JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 3, April 2006

A Conformance Testing Process for Space Applications Software Services

Ana Maria Ambrosio*
National Institute for Space Research (INPE), S J Campos, SP, Brazil, 12245-970

Eliane Martins†
Campinas State University (UNICAMP), Campinas, SP, Brazil, 13083-970

Nandamudi L. Vijaykumar‡ and Solon V. de Carvalho§
National Institute for Space Research (INPE), S J Campos, SP, Brazil, 12245-970

Comprehensive tests for space applications software are costly but extremely necessary. This software must be reliable and produced within schedule and budget. In a tentative of making the space mission software development for space agencies and industries more cost-effective, the European Committee for Space Standardization (ECSS) has been compiling a set of standards that specify the common core of some space application systems. Once the set of services is standardized, the conformance problem is raised. In this paper we present a testing process for standardized services, which is based on the IS-9646 standard for ISO protocol conformance testing. The process includes an approach to derive test and fault cases by combining conformance testing concepts with the software-implemented fault injection (SWIFI) technique. One advantage of this process is the generation of a re-usable abstract test suite which improves the testing effectiveness. Reliability and convergence in the test cases are increased the more the tests are applied. Additionally, the evaluation of the software behavior under external faults may be performed under the repeatable set of fault cases. The approach is illustrated with abstract test and fault cases derived for the *telecommand verification* service stated in the ECSS-E-70-41A standard. These services, successfully adopted in many missions, define the application-level communication between on-board and ground applications.

I. 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. Mazza[1] points out that the need of space software standardization is increased by the fact that nowadays 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 defining a set of standards for management, quality assurance and engineering since 1993, in the European Cooperation for Space Standardization (ECSS).[2]

Received 12 January 2005; revision received 21 December 2005; accepted for publication 29 December 2005.

* Senior Technologist, Ground System Division, P.O. Box 515, São José dos Campos, 12245-970, SP, Brazil, ana@dss.inpe.br

† Researcher, Institute of Computing, P.O. Box 6176, Campinas, 13083-970, SP, Brazil, eliane@ic.unicamp.br

‡ Researcher, Computing and Applied Mathematics Laboratory, P.O. Box 515, São José dos Campos, 12245-970, SP, Brazil, vijay@lac.inpe.br

§ Researcher, Computing and Applied Mathematics Laboratory, P.O. Box 515, São José dos Campos, 12245-970, SP, Brazil, solon@lac.inpe.br

In the engineering branch of the ECSS one may find a recently defined standard, the ECSS-E-70-41A,[3] 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[4]: 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.

The services of the ECSS-E-70-41A are often performed many times in several formats by multiple contractors during the life cycle of the satellite before the satellite is ever deployed for mission operations.[5] So, it would be more cost-effective to establish a set of previously defined abstract test cases, rather than creating completely new test cases for each implementation. Abstract test cases can be defined as a complete set of actions to achieve a specific test purpose (a pseudo-code) that has to be converted into a language that will be executed to determine a verdict on the unit under test. Moreover, since PUS services are publicly available as standards, it is essential that their implementations (computer programs) conform to their corresponding specifications.

In this paper we propose a testing process for validating the PUS services, named CoFI—conformance-and-fault-injection. It is based on both the ISO standard for protocol conformance testing and fault-injection testing technology. In the CoFI process, normal and exceptional behaviors are independently modeled from the service description for test and fault case derivation.

The next section presents an overview of the PUS stated in the ECSS-E-7041A standard; section III presents the conformance-and-fault-injection testing process; section IV addresses the approach for specifying the abstract test suite with fault and test cases; section V illustrates the abstract tests for the *telecommand verification* service. Finally, section VI discusses related works and section VII concludes the paper.

II. Space Communication and the Role of the Packet Utilization Services

Communication between ground and on-board systems is provided by many subsystems (hardware and software) at several communication levels. Figure 1 shows the main elements of a space mission: satellite, ground station and satellite control system.

The ECSS-E-7041A specifies services to be provided by the satellite (service provider) during its communication with the Satellite Control System (service client). These services, named packet utilization services (PUS), are characterized by a set of reports and requests transferred from/to on-board applications. For the requests and reports to be effective, ground and on-board communication uses the layered CCSDS-defined Telecommand System,[6] shown in gray in Fig. 2. This system comprises 5 layers, from the Physical to the Packetization layers, providing all the necessary controls to accurately deliver the telecommand (TC) application data 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 intermediate formats supporting the protocol communication.

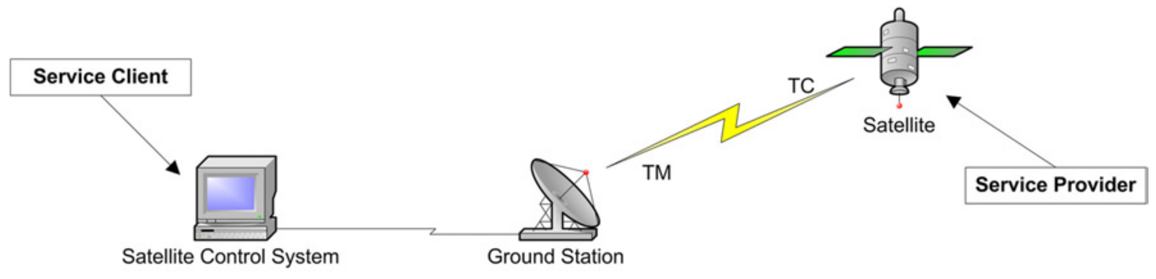

Fig. 1 Ground and on-board systems.

