ATIFS: a testing toolset with software fault injection

Eliane Martins[1]
Ana Maria Ambrosio [2]
Maria de Fátima Mattiello-Francisco[2]

[1]Institute of Computing (IC)
State University of Campinas (UNICAMP)
eliane@ic.unicamp.br

[2]Ground Systems Division (DSS)
National Institute for Space Research (INPE)
Av. Dos Astronautas, 1758 - Sao Jose dos Campos -12227-010 - SP - Brazil
FAX: +55-12-345-6625,
Ana@dss.inpe.br fatima@dss.inpe.br

ABSTRACT

This paper describes the ATIFS, a testing toolset which supports the activities of black-box tests for reactive systems, especially communication systems. In ATIFS, two types of testing are carried out: conformance testing and software fault injection. These testing types allow one to answer such questions about the system under test as: "does the system perform what is specified?", as well as "for how long does the system perform what is specified?" and "how does the system behave in the presence of faults in its environment?". This toolset was conceived and implemented aiming at providing a user with facilities for the activities of test case derivation, test execution and test result analysis. The general requirements that guided the ATIFS development, its architecture and an overview of the already implemented tools are focused on in this paper. The main tools were successfully used in the conformance tests of a real space application: telemetry reception software for real time communication with a balloon experiment developed at INPE. The test process using the ATIFS toolset in a space application as a case study was an important experience to deal with the constraints imposed by both the application test requirements and the tool prototypes.

Keywords: test automation, conformance test, fault injection, formal methods.

1. INTRODUCTION

Software testing is an expensive activity. It may consume from 50 to 75 percent of the development effort of a system [TB99]. Reactive systems, in particular, are more difficult to test because they generally possess distributed and concurrent features. These systems, found in the real world at air and train control centers, in telecommunication applications, in space systems, just to name a few, need extensive verification and validation to give confidence that they will be able to perform the critical functions. Testing is by far the most common verification and validation activity. So, reactive systems must be extensively tested. Due to the increased complexity of these systems, the test activity done manually is a hard task and prone to errors. Consequently the high quality level required for these kind of critical systems operation is hard to achieve.

Much effort has been put in to building testing tools. One may find commercial and academic testing tools which support different tasks related to the tests. Pressman [Pre97] classifies testing tools according to their functionality in the following categories: (i) data acquisition, (ii) static analysis of the code, (iii) dynamic analysis for code coverage, (iv) simulators that may replace part of the system and (v) management for planning, development and control of the tests.

There are few commercial tools to help the user with test case generation, especially those tests based on the system specification. This is because the specification generally is in natural language, being unable to be processed by a tool. To solve this problem, one has to write a specification using formal methods. The formal methods allow us to represent the system in a notation with a well-defined syntax and semantics; consequently the specification will be more precise than those written in natural language, being thus suitable to be processed by tools.

Formal methods are especially useful for tests. They allow automating both test case generation from the specification model and the analysis of system outputs produced during test execution. Many different formal notations exist for reactive systems. ATIFS adopted Finite State Machines, classical and extended, as the specification model, as these notations are frequently used to represent the behavior of reactive systems. ATIFS focuses on using formal methods for conformance testing.

Conformance tests aim at answering the question: “does the implementation realize the specified functionality?” The answer to this question is a first step to good quality for a system. However, it should not be the only question to be answered. For critical systems, as those above cited, it is necessary to answer questions like: (i) How long does the system continue to realize the required function? (ii) How does the system react whenever faced with unwanted and invalid behavior of the environment? For answering the last question, ATIFS uses the technique of fault injection. By testing a system in the presence of faults, either internal (introduced during development) or external (originating in the system’s environment), this technique is a useful complement to conformance testing. An overview of these types of testing is presented in Section 2.

ATIFS comprises seven tools: AnaLEP, VerProp, ConDado, SeDados, GerScript, FSoFIST, Antrex, which help the tester respectively in the following combined test activities (i) formal specification semantic and syntactic analysis, (ii) specification properties verification, (iii) test cases derivation from a formal specification for conformance test purposes, (iv) data selection, (v) editing of the automatically generated test cases and fault selection, in the case of the fault injection option, into a executable script, (vi) controlled test execution supported by a distributed test architecture, (vii) test results analysis and diagnosis generation after the test execution. The conceptual aspects of these tools are further described in section 3.

ATIFS is being developed as a cooperative project between the Computer Institute of Campinas University (UNICAMP) and the National Institute for Space Research (INPE). One of the objectives of the project is to provide INPE with a set of tools that will improve the quality of the space software systems actually developed in house by INPE. Another objective is to provide an open toolset that implements various testing techniques.

Section 4 presents the use of ATIFS in the test of a telemetry reception system developed for a balloon experiment at INPE. Section 5 presents some related work. Finally, section 6 discusses some lessons learned and suggests further research directions.

2. TYPES OF TESTING SUPPORTED BY ATIFS

Testing is the process of exercising a system aiming at revealing the presence of faults. A *fault* (or bug) is a mistake made by developers as the system development goes on. For example, a non-initialized variable in the code is a fault. An *error* is an activation of a fault. When the system comprising a non-initialized variable is executed, the use of this variable may cause wrong values in other variables, leading the system to a wrong state. The errors may be propagated to the system interfaces, thus constituting a failure. A *failure* is the manifestation of the system’s inability to execute the service it is supposed to do. A failure is perceived as wrong output values, by system abnormal termination, or by inability to fulfil time and/or space constraints [Bin00,ch3].

To reveal faults, the system undergoes a combination of inputs during testing. Based on the observable output a verdict may be given, indicating whether the tests have passed or failed. So faults are revealed whenever a failure occurs. To determine failure occurrence, it is necessary to have a trusted systems specification as a reference. The specification supports an oracle, that is the mechanism used to foresee the outputs that should be produced for the system [Bei95, ch1].

The term testing used in this text comprises the verification and validation activity of exercising an implementation with a set of pre-selected inputs and of observing its outputs. Test scope will depend on whether the implementation under test (IUT) corresponds to part of or the full system.

Tests are also classified according to the way the inputs are derived. Commonly used approaches are implementation- and specification-based. In implementation-based or white-box testing, test cases are derived from code analysis. In specification-based or black-box testing, test cases are derived from the system specification (or architecture). Grey-box testing lies in-between both, in which the structure of the system under test is known, but not the code of each of its elements.

2.1. CONFORMANCE TESTS

Conformance tests aim at determining if an IUT meets its specification [Hol91]. Conformance testing is essentially black-box tests, in which the only observable aspect of the IUT is the external input and output.

