Published in DASIA 2005 Conference Proceedings.

Systematic Generation of Test and Fault Cases for Space Application Validation

Ana Maria Ambrosio[1*], Eliane Martins[2],

Nandamudi L. Vijaykumar[1], Solon V. de Carvalho[1]

[1]National Institute for Space Research (INPE), Av. Dos Astronautas, 1758, S J Campos, 12227-010, SP, Brazil (ana@dss.inpe.br)
[2]Institute of Computing - Campinas State University, P.O. Box 6176 - Campinas, 13083-970, SP, Brazil

Abstract

The most critical activity of a V&V process is the test design, as it should be systematic and less dependent of the expert inspiration, mainly for companies developing software whose failures place millions of dollars at risk. Additionally, the trend towards service standardization for the most common space applications motivated us to define a conformance testing methodology added with fault injection concepts. The methodology, named CoFI, defines steps to generate tests that cover the conformance of an implementation with respect to a standard specification. It allows to generate repeatable and controllable test cases and is conceived to be automated and easy to learn. The idea is to translate the service behavior, written in natural language, into a FSM-based notation, then automatically generate test cases. Its main characteristic is to separate in distinct diagrams the normal, the exceptional behavior, those explicitly-specified and those based on mapping an external fault model. The paper describes the CoFI testing methodology and its use in two real case studies. In the first the *TC Verification* service specified in ECSS-E_70-41A was used for showing the feasibility of generating test cases from a publicly recognized standard specification. In the second, an OBDH-Scientific_Experiment protocol, developed at INPE, was used for evaluating the set of test and fault cases created by the CoFI methodology. Preliminary results pointed out that: (i) the methodology is simple and effective specially for creating fault cases based on a fault model; (ii) current standard service specification requires extra information to achieve an applicable set of test cases. Effectiveness metrics of the tests were obtained with specification-based mutant analysis.

1. Introduction

With the increasing use of acquiring third party software products by space agencies and companies, the conformance testing becomes more important.

* Corresponding author. FAX: +55-12-3945-6625 and E-mail address: ana@dss.inpe.br

Contractors make the software specification as part of contract and have to ascertain whether the delivered implementation really conforms to the respective specification. In other cases, the implementations should conform to standardized services. In the latter, the test case suite shall not be designed based on the software source code, as the source code is generally not available. The conformance is a kind of black-box testing so information about how the software should behave under certain inputs (dynamic specification) besides the data description (static specification) is equally important in the software documentation, in order to become the software testable for conformance. In other words, the software documents should also provide testability.

This paper presents a testing methodology named CoFI stated from conformance and fault injection, which establishes a guide for a tester to design conformance test cases and fault injection experiments (or fault cases). The faults to be covered are those mimicking problems caused by the environment space radiation to computational communication system on board of satellites. The proposed methodology includes the use of formal methods, which allows automation, so contributing to cost reduction in developing space applications. In order to mitigate formal methods side-effects [1] like the state space test number explosion, the methodology advocates to model normal and exceptional behavior into distinct models. Once the tests are derived from a normatized specification, in the CoFI approach the tests may be reused, i.e., they may be applied to validate different implementations. The general idea behind the CoFI approach is illustrated in Figure 1. In the figure, the ECSS-E-7941A is shown as an example of a normatized specification to which the CoFI may be applied. The Ferry-injection architecture is the proposed architecture for supporting communication fault injection [9], [10] in the methodology.