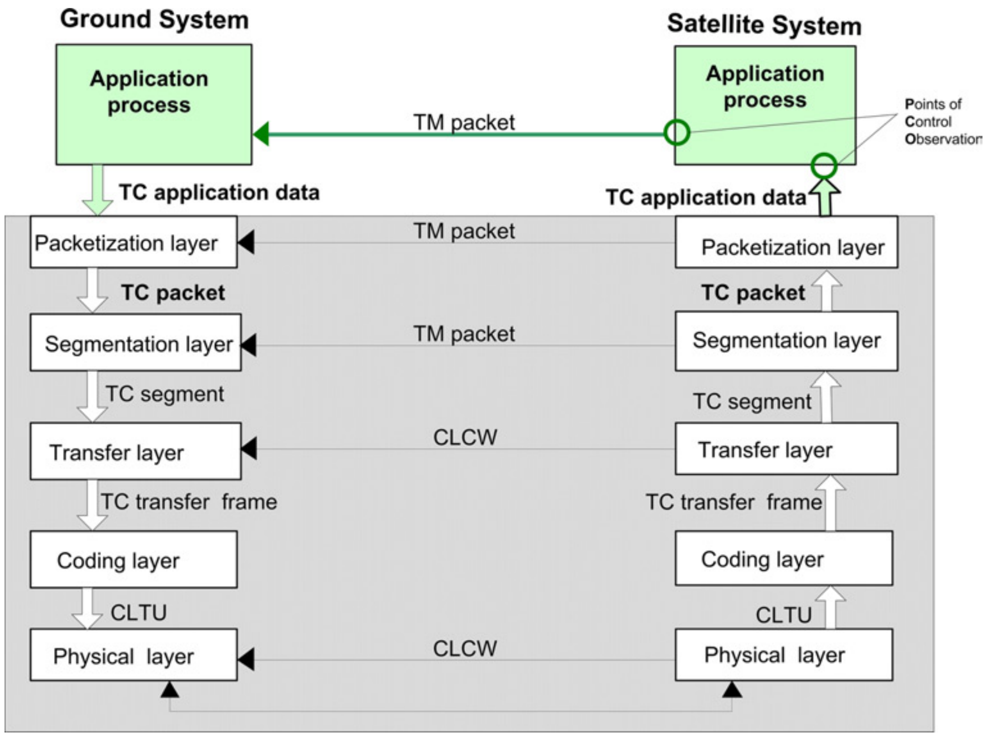

Fig. 2 Telecommand system layers.

The structure of the PUS services over the leveled CCSDS protocols is similar to the ISO-OSI basic reference model,[7] so the conformance testing process defined in the IS-9646 standard seems suitable, since points of control and observation (PCOs) can be established. PCOs define the system testability as they allow the tester to control and observe the occurrence of testing events.

The Packet Utilization Services are not only applied in different missions, but also in different software implementations 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.

III. CoFI: Conformance-and-Fault-Injection Test Process

A testing process is a "course of action to be taken to perform testing", according to IEEE Software Standards. It consists of specifying the test cases, adjusting the test cases to the test execution mean, executing the test, analyzing the results and producing reports about the test execution. In the CoFI testing process the evaluation power of the conformance testing given in the IS-9646 standard was extended with the SWIFI to accelerate the rate of exceptional events.

A. Conformance Testing

The International Organization for Standardization (ISO) and the International Telecommunication Union (ITU) have standardized procedures for communication protocols with practical guides for conformance testing. These standards, which are stated in the IS-9646 standard for Conformance Testing Methodology and Framework[8,9] have allowed testing laboratories around the world to perform ISO protocol testing and certification.

In the IS-9646, the testing process comprises activities in which abstract test suites are derived to certify a protocol implementation. A set of test cases is generated according to the specification. This set of test cases is known as abstract test suite as additional information about the specific implementation has to be incorporated in the test cases in order to make them executable. This approach makes an abstract test suite reusable to different implementations.

Why use the IS-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)[10] service implementation. Mertens[11] presents the validation of the SLE implementation in the cross-supporting program carried out by ESA and Jet Propulsion Laboratory (JPL).