In the telecommunication area, an effort has been undertaken to standardize the conformance testing of protocols of the Open Systems Interconnection (OSI) Reference Model. The standard ISO 9646 “OSI Conformance Testing Methodology and Framework (CTMF)” (1991) defines a methodology, establishes frameworks and defines procedures for conformance testing. The purpose is to improve the capability of both comparing and reproducing the tests results performed by different groups. The standard does not establish the way that the tests should be generated, rather, it defines a framework for structuring and specifying the tests.

The CTMF also defines conceptual architectures to support test execution. According to this standard, the test architectures (also named test methods) should be based on the location of the Points of Control and Observation (PCO). The specification of a protocol of the OSI reference model describes the behavior of an entity in terms of the inputs and the outputs passing by the upper and lower service interfaces, respectively named (N)-SAP[1] and (N-1)-SAP. Ideally, each SAP is a PCO that is directly used by the testers to communicate with the IUT. But generally one or more SAPs are not directly accessible during testing. The conceptual test architecture is illustrated in Figure 2.1.

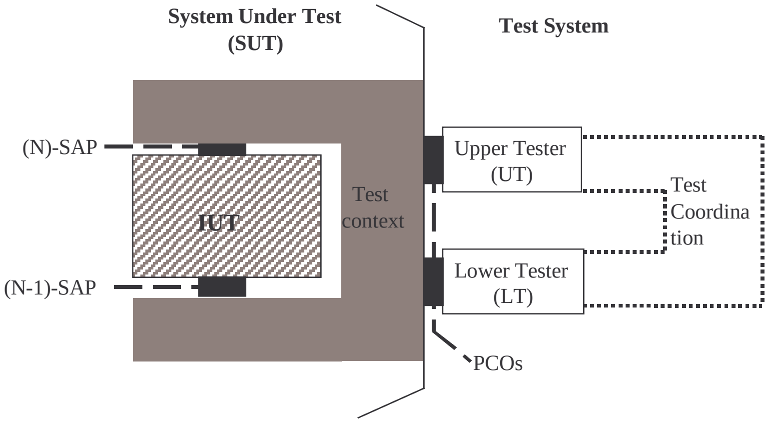


Figure 2.1. Conceptual test architecture

The test architecture defines the IUT accessibility model. It might be described in terms of [TB99]: (i) the accessible PCOs, (ii) test context, that is, the environment in which the IUT is embedded and that is presented during testing; (iii) testers – associated with each PCO, named upper tester (UT) and

---

[1] Service Access Point

lower tester (LT) respectively connected to (N)-SAP and (N-1)-SAP. Whenever both testers are used, coordination is required.

In the process of conformance testing we identify three main phases: test generation, test execution and test results evaluation. Since in conformance testing the IUT is considered as a black-box, test generation as well as test results evaluation should be based on the specification. Because the number of input combinations a complex system may accept is large, or even infinite, it is worth having tools to support these activities, in order to guarantee the required quality level. In CTMF, only test execution may be automated, because the specification may be informally described. An effort has been undertaken to define a formal framework for conformance testing based on formally specified protocols. In this way, test case generation can be automated, in that test inputs can be derived algorithmically from a formal specification, which can also be used for automatic test results evaluation. There has been effort on standardizing the use of formal methods in conformance testing [CFP96]. As an example of this initiative we can mention the work in [TB99], which presents the formalization of conformance testing based on Labeled Transition Systems (LTS) by defining an equivalence relation between the specification and the implementation. In ATIFS, conformance testing is based on a formal model of the system in the form of (Extended) Finite State Machines ((E)FSM).

2.2. FAULT INJECTION

Fault injection consists of the deliberate insertion of faults or errors into a system aiming at observing its behavior. This technique is very useful to validate the implementation of error recovery and exception mechanisms, as well as to determine the system behavior in the presence of environment faults.

There are several approaches for fault injection [HTI97]. Our work addresses fault injection by software, which causes changes in the state of the system under test, under the control of software. In this way both hardware and software failure modes might be emulated. The mechanism consists of interrupting the IUT execution to run the fault injector code. The latter can be implemented in various forms: as a routine started by a high priority interruption, a routine started by a trace mechanism, or as an extra code inserted into the IUT or in its context (operating system, underlying communication layer, for example).

The fault injector implemented in ATIFS aims to mimic communication faults which represent typical faults of distributed systems, like message loss, duplication, corruption and delay. Communication fault injectors are generally inserted between the IUT and the underlying service [DJM96], [EL92], [RS93], [SW97]. The fault injection mechanism implemented in the FSoFIST tool is described in section 3.

3. ATIFS DESCRIPTION

3.1. REQUIREMENTS

The ATIFS project had its initial conception in the beginning of the nineties. Since the beginning it was thought of as a set of integrated tools having a common standard graphical interface and a common data base to handle the data produced in each phase of the test process. The foundational requirements that oriented the ATIFS were: (i) the use of a formal specification, FSM or EFSM, from which to derive the test cases and analyze the test results; (ii) the support to several test activities like generation, implementation, execution, and analysis; (iii) portability (Unix/Linux and Windows operating systems); (iv) user interface homogeneity; (v) extensibility, allowing new tools to be easily aggregated; (vi) be as much as possible independent of the implementation under test.

3.2. ARCHITECTURE

The ATIFS architecture, shown in Figure 3.1, comprises the following tools: AnaLEP (analyzer of a specification written in LEP (from Protocol Specification Language, in Portuguese)); VerProp (FSM properties verifier); SeDados (Data Selector); ConDado (Data and Control test case generator), GerScript (Script Generator), FSoFist (Fault Injector), AnTrEx (Trace Analyser).