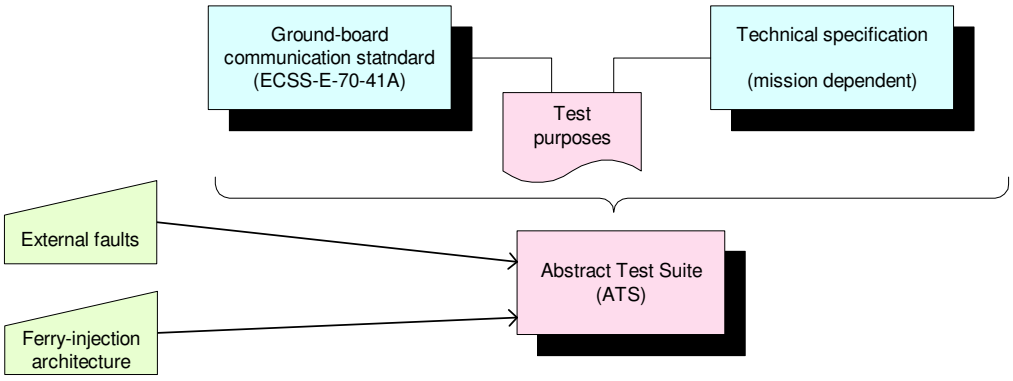

Figure 1: Some artifacts handled by the CoFI Approach

The CoFI testing methodology was conceived for satellite on-board communication software validation, but it is not restricted to this kind of application. The coherence of the methodology steps was empirically shown in two real examples: the Telecomand Verification service (ECSS-E-7041A) and the OBDH-EXP protocol

used in INPE`s scientific satellite missions. An analysis of the efficiency of the CoFI test sequence related to the adequacy criteria of finite state machine mutation faults was performed.

This paper is organized as follows: section 2 describes the testing methodology, section 3 illustrates the application of the methodology in two case studies and section 4 reports the evaluation results and remarks.

2. The testing methodology

The CoFI is a testing methodology that guides the test personnel to design both kinds of tests: conformance (test cases) and robustness (fault cases). The resulting set of test and fault cases are named abstract test suite. The abstract name means implementation independency.

The testing methodology is based on formal methods, however, it uses UML-diagrams, which may be re-used from the development phase (if available), and guides the creation of formal specifications in several smooth steps. These steps are illustrated in Figure 2 in a UML-activity diagram and shortly described in the following:

(1) *Find the services in the specification*: identify the services from a given normatized service description (and a corresponding technical specification) that should be tested, chose one service to proceed in the next steps;

(2) *Identify the service users and the physical communication medium*, i.e., define the systems that interfaces with the service and the hardware and software providing the communication medium;

(3) *Identify observable inputs, outputs, operational variables, the test architecture and the points of control and observation (PCO)*: this step consists of identifying the interfaces of the unit under test (UUT) and the observability provided for executing the tests. Only the accessible inputs should be chosen (otherwise the test can not be executed). Operational variables, defined in [2], describe counters, packet fields, special parameters, timers, etc., which allows one to better understand the UUT. The testing architecture and the PCOs allow preparing the test bed for the test execution. In this work we have been taking into account the Ferry-injection test architecture that provides communication fault injectors [9];

(4) *Describe the functionality of the service as use cases*: one or more use cases may be created. We suggest describing the use case including name, general description, entry and exit conditions and flow of activities, named *scenarios* [3]. Three types of scenarios should be defined: *basic scenario*, which describes the most common flow of activities that characterize the service; *alternative scenarios*, created from the basic scenario on considering all the possible alternative activities the service provides according to the specification; *exceptional scenarios*, map the exceptions as specified. In [3] one may found the rules to describe the alternative and exceptional scenarios.