B. Software-Implemented Fault-Injection

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.[12] SWIFI has been adopted to accelerate the rate of fault events during testing, which enables software designers to identify and remove the faults of a unit under test. 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.[13]

Why use SWIFI in conformance testing? The first reason to incorporate SWIFI in the testing process was to accelerate the rate of problems occurring in the unit under test. 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*) cannot be pre-determined, and hence an absolute conclusion about the dependability of the software cannot be drawn, this technique represents a step towards more reliable software behavior. SWIFI also has the advantage of being easily expandable to new classes of faults and it is free of physical interference as well. Moreover, fault injection has proved to be invaluable in space application testing due to its capability to model common space environment faults which commonly occur in a system's operational phase.[14]

C. The CoFI Activity Flow

The activities for performing the CoFI are organized into three phases: preparation for testing, test operation and test report production. The phases and activities have been chosen to comply with the IS-9646. Figure 3 illustrates the CoFI activities flow, where each box represents an activity and the arrows show what is received and produced by the activity. The first testing activities occur in parallel with the implementation. Table 1 summarizes concepts and acronyms used in this description based on IS-9646. The activities of the first phase, in gray boxes (Fig. 3), differ from the IS-9646 by dealing with fault definitions and formal specifications that have been added to automate the process.

D. The Activity Description

The CoFI testing process comprises six activities presented below.

1) ***Define Test Purposes:*** the *preparation for testing* phase starts with the definition of *test purposes*. This activity receives the service standard specification and produces the test purposes. According to IS9646, 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.
2) ***Specify Abstract Test Suite (ATS):*** This activity receives the test purposes and produces an abstract test suite. An ATS is provided for each ECSS-E-70-4A service. The ATS comprises *test cases* and *fault cases* which are implementation-independent and focused on the content of the specification. Scenarios are identified for one test purpose, taking into account the service capability, i.e., 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

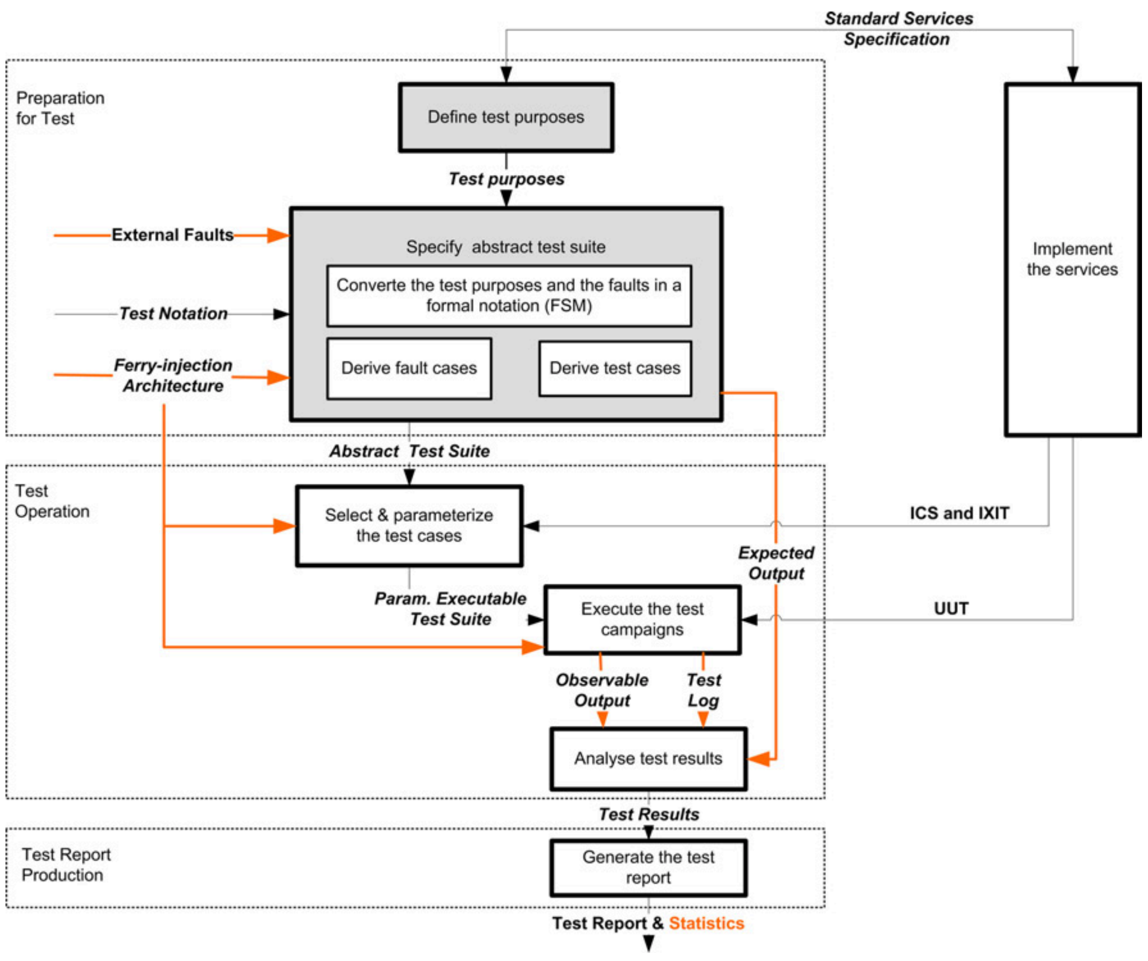

Fig. 3 CoFI testing process.