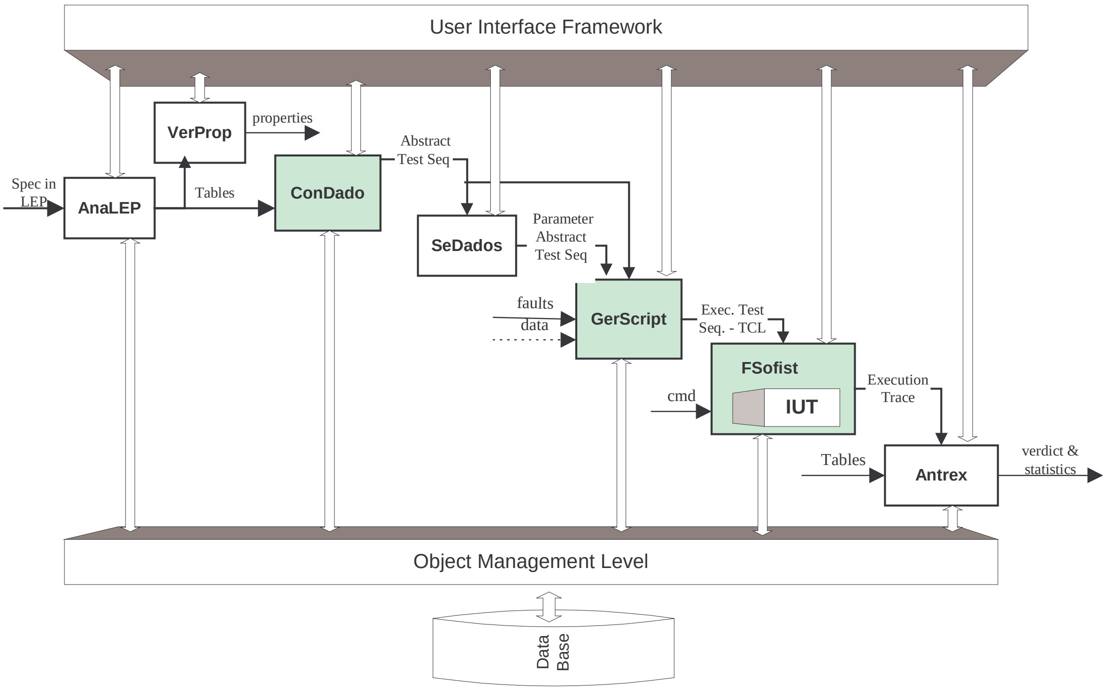


Figure 3.1. ATIFS architecture

In order to completely integrate the tools, an infrastructure was designed comprising: (i) a user interface framework, which facilitates and standardizes the interface of all tools [Gui96]; (ii) a data base shell to manage persistent objects. The four main tools, already implemented and validated, are described below.

3.3. ConDado

ConDado is a tool for automatic test case generation [Sab99]. The input to ConDado is a specification in the form of an FSM or EFSM. This specification can be described textually using the LEP notation. Currently, the tool has been enhanced with interface which allows a graphical representation of the specification. The test case generation is static (tests are generated and may be checked before execution) instead of dynamic (or on-the-fly, where the tests are generated during the test execution). The main feature of this tool is that it combines test generation for the control aspect of a specification (FSM model) and also for the data part (EFSM model). The output is an abstract test sequence (named according to ISO 9646 standard [ISO91]) in an IUT-independent notation. In the following subsections we briefly describe the main features of Condado; further details can be found in [MSA99, Sab99].

3.3.1. Test generation: control aspect

In order to generate test cases for covering the control aspect of the specification and for detecting output faults, ConDado implements a variation of the TT (transition tour) method. The algorithm searches the FSM, supposed to be strongly connected and deterministic, to traverse circuits starting and ending at the initial state. Each circuit represents a scenario of use of the system and is exercised by a test case. Although the TT method is not able to detect transition faults (e. g. if the IUT goes to a wrong state) the diagnostic algorithm implemented in Antrex addresses this aspect.

3.3.2. Test generation: data aspect

The data aspects considered by ConDado are relative to the format and parameter values of input interactions. These aspects are described in LEP using a syntax that is based on ASN.1[2].

Two testing techniques are used to generate test data: syntax testing and equivalence partition testing. Syntax testing [Bei90, ch. 5] is used here to produce valid formats of input interactions. The equivalence partition technique is used to produce valid data for input parameters. Although both testing techniques require generation of invalid inputs, this is not implemented to avoid increasing test sequence size. Fault injection may be used to cover these aspects.

For EFSM specifications, data aspects comprise also variables and predicates. The latter are associated with transitions and are conditions that must be satisfied for the transition to be fired when the input interaction occurs. Predicates are expressed in terms of the variables and parameters of input interactions. So, to satisfy a given test purpose (in other terms, to exercise a given transition circuit), data values should be selected so that all transition predicates in the circuit are satisfied.

ConDado does not address this issue, as data values are generated without taking the predicates into account. For the moment the user has two options: (i) to unfold the EFSM, obtaining the corresponding FSM without predicates (reference [CS87] has a good presentation on this subject) (ii) to change data values manually. The SeDados was planned to deal with this issue [Ube01].

An example of an input specification in LEP as well as the output generated by Condado is presented in 4.2.

3.4. GerScript

The GerScript tool transforms a test suite generated by ConDado, named here an Abstract Test Suite (in analogy to ISO 9646 standard [ISO91]), in an executable format [Jeu99]. The output of this tool is a test script in the Tool Command language (TCL). This notation was chosen because at the time the tools were developed, it was a popular interpreted language, used in some fault injection tools (e.g, [DJM96]). Besides, TCL syntax is quite similar to C, which makes it easier for the user to create his own test cases without the burden of learning a new language. Moreover, interpreters for this language are freely available and can be easily incorporated into C or C++ programs. The script, consisting of the Executable Test Suite, may be organized in groups [ISO91].

For fault injection tests it is also possible for the user to define the faults to be injected. A fault is characterized by a set of attributes: (i) fault model, which can be one of the following: message omission, corruption, duplication or delay; (ii) a mask, used for message corruption, which defines the value to be used to alter message contents, (iii) repetition pattern, which indicates whether a fault is to be transient (inserted only once), intermittent (inserted periodically) or permanent (inserted in all messages); (iv) fault location, that indicates which part of the message is to be corrupted; and (v) fault start, that indicates how many messages should be transferred between the tester and the IUT before fault injection starts.

3.5. FSoFIST

FSoFIST stands for Ferry-clip with Software Fault Injection Support Tool [Ara00]. This tool aims at supporting the test execution. It presents a user interface allowing the tester to control and monitor the tests step by step. It has facilities to start, stop and continue a test session any time. It also provides a framework designed according to the ferry clip architecture that allows conformance testing of

---

[2] Abstract Syntax Notation One -

different protocol implementations. This architecture has been extended here, to support the injection of faults with minimum intrusion in to the system under test.

The ferry-clip architecture [ZR86], [ZLD+88], [CLP+89], proposed for protocol testing, presents the following advantages: (i) it is intended to implement the different standard test architectures, (ii) it supports the execution of various types of tests beyond conformance [CVD92], (iii) it presents a high degree of portability and modularity, and (iv) it presents a low degree of intrusion in the IUT.

Figure 3.3 illustrates the distributed architecture of the FSoFIST. The Active Ferry (AF) and the Passive Ferry (PF) are the main elements of the ferry architecture. They are responsible for the test data transfer according to a simplified ferry protocol. Additionally, they adapt the data into the format understandable to the IUT, so the Test Sequence Controller is not modified for each new IUT.