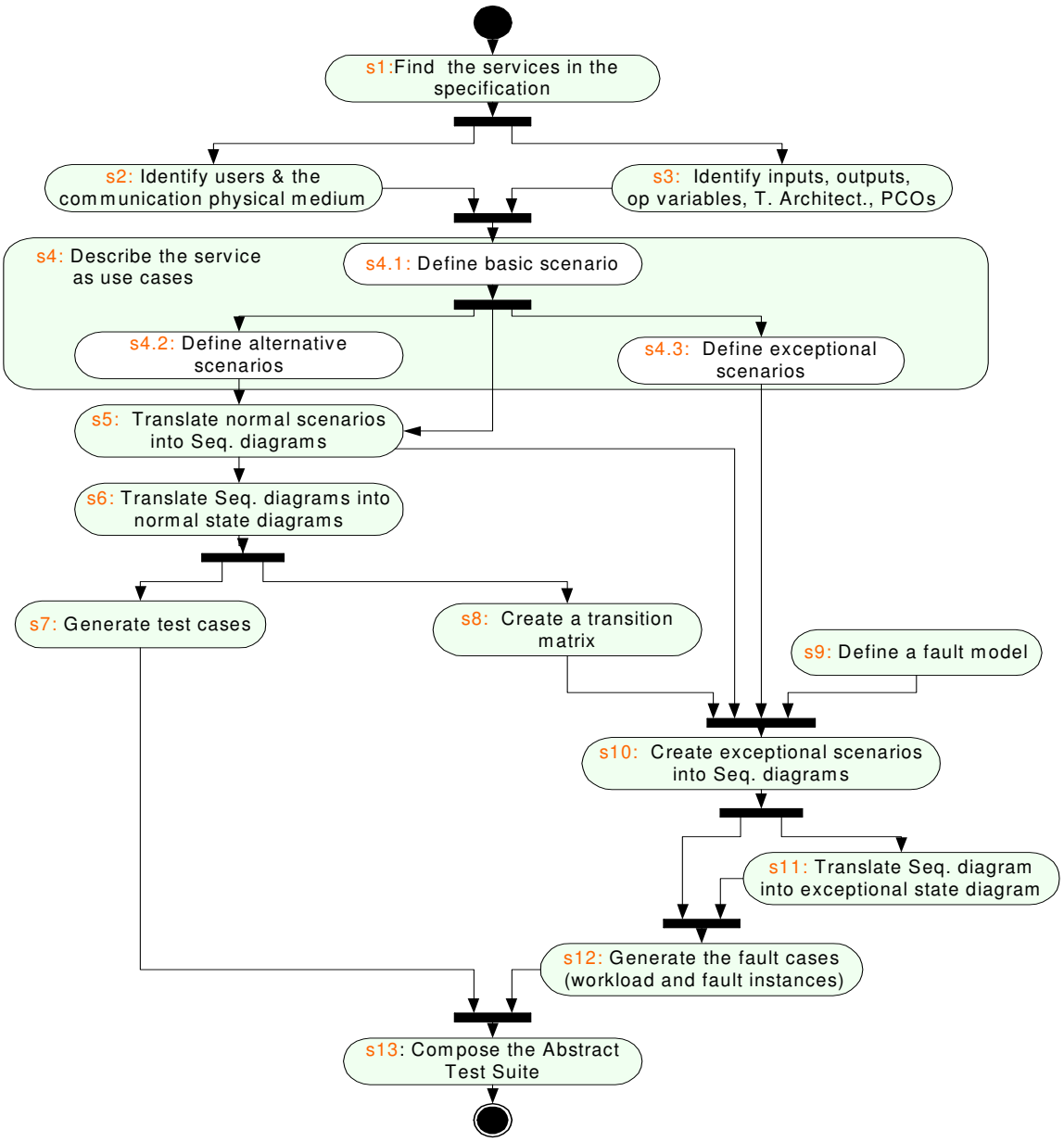

Figure 2: Steps for generating an abstract test suite (activity diagram)

(5) *Translate normal scenarios into Normal Sequence Diagrams*: the basic and the alternative scenarios are described into sequence diagrams;

5.1) identify the entities interacting with the service under test. At least three entities should exist: the unit under test, its peer, and the communication medium. Each entity is a vertical line in the diagram,

5.2) translate the scenario as a sequence of input and outputs of the unit under test (drawing interactions in horizontal lines);

(6) *Translate the Normal Sequence Diagrams into Normal State Diagrams*: in this step normal sequence diagrams modeling all the sequence of inputs and their corresponding observable outputs are translated into state diagrams (which formally are finite state machine -FSM). Rumbaugh et all. describe how to make such a translation [4];

(7) *Generate test cases*: on having the specification in a FSM, one may use a tool able to traverse the transitions of a state diagram or manually combine sequence of events which characterize the test cases for normal situations. Condado tool [5] has been used in the examples illustrated here;

(8) *Create a transition matrix*: translate the normal FSMs of the step 6 into a transition matrix. In this matrix the rows are events, the columns states and the elements are the action (or outputs) and the next state. Generally, this is a sparse matrix (there is non-specified fields). In this step such a matrix should be completed. In doing this, the person responsible for the tests is forced to define what the service should do when a correct event occurs in a wrong state. (Lacks in the standard or the technical specification, means unspecified tests;

(9) *Define the fault model*: the fault model express the communication medium's faults where the service will be run. A communication fault model includes faults like: message lost, delayed, corrupted or duplicated. (Other kind of fault like memory fault (single bit-flip, multi-bit-flip) or processor may be used if the testing architecture may support them for executing the tests). The choice of the fault model is restricted to a fault injector available as the execution of the *fault cases* requires a fault injector. Only the executable faults may be mapped. A fault of a communication fault model is characterized for the following parameters: *instant* (when to apply), *type* (what kind of fault), *mask* (how modify the message in case of corruption fault), *location* (where to inject), *repetition pattern* (how many times to inject). The set of faults comprises the faultload. In order to design the exceptional scenarios and fit them well with the testing execution, fault primitives should be defined;