be received (by situations we mean, the different values carried into packet fields, counters, timers, that cause behavioral changes. These values may be chosen using a combination of the testing data selection techniques: Boundary Value Analysis (BVA) and Equivalence Class partitioning (EC).[15]) Just to clarify how these techniques can be used, a very brief explanation is in order. The EC technique consists in selecting distinct sets where each set contains values that are considered equivalent. This means that not all values of each set have to be tested. By applying the BVA to each set, it is possible to achieve a reasonable set of test cases. All the specified reports should be covered by at least one test or fault case. 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 to derive the fault cases. Assumptions about the service testability (i.e., the points of control and observation) are required for 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 methodology. This methodology[16,17] is shortly presented in Section IV.

Besides the points of control and observation, facilities for executing the tests and injecting the faults also influence the definition of cases in ATS. The test architecture strongly affects the conformance requirements that can effectively be checked by the testing means.[18] As a result of previous research we have adopted the *ferry-injection architecture*, which is illustrated in Fig. 4, to run the tests and execute fault-injection.[19] In this architecture, the testing system and the system under test are as independent as possible of each other. This

Table 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 abstract test case is a complete and independent specification of the actions required to achieve a specific test objective. 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.
Implement. 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 test 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 testing.

independence is obtained by the ferry-clip concept, whose idea is to transport test data transparently from the testing system to the system under test.[20,21] The ferry-clip is composed of the following components: active ferry, passive ferry and ferry-channel. The test engine, the main component of the Testing System uses an executable test script to run the tests. The executable script includes both, the lower tester (LT) and the upper tester (UT), whose concepts may be found in IS-9646.[8] For space applications no distinction between upper and lower tester is required, instead multipoints should be recognized. In this architecture only communication faults have been controlled by the Fault Injection Controller (FIC) and executed by the Fault Injection Module (FIM). The communication faults involve loss, duplication, delay or alteration of messages. Experiments in configuring this architecture for testing space applications may be found in.[22,23]

Once the test/fault cases are defined, they may be translated into either TTCN[¶] *testing notation* or to 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 phases.

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. This choice takes into account not only the test architecture, but also the testability of the UUT. The selected cases are completed and parameterized, with the provided values of counters and timers documented in ICS and IXIT. Here, the parameterized test and fault cases are translated into an executable language. In this stage, the test set is named *parameterized executable test suite* (PETS). In CoFI, PETS may be written in PLUTO,[24] or in any other script-like language.
4) ***Execute the Test Campaigns***: in this activity the PETSs are ready and the testing 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 in *Test Log* not only for result analysis but also for future reference. The *observable output* is associated to each executing test/fault case.

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

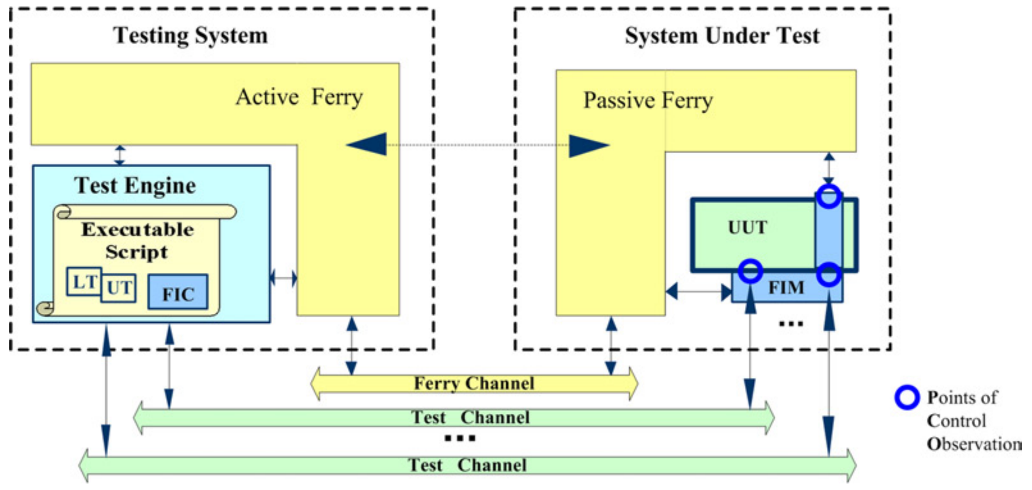

Fig. 4 Ferry-injection architecture.

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

Roughly, the CoFI differs from the IS-9646 test process in the following aspects: (i) it needs to define faults; (ii) it includes fault cases besides the test cases in the ATS, (iii) 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)), (iv) it generates statistics about the test results, (v) it includes the ferry-injection architecture, (vi) it generates test cases from a formal specification.

IV. Approach for Specifying the ATS

The *specify ATS* activity, defined in the CoFI process (see section III), includes an approach to derive the abstract test and fault cases. This approach requires the tester to translate the service specification into a formal notation, a Finite State Machine.