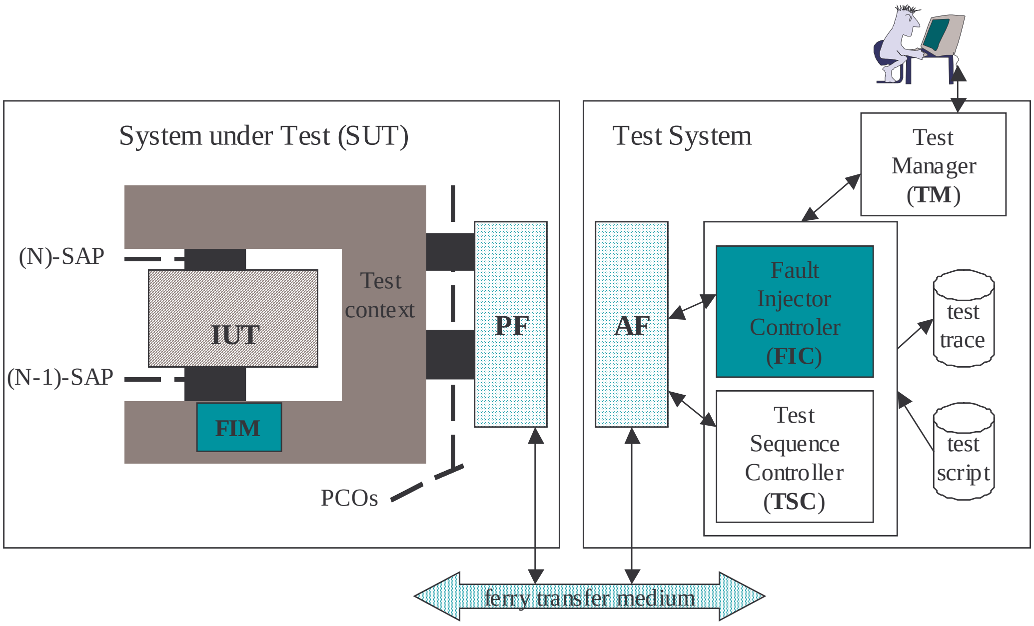


Figure 3.3. FSoFIST architecture

As shown in figure 3.3, the test data is transported from the Test System (responsible for test coordination) to the System Under Test by the *ferry transfer medium*, which is an IUT-independent communication channel.

The test manager (TM) starts and stops a test session automatically or under a user command. It provides a user interface allowing the user to follow the executed test steps and to enter information before the test execution. It activates and deactivates the Test System components.

The Test Sequence Controller (TSC) applies a test sequence to the IUT, using the ferry clip protocol according to the available PCOs.

The Fault Injection Controller (FIC) controls fault injection testing according to the script. It sends information about the faults to be injected to the FIM (Fault Injection Module) and stores data collected by the latter. The FIM resides in the test context. It intercepts the messages received by the IUT, and inserts the faults determined by the FIC.

FSoFIST was developed under the Solaris 2.5 operating system, where it was first used. It was later ported to Linux. Its PF component ran initially in Solaris but was transferred to MS Windows for the case study presented in section 4.. The AF is written in C++ and the PF in Perl. The communication between them uses the socket library, which facilitates the component extension to the case of testing several IUTs.

An example of input to FSoFIST as well as the output log can be seen in 4.2.

3.6. AnTrEx

This tool implements an oracle, a mechanism that analyses whether or not the observed outputs conform to what is specified. Analysis is performed on an execution trace that is comprised of the observed interactions (viz. system inputs and outputs) [Ste97].

The specification model (used for test case generation) is also used for trace analysis. As a result, a verdict of pass or fail is emitted for each test case. For those with a verdict of fail, a diagnostic is provided with the probable cause of the failure.

4. CASE STUDIES

This section presents some examples using the ATIFS tools. These examples, as well as the experiments performed in a case study described in 4.2 were aimed at validating the main tools of ATIFS.

4.1. Initial experiments

To validate the test approach implemented by Condado we used the specification of the transfer layer of the CCSDS[3] protocol stack, used in the Telecommand Station of the SACI-1 project [Car97]. The protocol was specified as an FSM possessing 6 states, 46 inputs and 235 transitions. The following table shows the CPU times as well as the number of test cases generated when varying the number of transitions of the original specification. Nine different machines were generated, designated M1 to M9, where M9 is the original FSM. Table 4.1 shows some results obtained. Further results are presented in [MSA99].

Table 4.1. Some test generation results using ConDado.

Model	# Transitions	Constraints	CPU time (sec)	# Test cases
M1	220	—	1.74	324
M2	221	—	2.21	406
M2	221	Cover only one transition	1.59	82
M3	222	—	4.85	689
M3	222	Cover only one transition	4.06	283

The experiment was performed on a Sun SPARC station under Unix. The results in Table 4.1 show that the use of constraints can considerably reduce the number of test cases, but the same does not hold for the CPU time. This is because the test case generation algorithm is implemented in Prolog, which first generates a test case and then checks whether it satisfies the constraint; if not, the test case is discarded. We envisage the implementation using a C/C++, not only to improve the performance of the tool but also to allow other coverage criteria that allow for a generation of shorter test sequences than the one currently implemented.

4.2.Telemetry Reception System

The Telemetry Reception Software (TMSTATION) of the MASCO Telescope implements a ground entity of the ground-board communication protocol, which receives in real time the data from X-ray sky imaging got from the MASCO telescope (a Brazilian space project) [VBF+01]. The data is acquired during approximately 30 minutes during the telescope’s flight on board a balloon. The experiment was developed by the Astrophysics Division at the National Institute of Space Research (INPE), and will be launched in 2003. The imaging data acquired by a hardware detector is organised into frames, stored in files on board, and transmitted in real time to a ground station, where it is received by the TMSTATION software [Mat00].

---

[3] Consultative Committee for Space Data Systems.

The main function of TMSTATION is to separate the frames, sequentially received through a serial channel (RS-422 interface), in distinct files, in exactly the same way they are stored on board. The separation is based on the identification of a pattern that consists of a string of at least 5 hexadecimal words (**aa55H**) consecutively, which represents the end-of-frame pattern. The TMSTATION behaviour was formally specified in a FSM. It was implemented in C under the LabWindow/ CVI environment for Windows [LWC96]. We present the partial results obtained up to now.

4.2.1 Conformance testing

Test case generation for conformance testing of the TMSTATION software was based on the FSM model presented in Figure 4.1. To simplify matters, we considered that the end of frame pattern is a sequence of exactly five times the hexadecimal string **aa55**. The specification considers the situations of missing data during the balloon flight transmission to ground. Because of this missing data the TMSTATION software may separate files, on ground, in a different way from the data stored on board. In this way, frames may be truncated (missing words in the data field) or extended (missing the end-of-frame pattern, causing two frames to be considered as one). In the model presented in figure 4.1:

- the input interaction **Naa55** represents a particular string composed of 50 characters not comprising any **aa55H** word;
- the states P1, P2, P3 and P4 recognize the end-of-frame pattern;
- we considered only one transition back to the RAW DATA state from P3, instead of one from each of P1, P2 and P4. This transition is enough to represent the break in the **aa55H** sequence that ends the frame.