(10) *Create exceptional scenarios in Exceptional Sequence Diagrams*: this step consists in representing in sequence diagrams the exceptional behavior. Three types of exceptional scenarios are mapped:

10.1) the scenarios identified in step 4.2, i.e., the exceptions stated in the specification,
10.2) the scenarios identified in step 8, whose known inputs lead the system to unknown state or to a crash (sneak paths),
10.3) scenarios created on combining the fault model to the normal scenarios identified in the step 5.

Include the same entities of the step 5 and add the fault injectors. The Ferry-injection architecture comprises two fault injectors: the controller (FIC) and the executor (FIM). The fault primitives, defined in step 9, are used here as a special interaction between the fault injectors, which indicates the fault to be applied in order to establish such a scenario. This interaction is named *fault interaction*.

(11) *Translate the Exceptional Sequence Diagrams into a Exceptional State Diagram*: in this step a FSM modeling all the exceptional scenarios is generated;

(12) *Generate fault cases*: derive fault cases either directly from the sequence diagrams or from the exceptional state diagrams. For the second option, existing tool may be used but information on fault transitions should be translated to commands to the fault injectors. In the first case, although automation is not provided, the information to the injectors is explicitly designed;

(13) *Compose an Abstract Test Suite*: the set of test and fault cases (see steps 7 and 12) are translated into a testing language. ISO defined the TTCN[1] as a test notation independent of any implementation language [6]. The PLUTO[2] [7], defined by ESA may be also chosen to write the tests. Another option is the UML Testing Profile [8] recently defined by OMG.

3. Experimental evaluation through case studies

3.1. ECSS- TC Verification – an standard service

The TC verification service specified in ECSS-E-7041A was used as a case study for the analysis of the feasibility of the CoFI methodology when stating test case design from a standard service description. On firstly applying the CoFI steps based exclusively on such a standard description, the resulting formal specification is presented in Figure 3. A one-state diagram represents the normal service behavior, which seems to be little realistic.

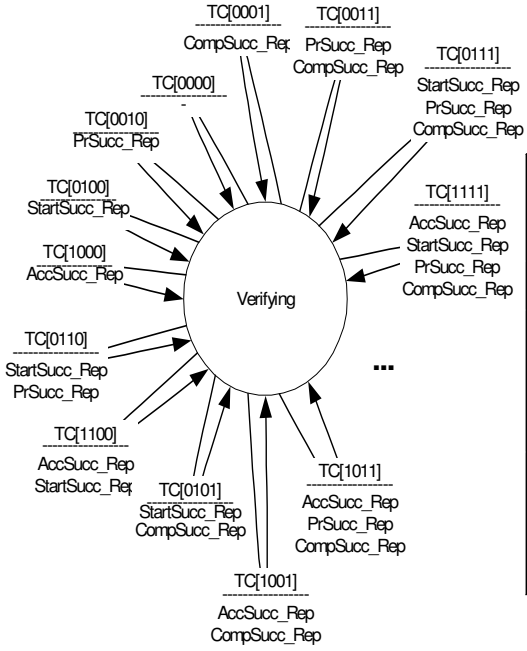

Figure 3: (a) state diagram

(b) test case in TTCN notation

Test Case Name : telecommand acceptance, start and completion execution report with success **Purpose** : To verify the telecommand execution.			
Behavior Description	**Constraint**	**Comments**	**Expected Output**
PCO_1 ? TC PCO_3 ! AccSuc_Rep StartSuc_Rep CompSuc_Rep	[Ackbits = 1101]	A TC arrives requiring reports for acceptance, starting and completion execution steps	Acceptance Success Report Start-execution Success Report 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_3: is the interface with the ground system TC: is an event that carry out telecommand packet data for TC verification purposes.			

[1] Testing and Test Control Notation

[2] Procedure Language for User in Test and Operations

The test cases are straightly identified, each arc represents a test case. Figure 2(b) illustrates a test case written in TTCN, in which the two points of control and observation identified are: PCO_1 - from where the service receives TC packets and PCO_2 - from where the service sends TM packets.