On one hand, the formal notation-based specification allows the use of formal algorithms to automatically generate test cases, thereby significantly reducing time and effort with tests and assuring a greater coverage of the specification. On the other hand, however, getting the specification into a formal notation is hard and time-consuming work, requiring much iteration.[25]

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 are written in separated diagrams from scenarios of exceptional situations. From the former, the test cases are generated and from the latter, the fault cases are generated. Thus, the total number of cases generated from each diagram is smaller when compared to that used from a global modeling.

The general flow of the CoFI approach has already addressed a quantitative evaluation of the test case set and has illustrated the test set results obtained with two different examples.[16] The same CoFI has also focused on deducing faults to be injected to test an in-house developed protocol for communication between a scientific experiment and a principal satellite on-board data handling is presented.[17] In the following, the steps of the CoFI methodology focus on the *telecommand verification* service:

1) find the services in the specification;
2) identify the service users (servers and clients) and the physical communication medium;
3) identify the requests, reports (i.e., the observable inputs, outputs), and the *operational variables.* (The operational variables[26] defined here correspond to those variables whose variation of their values causes service behavior changes.) Next, define the test architecture and the points of control and observation (PCO);
4) describe the service functionality as a *use case* in a basic scenario, then, create *normal* and *exceptional scenarios* from the basic scenario, taking into account situations deduced from the operational variables. A scenario is a specific realization of success or failure of a use case described in a finite number of actions. The exceptional scenarios represent all the exceptions; cover the error codes to be dealt with by the service. Literature has already addressed on some rules to describe and identify scenarios;[27]
5) translate normal scenarios into *Normal Sequence Diagrams*: design the sequence diagrams on mapping all requests, reports and operational variables exchanged between the server and client providers of the *normal scenarios*:
   a. 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,[16]
   b. describe the scenario as a sequence of input and outputs of the unit under test (horizontal lines);
6) translate[28] the Normal Sequence Diagrams into *Normal State Diagrams*: create a normal finite state diagram with the events that trigger the normal, explicitly-specified outputs;
7) generate *test cases*, submitting the *Normal State Diagrams* to reachability analysis algorithms for graph search;[29,30]
8) create a *transition matrix*: translate the Normal State Diagram of the step 6 into an event $\times$ state matrix, where, each line is a valid event; each column is a state (the intervals between events in the sequence diagram of the server provider line), each cell is both an output (or an action) and the next state. The outputs may be: (i) a normal, explicitly-specified output; (ii) an exceptional, explicitly-specified output; (iii) an abnormal termination or rejection of the input (illegal transition); (iv) a non-specified output (an output is not represented in the specification, however, the implementation emits an output such as a diagnostic messages for eventual internal errors). The events may be caused by request arrivals or by changes in the operational-variable values;
9) define a *fault model*, which expresses communication medium's faults, memory or processor faults where the service will be run. This model helps the definition of the fault cases on next step;
10) create exceptional scenarios in *Exceptional Sequence Diagrams*:
    a. besides the three entities (the unit under test, its peer, and the communication medium) identified in step 5, include the fault injectors. The Ferry-injection architecture comprises two fault injectors: the controller (FIC) and the executor (FIM) (Fig. 4),
    b. describe the exceptional scenario as a sequence of input and outputs of the unit under test (horizontal lines).[17] Three types of exceptional scenarios may be mapped:
       i. those stated in step 4,
       ii. those identified in the transition matrix (step 8),
       iii. those resulted from combining the fault model to the normal scenarios described in the step 5,
    c. indicate the fault type interacting between the fault injectors and what takes place on the fault occurs;
11) translate the Exceptional Sequence Diagrams into an *Exceptional State Diagram;*
12) generate *fault cases*: derive a *fault case* from each scenario in obtaining the sequence of inputs (all normal inputs arriving in the unit under test), the corresponding expected output and the fault description parameters. Communication faults are characterized by the following parameters: *when*—time or the message identification, *what*—fault type, *how*—mask or byte position, *where*—the communication fault associated to a PCO.

It is worth clarifying that this approach assumes the following definitions: an implementation-independent test is characterized as a sequence of inputs and corresponding expected outputs. A pair [input, expected output] is represented in a transition of a finite state machine. So, a test case is a path, starting from the initial state and finishing in a reachable state. The fault to be injected is characterized by the instant to inject and by the fault instance. The instant is defined relatively by the set of inputs from the initial state and subsequent inputs up to a specific (reachable) state.

V. Abstract Test and Fault Cases for the ECSS-E-70-41A

This section presents an overview of the approach to derive test and fault cases applied to the specified *telecommand verification* service.[3] According to the ECSS_E_7041A, a telecommand is carried on the telecommand packet structure, which is composed of a header and a packet data field as shown in Fig. 5.

The *telecommand verification* service provides feedback about the execution of a given telecommand at any of its meaningful stages. In 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. This indication is done by setting any of the four-bits of the Ack field of the packet data field. The Ack bits, when set to 1 indicates that the service is required to generate the respective reports: acceptance-report, start-report, progress-report and completion-report. Table 2 summarizes the service capabilities by listing the reports for success and for failures.