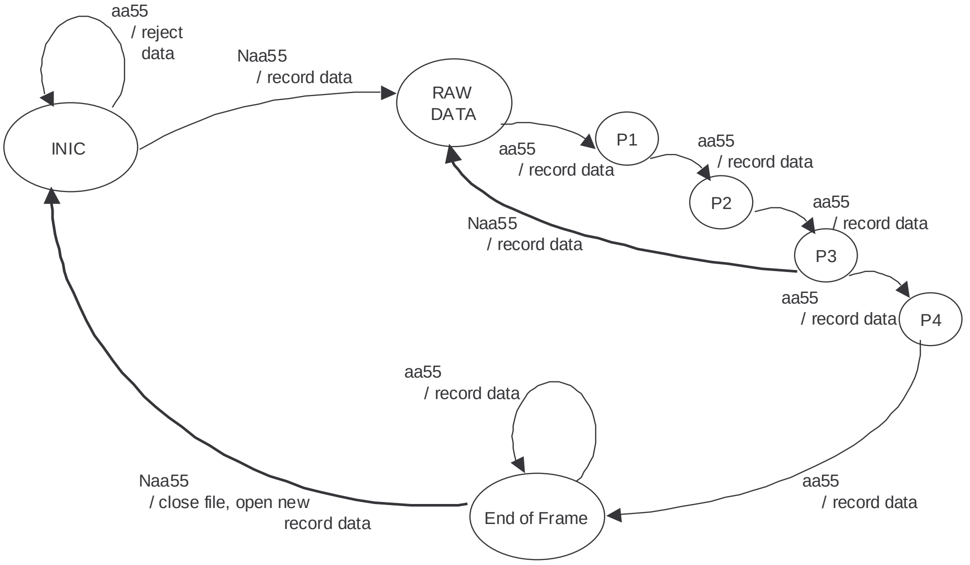


Figure 4.1: TMSTATION behaviour in FSM

Figure 4.2(a) presents the specification in LEP, whereas in part (b) has a list of the test cases generated. Both are partially shown, for sake of brevity.

```
STATES:
#inic;  -- the “#” identifies the initial state
raw_data;
p1;
p2;
...
p4;  -- waits last aa55h of the end-of-frame pattern
end_of_frame;
```

```
-- 1. test cases - invalid data
    senddata(U,aa55)  recdata(U,RejectData)

-- 2. test case – normal or truncated data field with one extra
                  standard
   senddata(U,Naa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
```

```
INPUTS:
Na55;
aa55;

OUTPUTS:
Reject_data;
Record_data;
Close_file;
OpenNewFile

...
-- Transitions are described according to the
-- following format:
-- *<transition id> “>”<current state>
-- “?”<input event> “!”<output>  “<”<next state>

TRANSITIONS:
*t0 >inic ?Naa55 !Record_data < raw_data ;
*t1 >raw_data ?Naa55 !Record_data < raw_data;
*t2 > raw_data ?aa55 ! Record_data <p1;
...

*t9 >end_of_frame ?aa55 !Record_data <inic;
*t10 >inic ?aa55 !Reject_data <inic;
```

```
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,Naa55)  recdata(U,CloseFile OpenNewFile
   RecordData)

-- 3. test case- normal or truncated frame
   senddata(U,Naa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,Naa55)  recdata(U,CloseFile OpenNewFile
   RecordData)

-- 4.testcase - frame with end-of-frame occurrence into the data
                field and one extra (truncated frame)
   senddata(U,Naa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,Naa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,aa55)  recdata(U,RecordData)
   senddata(U,Naa55)  recdata(U,CloseFile OpenNewFile
RecordData)
```

Figure 4.2: (a) Specification of TMSTATION software in LEP. (b) TMSTATION test cases generated by ConDado

With the FSM simplification described above, the test cases automatically generated by ConDado were quite representative. We divided the test cases according to their coverage into the following situations:

- **normal frame reception** (no data missing): test cases 2 and 3, respectively. In these test cases the parameter value for the input **Naa55** is a data field string that is 50 words long and does not contain **aa55**H.
- **truncated frame reception** (some words missing in the data field): test cases 2 and 3, as above, but the parameter value for the input **Naa55** is a data field string that is 25 words long.
- **extended frame reception** (some words missing in the end-of-frame pattern): test case 4, in which the parameter value for the input **Naa55** is any data field string composed of at least one non-aa55 word. We have used 10 words in the test execution.

4.2.2 Test Configuration

FSoFIST was configured for the tests according to figure 4.3. Only the PF component was modified in order to integrate with the new IUT. This first module version was on Linux, and here it was ported to Windows. The physical serial channel connects the two processes, PF and IUT, both residing in the same machine, in Windows environment. The Lab-CVI library was used for the communication through the serial channel, acting then as the test context.