In order to create more realistic test cases, some assumptions were taken, concerning the communication between the service and the TC executor. Information about the TC execution must be identified by the service; otherwise it is impossible to produce the requested reports. This interface was assumed in an abstract form, thus giving flexibility to different implementations. This assumed interface, a PCO_3 from where commands like: StartOK, PrOK(p), FinOK, StartNOK, PrNOK(p), FinNOK are received by the service, indicates respectively that a TC execution have stated, progressed and finalized correctly, or incorrectly. In providing such a definition, more realistic test cases were derived. The fault model applied to this interface includes message lost, delayed and corrupted. Part of the sequence diagram modeling the delayed message from the TC Executor is shown in Figure 4.

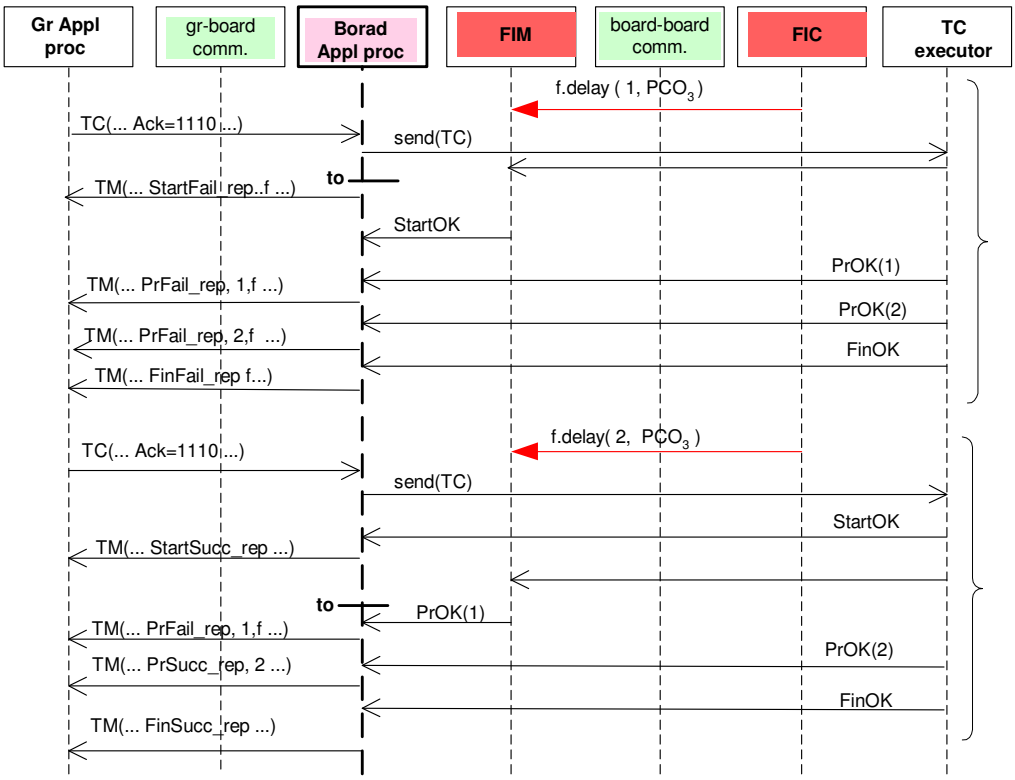

Figure 4. Exceptional Sequence Diagram for delay in PCO_3

The normal and exceptional state diagrams are presented in Figures 5 and 6 respectively. Figure 6 includes only the specified exceptions. The diagram modeling exceptions related to the 10.2 are not presented here for lack of space.

On combining the fault model (delay and lost message) and information corruption one may have the state diagrams shown in Figure 7(a) and 7(b). In the latter diagrams the state named *received TC** represents the set of states: *Verify fields, Cksum correct, Apid correct Length correct, Type correct and Subtype correct, Fields correct*.

The set of test cases were created by submitting the FSMs to Condado. A total of 107 test and fault cases were derived.