Test case: In step 3 (see Section IV), the requests and reports are obtained from the capability set (see Table 2); the operational variables are obtained from the Ack bits whose four-bit combination leads one to deduce 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 for conformance tests. We are assuming three PCOs for this service: PCO_1 – observes the telecommand packet data arrival, PCO_2 – allows us to recognize the status of the telecommand execution and PCO_3 – allows us to observe the generated telemetry reports to be sent to the service client. Figure 6 illustrates the three PCOs for the *telecommand verification* service.

The test cases are based on events triggered in any of the three PCOs. From PCO_1 TC packets flow. If the bits of the Ack field of the incoming TC are set, for example, to 1101, one may say that the service should generate the following reports to the client: TC_acceptance_Success, TC_execution_started_Success and TC_execution_completed_Succes, but the TC_execution_progress_Success report is not required. The abstract test case for this scenario is presented in Table 3 in TTCN.

The corresponding executable test case is illustrated in Table 4, in a C-like language. Intermediary steps of the approach include the creation of some UML diagrams which have not been included here for the sake of space.

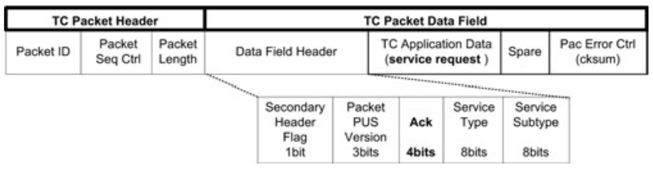

Fig. 5 Telecommand packet structure.

Table 2 Capability set of the telecommand verification service.

Sub-type service	Requests and reports
1	TC acceptance Report – Success
2	TC acceptance Report – Failure
3	TC execution started Report – Success
4	TC execution started Report – Failure
5	TC execution progress Report – Succes
6	TC execution progress Report – Failure
7	TC execution completed Report – Succes
8	TC execution completed Report – Failure

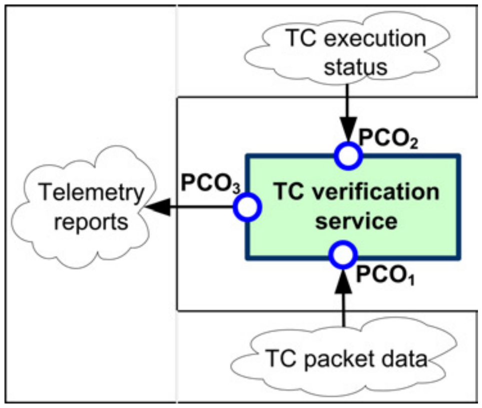

Fig. 6 Points of control and observation for the *telecommand verification* service.

Fault case: the *telecommand verification* service specifies that reports indicating failure are always transmitted, so failure reports are not required in packet data fields. 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: (i) the necessary inputs to bring the UUT to the state in which the fault should be injected and (ii) the fault parameters to be passed to the fault injector. The necessary inputs are derived from the exceptional state machine, which are: TCApp_data in PCO$_1$ and AccOK, StartOK in PCO$_2$. The fault parameters are: *when* – following the last

Table 3 Dynamic behavior description test case – abstract.

Test Case Dynamic Behavior

Test Case Name: telecomand acceptance, start and completion execution report with success
Purpose: To verify the telecommand execution.

N.	Behavior description	Constraint	Comments	Expected output
1	PCO$_1$? TCApp_ data		TC packet data arrive requiring reports for acceptance, starting and completion execution steps	–
2	PCO$_2$? AccOk PCO$_3$! AccSuc_Rep	[Ackbits = **1**101]	AccOk event trigged for verification fields	Acceptance Success Report
3	PCO$_2$? StartOk PCO$_3$! StartSuc_Rep	[Ackbits = 1**1**01]	StartOk event trigged at the execution step	Start-execution Success Report
4	PCO$_2$? ProgOk	[Nstep = 1 = Max] [Ackbits = 11**0**1]	ProgOk event trigged at the end of the second execution step. Only one progressive step exists, i.e., the TC requires 3 steps (start, progress and completion), but the progress report is not required	–
5	PCO$_2$? CompOk PCO$_3$! CompSuc_Rep	[Nstep = 1 = Max] [Ackbits = 110**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.
PCO$_2$: is the interface with the application executing the TC.
PCO$_3$: is the interface with the ground system.
TCApp_data: is an event that carries out Telecommand packet data for TC verification purposes.

Table 4 Dynamic behavior description test case – executable.

Test Case Name: telecomand acceptance, start and completion execution report with success

```
// Inicialize the variables
        TC_App_data.header= 0×ab 0×79 0×08; // hypothetic value
        TC_App_data.Ack= 0×13;                  // 1101 binary;
        PCO1 = port1; // address point of control and observation
        PCO2 = port2;
        PCO3 = port2;
//open ports
        open_port (PCO1); open_port (PCO2); open_port (PCO3);
//Send and receive data in the ports according to the test case
        senddata(PCO1, TC_App_data); receivedata();
        senddata(PCO2, AccOK); receivedata(PCO3, TM_AccSuc_Rep);
        senddata(PCO2, StartOK); receivedata(PCO3, TM_StartSuc_Rep);
        senddata(PCO2, ProgOK);
        Nstep = Nstep ++;
        while Nstep <=MAX { sendata(PCO2, ProgOK); receivedata(PCO3,);}
        senddata(PCO2, CompOK); receivedata(PCO3, TM_CompSuc_Rep);
        close_port (PCO1, PCO2, PCO3);
```

Table 5 Dynamic behavior description fault case.