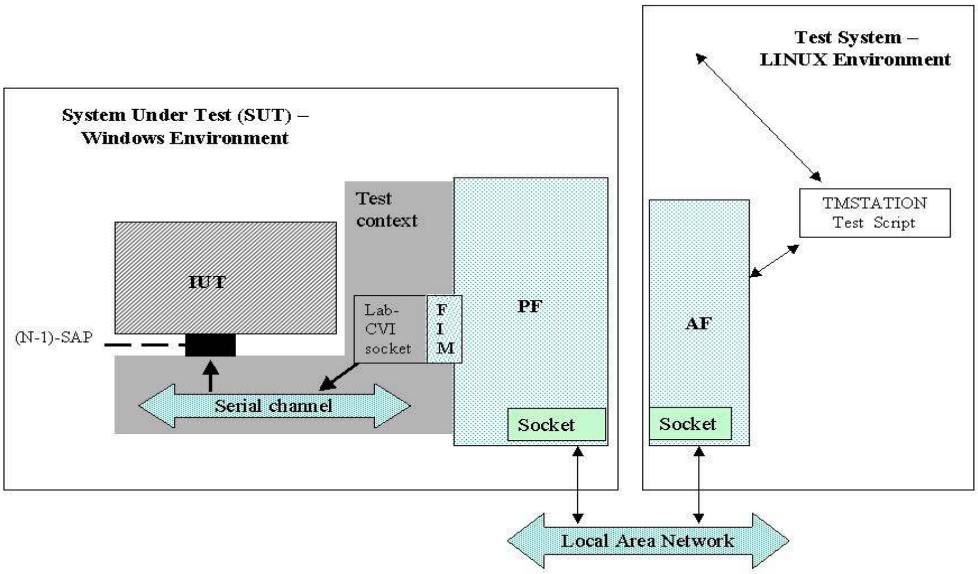


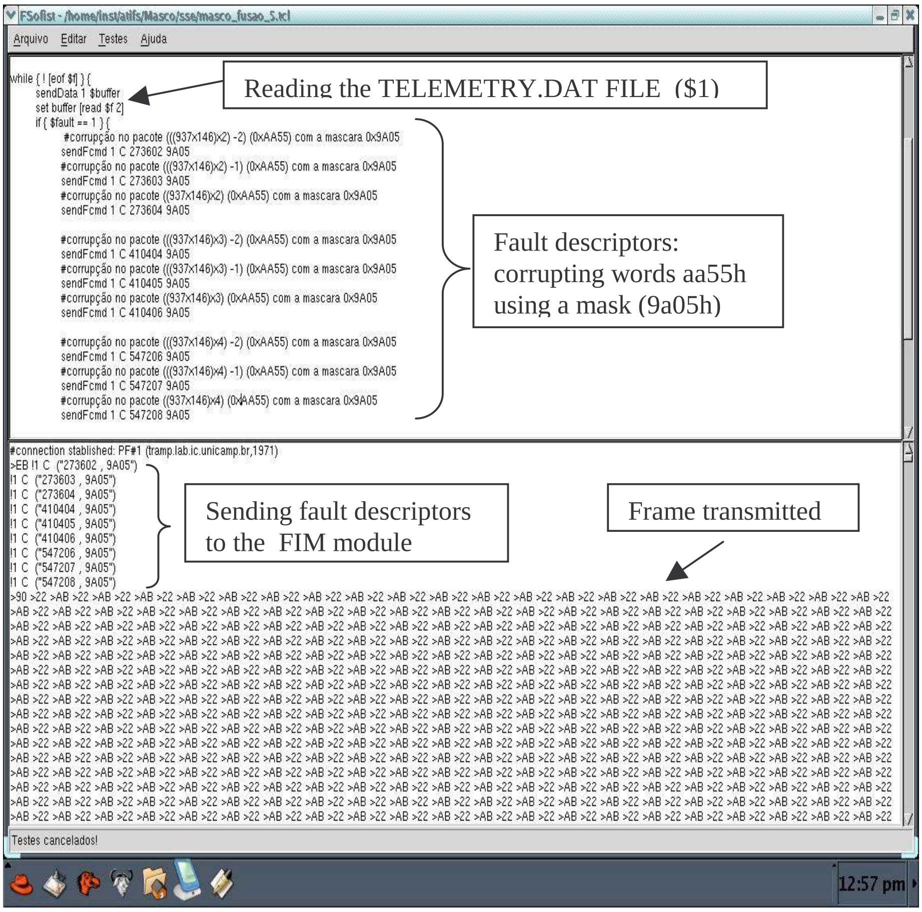
Figure 4.3. FSoFIST configuration for TMSTATION application

4.2.3 Fault Injection

We also performed fault injection experiments. The configuration used was quite similar to that presented in figure 4.3. In this architecture we introduced the FIM module inside the PF. Faults are injected to emulate ground-board link failures. For that reason, we assumed that messages transferred to the IUT through the serial channel are delivered in sequence, without modification. So, the FIM was not introduced inside the test context, as it should be; instead, it uses this context to interface with the IUT. This configuration was useful given that our main interest was solely to validate FSoFIST module's behavior.

Given that wrong frames are now generated through fault injection, we did not use the script produced for conformance testing. Instead, test inputs were obtained from a TELEMETRY.dat file, containing only a sequence of valid frames. This file contains the output data generated by the onboard system. For fault injection tests, the scripts were generated according to the fault model specified. Faults were selected deterministically to meet the frame violations presented in 4.2.1.

Figure 4.4 shows an example of FSoFIST window during test execution. The upper part of this window shows the script used to inject faults. In this script, the last three words of the end-of-frame pattern are changed to 9A05, intermittently on the 2nd, 3rd and 4th frames of a sequence composed of five frames. The lower part of Figure 4.4 shows partially the log generated during the tests.



```
while { ! [eof $f] } {
        sendData 1 $buffer
        set buffer [read $f 2]
        if { $fault == 1 } {
                #corrupção no pacote (((937x146)x2) -2) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 273602 9A05
                #corrupção no pacote (((937x146)x2) -1) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 273603 9A05
                #corrupção no pacote ((937x146)x2) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 273604 9A05

                #corrupção no pacote (((937x146)x3) -2) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 410404 9A05
                #corrupção no pacote (((937x146)x3) -1) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 410405 9A05
                #corrupção no pacote ((937x146)x3) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 410406 9A05

                #corrupção no pacote (((937x146)x4) -2) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 547206 9A05
                #corrupção no pacote (((937x146)x4) -1) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 547207 9A05
                #corrupção no pacote ((937x146)x4) (0xAA55) com a mascara 0x9A05
                sendFcmd 1 C 547208 9A05
```

```
#connection stablished: PF#1 (tramp.lab.ic.unicamp.br,1971)
>EB !1 C  ("273602 , 9A05")
!1 C  ("273603 , 9A05")
!1 C  ("273604 , 9A05")
!1 C  ("410404 , 9A05")
!1 C  ("410405 , 9A05")
!1 C  ("410406 , 9A05")
!1 C  ("547206 , 9A05")
!1 C  ("547207 , 9A05")
!1 C  ("547208 , 9A05")
>90 >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
>AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22 >AB >22
```

```
Testes cancelados!
```

Figure 4.4. FSoFIST graphical interface showing a fault injection script and corresponding log.

We are currently using the tools to validate another version of the TMSTATION software that checks the status of the frames (truncated, extended, normal) based on the contents of their sub-frames[4]. This specification has 8 states, 15 inputs and a total of 22 transitions. The number of test cases in the complete test suite derived by Condado was 4,742.

We are also testing the Conference Protocol, presented in [TB99]. An FSM model was generated based on the specification[5]. This protocol provides a multicast service, like a “chat-box”, to users participating in a conference. For a conference with three partners (A, B and C) we obtained a partial model with 5 states and 17 transitions, representing the different ways to join and leave a conference. A total of 10,429 test cases were derived for this specification.

Of course, constraints offered by Condado are being used in both cases to obtain practically useful test sequences.

