ProgSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[v--1-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,PrOK[v--01]) recdata(L,)
senddata(U,PrOK[v--0-]) recdata(L,ProgSucc_Report)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)

senddata(L,TCarriv) recdata(L,receiveTC)
senddata(U,AccOK[-11--]) recdata(L,AccSucc_Report)
senddata(U,StOK[--01-]) recdata(L,)
senddata(U,PrOK[f--01]) recdata(L,)
senddata(U,CompOK[f---0]) recdata(L,CompSucc_Report)
38