Test Case Dynamic Behavior

Test Case Name: Telecommand start execution report with success and failure in the progress execution
Purpose: To verify the telecommand execution.

N.	Behavior description	Constraint	Comments	Expected output
1	PCO_1 ? TCApp_data		A TC packet data arrives requiring reports for starting execution step	–
2	PCO_2 ? AccOk	[Ackbits = 0100]	AccOk event trigged at the end of fields verification	–
3	PCO_2 ? StartOk PCO_3 ! StartSuc_Rep	[Ackbits = 0100]	StartOk event trigged at the end of the first execution step	Start-execution Success Report
4	*Fault: PCO_2 ? ProgNOk PCO_3 ! ProgFail_Rep		ProgNOk event trigged at the end of the second execution step	Progress Failure Report

StartOK event, *what* – generate an extra message, *how* – provide an extra message masked as ProgNOK, *where* – in PCO_2. The abstract fault case for this scenario is shown in Table 5 in TTCN notation.

VI. Related Work

In the literature, the SAMSTAG[31] and TOP[32] approaches which are similar to CoFI, define a model-based test case generation method based on the IS-9646. The test method named 'SDL and MSC based test case generation' (SAMSTAG) aims at developing the automatic generation of abstract test cases for telecommunication protocols. It considers both, the specifications and the formally-defined 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.

The N+ state-based testing strategy,[26] also using UML state models, focus on the testing needs of object-oriented implementations. In this strategy, the test cases aim at covering round-trip and sneak paths of a specification given in a state diagram. The sneak paths test illegal or non explicitly-defined transitions, whereas 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.

VII. Conclusion

This paper presented the testing process named CoFI (Conformance and Fault-Injection) which was conceived firstly to improve the validation of the set of services stated in ECSS-E-70-41A. This standard specifies the services for application-process communication between ground and on-board systems, such as, telecommand verification, on-board scheduling operations, on-board monitoring, storage and retrieval, memory and time management, event reporting, etc.

The CoFI is based on the conformance test process standardized in IS-9646 and includes testing activities with software-implemented fault injection (SWIFI) to improve the evaluation power of the conformance testing.

In order to provide automation for the testing process, the CoFI includes a test and fault case generation approach based on UML diagram transformation. From these transformations a reusable abstract test suite is created, starting from the standardized specification given in a textual notation. The approach presented here assures not only efficiency of the testing activities, but also improves reliability in the space application software products. 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 *telecommand verification* service, defined in the ECSS-E-70-41A space software standard prepared by the European Space Agency, was used to illustrate an abstract test case and a fault case derived from the referred specification.

Acknowledgments

The authors would like to thank the researchers from the European Space Agency that motivated us to continue research on space software validation. Thanks are also due to the anonymous referees for their valuable comments.

References

[1]Mazza, C., "Standards: the Foundations for Space IT," presentation in the Workshop: Space Information Technology in the 21st Century, Sep. 27, 2000, Darmstadt, Germany. URL: www.esoc.esa.de/pr/documents/workshops/it_2000/it_in_future/ESA_C_Mazza.ppt [cited 2 Dec. 2004].

[2]European Cooperation for Space Standardization (ECSS) http://www.ecss.nl/.

[3]European Space Agency (ESA), European Cooperation for Space Standardization (ECSS) – Space Engineering – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization, ECSS-E-70-41A, Jan. 2003.

[4]Merri, M., Melton, B., Valera, S., Parkes, A., "The ECSS Packet Utilization Standard and Its Support Tool," *Proceedings of AIAA International Conference on Space Operations*, 2002. URL: www.spaceops2002.org/papers/SpaceOps02-P-T5-06.pdf [cited 13 Dec. 2002].

[5]Merri, M, Rüting, J., Schurman, P., "Validation of the ESA Packet Utilization Standard by Object-oriented Analysis," *Proceedings of International Conference on Space Operations*, 1996.

[6]Consultative Committee for Space Data Systems (CCSDS), CCSDS-201.0-B-3 to CCSDS-203.0-B-2 – Telecommand System. URL: htpp://www.ccsds.org [cited 13 Dec. 2004].

[7]ISO/TC 97/WG1 IS 7498. Basic Reference Model for Open System Interconnection. 1983.

[8]Information Technology – International Organization for Standardization/International Electrotechnical Commission – ISO/IEC 9646 – Conformance Testing Methodology and Framework, International Standard IS-9646. ISO, Geneva, 1991. Also in: ITU-T Recommendations X.290 to X.296.