5. RELATED WORKS

Numerous approaches have been proposed for the validation of reactive systems. Our work is primarily related with studies in the areas of protocol conformance testing and fault injection. Section 2 presented the main aspects concerning these two fields. Here we discuss some previous work.

A closely related area is conformance testing based on formal methods. As discussed in [Hol91] the approach used in this study for test case generation is aimed at showing the behavioral equivalence

---

[4] Each frame is composed of various sub-frames, each of which is 146 words long.

[5] Obtained on http://fmt.cs.utwente.nl/ConfCase/v1.00/specifications/confprot1.efsm

between an specification, described as finite, strongly connected, deterministic automata, and an IUT that is supposed to implement it. In [UY91], [BDA+97] there are examples of previous work in this area. A work in this field that is very close to ours is the one presented in [TPB96], which describes the TAG tool that automatically generates test cases from FSM specifications. From this work we borrowed the FSM textual description. Another point in common with this tool is that Condado also provides complete or selective test derivation. When complete test derivation is used, a complete test sequence is generated which covers each path that begins and ends at the initial state. In selective test derivation, one must specify one or more transitions in the FSM, and only paths including those transitions are covered by the test sequence. However, Condado differs from TAG in three main points: (i) Condado does not implement state identification approaches for test derivation. (Although, these approaches allow for the identification of invalid states, they do it in a very high cost which makes them hard to use in practice); (ii) test cases in TAG are specified in SDL, whereas in Condado, TCL is used for that purpose; (iii) the test cases produced by the TAG tool do not include test parameters, whereas in Condado they do.

An approach that is parallel to ours is the one based in Labeled Transition Systems (LTS). A conformance testing framework, based on the concept of implementation relation. An implementation relation is a relation between the traces of the specification and the implementation [Tre99]. Different testing tools have been built based on this conformance framework. An example is TORX [TB99]. The tool supports the approach called on-the-fly testing, which combines test derivation and execution in an integrated manner. Instead of deriving a complete set of test cases (comprised of test events, each test event representing an IUT interaction), the test derivation process only derives the next test event which is immediately executed. This approach is useful to avoid an unmanageably large number of test cases which may be derived when using batch approaches. For the moment, ATIFS only implements the batch approach. To manage the large number of test cases the user can use selective test derivation or reduce the specification.

From the protocol conformance testing field we also borrowed the ferry-clip approach to build our test execution support tool, the FSoFIST. The ferry-clip approach was introduced in 1985 [ZR86]. In this approach both, the test channel and the ferry channel pass through the IUT. The PF replaces the UT in the System Under Test, and an enhanced UT resides in the same machine as the LT. As a result, the amount of test code in the SUT is reduced and synchronization between the UT and LT is easier. Many refinements has been applied to this architecture since then. In [ZLD+88] a reduction in the PF is proposed, by removing the interface region. The interface region converts data to/from the IUT, masking IUT dependent features from the testers. Part of these features were moved to a new module, called Service Interface Adapter. This module was introduced in the Test System to convert data from to/from the upper IUT interface. The other part constituted the encoder/decoder in the Test System, used to convert data to/from the IUT lower interface. Also, the ferry channel no longer passes through the IUT; it uses the underlying communication layer instead. The Ferry Clip based Test System (FTCS) was proposed in [CLP+89], where a Ferry Control Protocol provides a standardized interface on top of an existing protocol, which actually transfers the test data (the ferry transfer medium protocol). In this architecture, both interfaces of the IUT are controlled and observed by the PF. We borrowed this idea to build our *ferry injector*. Besides the improvement on the control and observation of the IUT interfaces, this architecture also provides an independent communication channel for ferry connection. This allows the Ferry Transfer Medium Protocol to be as simple as possible, in order to reduce the complexity of the PF. In addition, this guarantees the availability of a connection between the Test System and the System Under Test in case of crashes which can occur as a consequence of fault injection.

The studies presented so far are aimed at conformance and interoperability testing, but do not consider fault injection.

Fault injection on distributed systems is a very active area. In the past, most common fault injection approaches were made by hardware or through fault simulation. These approaches have been used even for software validation. For example, in [AAC+90] the validation of an atomic multicast protocol using hardware implemented fault injection is presented. More recently, software-implemented fault injection (SWIFI) is being used for that purpose. SWIFI approaches can be classified as *static*, when faults are introduced at source code, or *dynamic*, when faults are introduced during runtime [HTI97]. The tools presented here cover both approaches. In runtime fault injection, an additional software is necessary to inject faults into the system as it runs; this extra software is the fault injector. One main concern when building software fault injectors is intrusiveness. To reduce it to a minimum, the architecture of most of the tools is divided in two parts: a SUT-independent part, responsible for management functions and na interface with the user, which generally resides in a control node, other than the one hosting the SUT [SVS+88], [HRS93], [DJM96], [DNC03]. The ferry architecture is suitable for that purpose, because it defines a distributed test architecture in which the IUT-independent components reside in another host for sake of reduced intrusiveness. This was one of the reasons why this architecture was chosen for FSoFIST.

With respect to the SUT-dependent part, various mechanisms can be used. Some tools introduce the fault injection and monitoring features (or at least, a call to library functions responsible for those features) inside the source code. Another common approach when injecting communication faults consists of creating an extra layer between the IUT and the underlying communication service. This extra layer, also called fault-injection layer, may be introduced at kernel level or between the IUT and the layer immediately below it. FIAT [SVS+88] implements both approaches. EFA [EL92] and VirtualWire [DNC03] introduce an extra layer on top of a link layer in the protocol stack at kernel level. Orchestra [DJM96], on the other hand, introduces an extra layer (probe and fault injection layer) immediately below the IUT layer. This allows for testing from lower to higher levels. Fault injection features are introduced into a library that must be linked with the IUT before execution. Emulating a network layer is also a mechanism that can be used for fault injection. A tool called Mendosus [LMN+03] emulates a LAN to provide the user with a virtual network with tunable performance and fault injection characteristics. Due to the use of the ferry architecture, FSoFIST allows various of these mechanisms to be implemented; the PF and the FIM modules can be configured according to the user needs. A similar work can be found in [SW97]. There, the authors present a framework for the development of fault injection tools for the validation of distributed systems. This work differs from ours in that they are only concerned with fault injection support. Test case generation and results analysis are not considered.