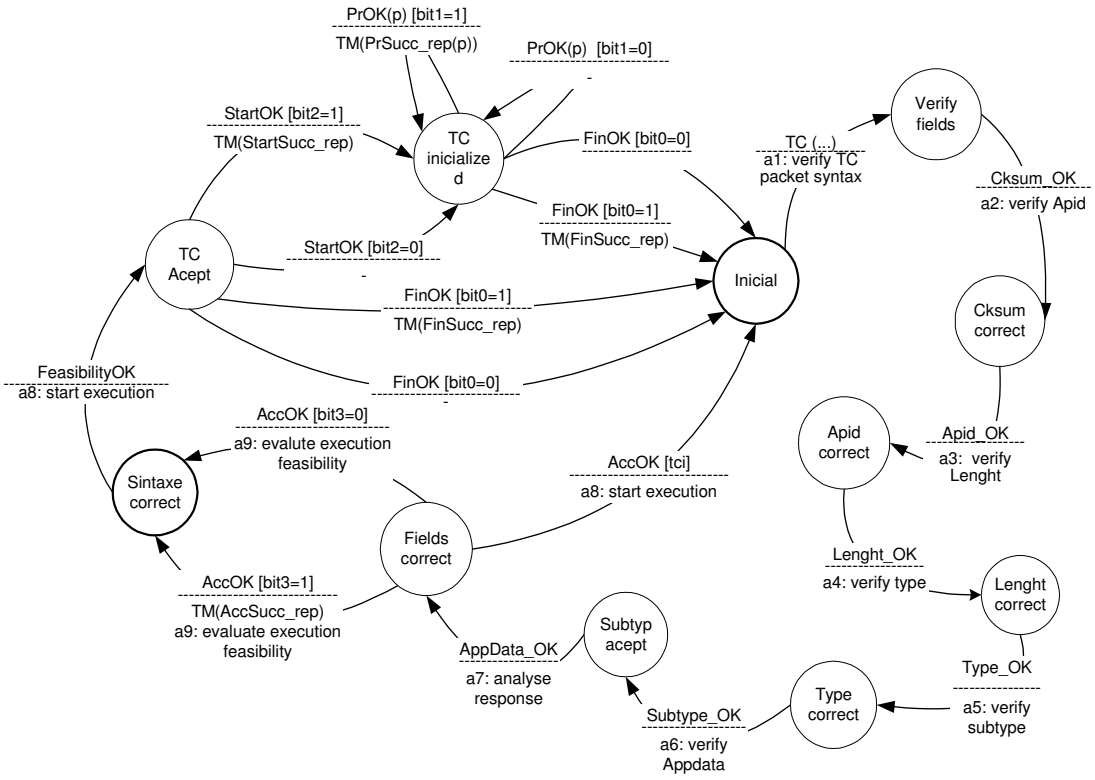

Figure 5: Normal State Diagram

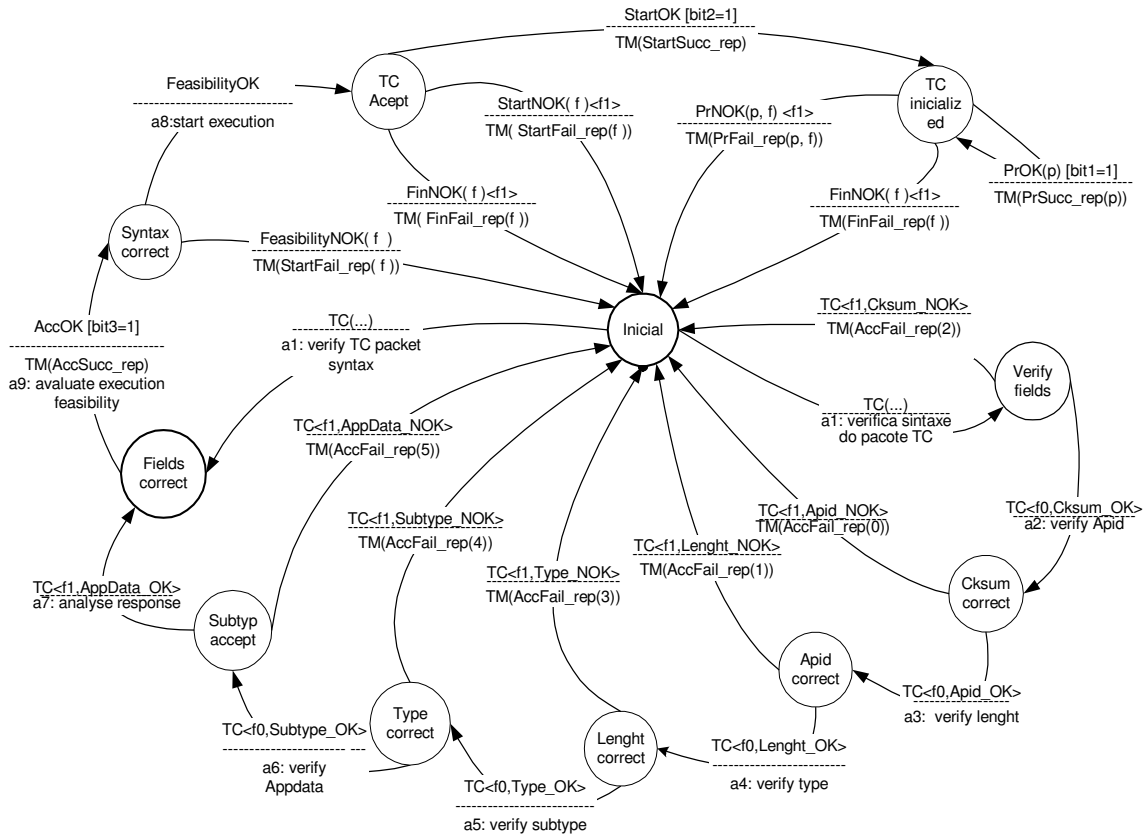

Figure 6: Exceptional State Diagram

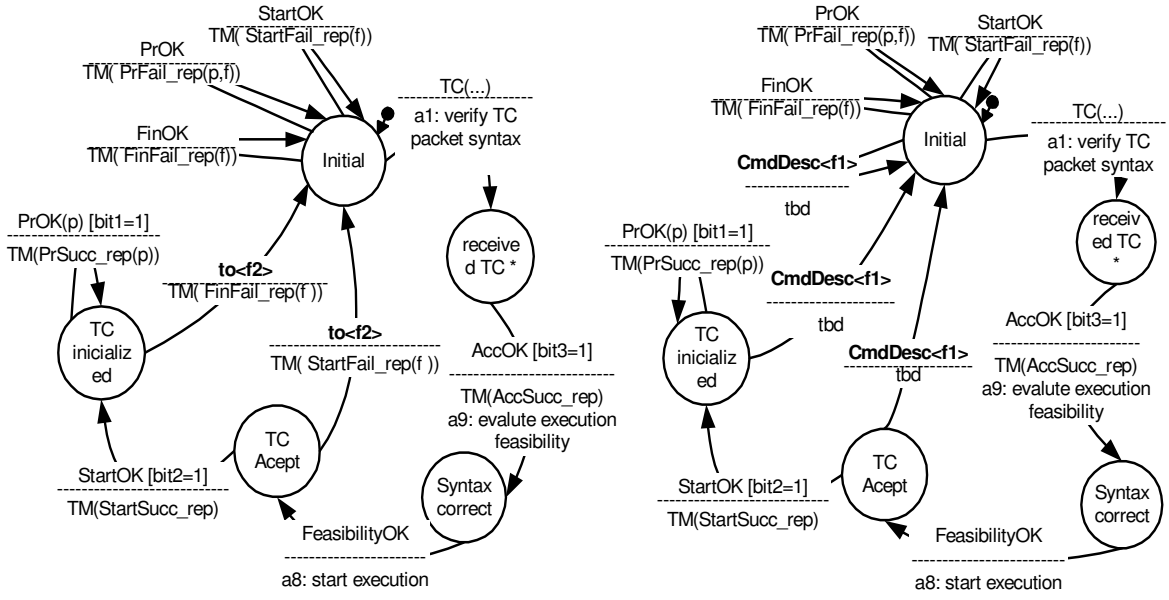

Figure 7: Exceptional State Diagram for (a) delay and (b) corruption

3.2. - OBDH-EXP protocol:

In order to evaluate the CoFI approach in an application where tests could be executed, the *commands recognition software*, part of the communication protocol developed at the National Institute for Space Research (INPE) was used. This protocol is in charge of providing the communication between on-board data handling computer (OBDH) and an intelligent satellite payload for Brazilian scientific satellites, the APEX – *Alpha, Proton and Electron Monitoring Experiment in the Magnetosphere*. The APEX sends data only under OBDH request, in a master-slave relationship. The physical communication is the RS-422 serial interface in asynchronous mode. Whenever the experiment fails, the OBDH times out. From this case study, we present some fault cases generated. As this is an in-house developed software, the test and fault cases, the workload (inputs) and the fault scenarios were written in symbols recognized by the test personnel and applied to a corresponding implementation (see Table 1). In this case study all information about the implementation was available, so the full test suite could be executed and the results could be analyzed. The diagrams are not shown here for lack of space.