[9]Baumgarten, B., Giessler, A., *OSI Conformance Testing Methodology and TTCN*, Elsevier, Amsterdam, 1994.

[10]Consultative Committee for Space Data Systems (CCSDS) – SPACE LINK EXTENSION SERVICES – CCSDS 910.0-Y-1 YELLOW BOOK, April 2002.

[11]Mertens, M., “Advanced Protocol Testing Methods and Tools,” *Proceedings of AIAA International Conference on Space Operations*, 2002.

[12]Voas, J. M., McGraw, G., *Software Fault Injection: Inoculating Programs against Errors*, John Wiley & Sons, Inc., New York, 1998.

[13]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*, Vol. 16, No. 2, 1990, pp. 166–182.

[14]Madeira, H., Some, R. R., Moreira, F., Costa, D., Rennels, D., “Experimental evaluation of a COTS system for space applications,” *Proceedings of IEEE International Conference on Dependable Systems and Networks* (DSN), 23–26 June 2002, Washington, D.C., USA. URL: http://computer.org/proceedings/dsn/1597/15970325abs.htm. [cited 02 Feb. 2004].

[15]Pressman, R. S., *Software Engineering: A Practioner’s Approach*, 4th ed., McGraw-Hill Companies, Inc., New York, 1982.

[16]Ambrosio, A. M., Martins, E., Vijaykumar, N. L., Carvalho, S. V., “Systematic Generation of Test and Fault Cases for Space Application Validation,” *Proceedings of the 9th ESA Data System in Aerospace* (DASIA), 30 Mai – 2 Jun. 2005, Edinburgh, Scotland. Noordwijk: ESA Publications, 2005. Papers on Disc [CD-ROM]

[17]Ambrosio, A. M., Martins E., Vijaykumar N. L., Carvalho, S. V., “A Methodology for Designing Fault Injection Experiments as an Addition to Communication Systems Conformance Testing,” *Proceedings of the First Workshop on Dependable Software – Tools and Methods in the IEEE Conference on Dependable System and Network*, 28 June – 1 July 2005, Yokohama, Japan.

[18]Cavali, A. R., Favreau, J. P., Phalippou, M., “Standardization of formal methods in conformance testing of communication protocols,” *Computer Networks and ISDN Systems*, No. 29, 1996, pp. 3–14.

[19]Martins, E., Ambrosio, A. M., Mattiello-Francisco M. F., “ATIFS: a testing toolset with software fault injection,” *Proceedings of the Workshop of Software Test* (SofTest). University of York: 4–5 Sep. 2003.

[20]Zeng, H. X., Rayner, D., “The impact of the Ferry concept on protocol testing,” *Proceedings of the 5th International Conference on Protocol Specification, Testing and Verification*, 10–13 June 1985, Toulouse-Moissac, France. North-Holland: IFIP WG6.1. 1985. pp. 519–531.

[21]Chanson, S. T., Lee, B. P., Parakh, N. J., Zeng H. X., “Design and Implementation of a Ferry Clip Test System,” *Proceedings of the 9th IFIP Symposium on Protocol Specification Testing & Verification*, Enscchede, The Netherlands, 1989, pp. 101–118.

[22]Martins, E., Mattiello-Francisco, M. F., “A Tool for Fault Injection and Conformance Testing of Distributed Systems,” *Proceedings of the 1st. Latin-American Dependable Computing Symposium*, Lecture Notes in Computer Science, Springer Verlag, Sep. 2003, pp. 282–302.

[23]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,” *Proceeding of AIAA International Conference on Space Operations*, 2004.

[24]European Space Agency (ESA) – European Cooperation for Space Standardization (ECSS) – Space Engineering – Procedure Language for User in Test and Operations (PLUTO), ECSS-E_70–32, Issue Draft 5, 2001.

[25]Larson, W., Vertens, J., *Space Mission Analysis and Design*, 2nd ed., Kluwer Academic Publisher, Dordrecht, 1992.

[26]Binder, R., *Testing Object-Oriented Systems: Models, Patterns and Tools*, Addison-Wesley, Boston, 2000.

[27]Ryser, J., Glinz, M., “SCENT: a method employing scenarios to systematically derive test cases for system test,” Zürich: Institut für Informatik, Universität Zürich,. Technical report 2000/03, 2000.

[28]Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-oriented modeling and design*, Prentice Hall International, New York, 1991.

[29]Ural, H., “Formal methods for test sequence generation,” *Computer Communication*, Vol. 15, No. 5, 1992, pp. 311–325.

[30]Lee, D., Yannakakis, M., “Principles and Methods of Testing Finite State Machines: a survey,” *Proceedings of IEEE*, Vol. 84, No. 8, 1996, pp. 1090–1123.

[31]Grabowski, J., “SDL and MSC Based Test Case Generation – An Overall View of the SAMSTAG Method,” University of Berne, IAM-94-0005, 1994.

[32]Koné, O., “An Interoperability Testing Approach to Wireless Application Protocols,” *Journal of Universal Computer Science*, Vol. 9, No. 10, 2003, pp. 1220–1243.