Table 1: Fault cases for OBDH-EXP protocol implementation

	Input	***Observed Output***	***Comments***
1	EA	timeout	SYNC - not-recognized
2	EB (delay)	no reaction	No reaction as expected
3	EB (delay) 92	timeout	EXPID - not received
4	EB 91	timeout	EXPID - unknown
5	EB92 20	timeout	TYPE - invalid
...	...	...	...
33	EB92 1A 4 0000 0000	timeout	Incorrect address

4. Conclusion

The paper presented a methodology to systematically designing tests which focus on errors caused by the space environment problems. The experiments pointed out that the methodology is effective to reduce testing costs in a mission as test and fault cases may be re-used. Additionally, the methodology provides mechanisms to plan the test coverage and effort early in the development.

In order to evaluate the CoFI set of test and fault cases, the specification-based mutation concept was used, supported by prototyped tools of the PLAVIS project [11]. A FSM of the service behavior including the total specification was used for mutant creation. The mutants are classified according to the faults they represent as shown in Table 2 for both examples presented in this paper. Results of the experimental evaluation with the case studies are summarized in Table 3, in which number of test and fault cases, partial models and the mutation score are presented.

Table2: Mutant operators

Mutant Operators	# mutants for *TC Verification*	# mutants for OBDH-Exp Protocol
Modified Output	546	48
Deleted Output	26	16
Deleted Transition	21	9
Modified input	508	68
Deleted Input	30	16
Modified Initial state	10	4
	1141	**161**

Table3: Evaluation results

	# Partial models (FSMs)	# External fault types	Test cases	Fault cases	Mutation score
TC Verification	5	3	21	86	0.978
OBDH-Exp Prot.	5	4	20	594	1.000

Some difficulties were found in mapping the *TC Verification* service (ECSS-E-7041A) into FSMs. This difficulty may be partially explained by omissions in the service description given in natural language. It points that such standards lack of testability requirements thus; extra information is necessary to achieve an applicable set of test cases.

Preliminary results pointed out that the methodology is simple and effective specially for creating fault cases based in a fault model. Additionally, on establishing the test architecture and the points of control and observation previously in the test design, as the CoFI methodology states, the tests may help the identification of lacks even in a publicly recognized space service specification.

References

[1] Lai, R. A survey of communication protocol testing. The Journal of Systems and S/w, 62, 2002, pp. 21-46.

[2] Binder, R. Testing Object-Oriented Systems-Models, Patterns and Tools – Addison-Wesley 2000

[3] Ryser, J. Glinz, M. (2000). SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. Technical Report, Institut für Informatik, Universität Zürich.

[4] Rumbaugh, et al. Object-oriented Modeling and Design. Prentice Hall, 1991.

[5] Martins,E; Sabião,S.B; Ambrosio, A.M. ConData: a Tool for Automating Specification-based Test Case Generation for Communication Systems. S/w Quality Journal, 8 (4) (1999) 303-319.

[6] European Telecommunications Standards Institute. ETSI ES 2001-873-1. Methods for Testing and Specification (MTS) - The Testing and Test Control Notation, v2.2.1, 2003.

[7] European Space Agency- ECSS-E-70-32A - Space engineering Procedure Language for User in Test and Operations. Issue Draft 5, 2001. Noordwijk: ESA publication Division.

[8] Management Grou (OMG). UML 2.0 Testing Profile Specification. Disponível em:<http://www.omg.org/docs >.

[9] 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.

[10] 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. Disponível na biblioteca digital URLib: <WWW.SPACEOPS2004.ORG>. Acesso em: 2 sep. 2004.

[11] PLAVIS (2005). Plavis - platform for software validation & integration on space systems. Online in: http://www.labes.icmc.usp.br/plavis/index.html, last access: 29/04/2005.