Another aspect in SWIFI is related to input generation, which includes the workload as well as the faults to be injected. The workload is used to activate the SUT. In case of communication systems, the workload constitutes the messages exchanged with the IUT during an experiment run. Except for Doctor [RS93], which can generate synthetic workloads based on a specification; workload generation is usually left to the user. In ATIFS, this task can be performed automatically by Condado. Users can create their own scripts manually, either using GerScript or the FSoFIST tools. In regards to the faults, they can be described in a tool-specific notation, as for example, in FIAT, or in a language such as TCL (e.g., Orchestra [DJM96]) or C (e.g., EFA [EL92]), or else in a special-purpose language, as in VirtualWire [DNC03]. In ATIFS, the test cases (used as workload) as well as the faults are all described in TCL. Most tools generate faults randomly, as the experiments are aimed at evaluating dependability measures (e.g., the time taken between the activation of an error and its detection by an error recovery mechanism). In ATIFS, faults can be generated deterministically, as in Orchestra and EFA, for example, in this way increasing the fault revealing potential with fewer tests.

6. CONCLUSIONS AND FUTURE WORK

ATIFS is a toolset aimed at providing support for the testing of reactive systems. The set of tools developed up to now helps the user in various test activities, namely test case generation, test execution and results analysis. Test case generation is based on a specification in the form of Finite State Machines. Various testing techniques were combined to generate tests, covering the control part (valid sequence of interactions) as well as the data part (format and parameter values of the input interactions) of the specification. ATIFS supports two types of testing for the moment: conformance testing and fault injection testing. ConDado generates test cases based on the formal specification of the system under test in order to satisfy the conformance of the implementation with respect to the corresponding specification. In addition, FSoFIST is able to inject faults mimicking communication problems caused by the environment.

We are now using the tools with various case studies in order to validate not only their implementation but also the techniques implemented. In this paper we show partial results relative to the tests of a space application. Further testing is under way, in particular fault injection testing. The AnTrEx tool will also be validated using these case studies, since it has not been used in real day-by-day situations yet.

Besides continuing the testing activities with the MASCO application and the Conference Protocol, we also plan to implement the SeDados tool that will complement ConDado, allowing test case generation from an EFSM.

The toolset promises to support a test methodology that is the aim of further research. A well-established methodology is invaluable to put research into practice. The main difficulties found in the case study were to identify the simplifications on the original FSM specification in order to avoid test case explosion, mitigating the non-representative ones.

ACKNOWLEDGMENT

The authors would like to acknowledge the financial support from the Brazilian funding agencies National Research and Development Council (CNPq) and the Research Aid Foundation of S.Paulo State (FAPESP) for their partial support to this project. The authors also would like to acknowledge the referees whose insightful remarks considerably helped to improve the quality of the paper.

REFERENCES

[AAC+90] Arlat, J., Aguera, M., Crouzet, Y., Fabre, J.-C., Martins, E., Powell, D. Experimental Evaluation of the Fault Tolerance of an Atomic Multicast System. IEEE Trans. Reliability, 39 (4), (October/1990), 455-467

[Ara00] Araújo, M. R. R. Fsofist – a tool for fault-tolerant protocols testing. MSc dissertation, Computing Institute – Campinas University, Brazil, Oct. 2000 (in Portuguese)

[Bei90] Beizer, B. Software Testing Techniques. International Thomson Computer Press, 2nd. Edition, 1990.

[Bei95] Beizer, B. Black-Box Testing. John Wiley & Sons, 1995.

[Bin00] Binder, R. Testing Object-Oriented Systems – Models, patterns and Tools. Addison-Wesley, 2000.

[BDA+97] Bourhfir, C.; Dssouli, R.; Abdoulhamid, E.; Rico, N. Automatic executable test case generation for extended finite state machine protocols. Mar. 1997, In URL: www.iro.montreal.ca/labs/teleinfo/PubListIndex.html.

[Car97] Carvalho, M.J.M. Implementation and Testing of the Transfer Layer Protocol of the SACI-1 Ground Station Telecommand System. MSc Dissertation, Federal University of Rio Grande do Norte, December/1997. (in Portuguese)

[CFP96] Cavalli, A. R.; Favreau, J.P.; Phalippou, M. Standardization of formal methods in conformance testing of communication protocols. *Computer Networks and ISDN Systems*, 29, pp 3-14,1996.

[CLP+89] Chanson, S. T., Lee, B. P., Parakh, N. J., Zeng, H. X., Design and Implementation of a Ferry clip Test System, *Proc. 9th IFIP Symposium on Protocol Specification Testing & Verification*, Enschede, Holland, Jun. 1989.

[CS87] Castanet, R.; Sijelmassi, R. Methods and semiautomatic tools for preparing distributed testing. *Proc. Of Protocol Specification, Testing and Verification VII*, pp177-188, 1987.

[CVD92] Chanson, S.T.; Vuong, S. ; Dany, H. Multi-party and interoperability testing using the Ferry Clip approach. *Computer Communications*, v.15, n.3, Apr. 1992.

[DJM96] Dawson, S.; F. Jahanian,; T. Mitton. ORCHESTRA: a Fault Injection Environment for Distributed Systems. 26th. Int’l Symposium on Fault-Tolerant Computing (FTCS), Jun,1996. Available on site: http://www.ecs.umich.edu.

[DNC03] De, P.; Neogi, A.; Chiueh, T-C. VirtualWire: A fault injection and analysis tool for network protocols. Proc. IEEE 23rd. International Conference oin Distributed Computing Systems, Providence, Rhode Island, USA, 2003.

[EL92] Echtle, K.; Leu, M.. The EFA Fault Injector for Fault-Tolerant Distributed System Testing. *Proc. Workshop on Fault-Tolerant Parallel and Distributed Systems*, Amherst, USA, 1992.

[Gui96] Guimarães, M.S. A user interface framework for an integrated test toolset. Master dissertation, Computing Institute – Campinas University, Brazil, Nov. 1996. (in Portuguese)

[Hol91] Holzmann, G.J. Design and Validation of Computer Protocols, Prentice Hall, 1991.

[HRS93] Han, S.; H.A. Rosenberg,; K.G.Shin. DOCTOR: an Integrated Software Fault Injection Environment. Technical Report, University of Michigan n° CSE-TR-192-93, 1993.

[HTI97] Hsueh, M; Tsai, T. K; Iyer, R. K, Fault Injection Techniques and Tools. *IEEE Computer*, pp 52-75, Apr. 1997.

[ISO91] ISO 9646. OSI Conformance Testing Methodology and Framework , ISO 1991.

[Jeu99] Jeukens, A. “A script generation system”. Technical Report, Campinas University, Campinas, Brazil, Jul. 1999. (in Portuguese)

[LMN+03] Li, X.; Martin, R.; Nagaraja, K.; Nguyen, T.D.; Zhang, B. "Mendosus: a SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services". Technical Report DCS-TR-471, Rutgers University. Obtained in June/2003 on URL: http://www.panic-lab.rutgers.edu/~binzhang/mendosus.pdf.

[LWC96] LabWINDOWS/CVI-C for Virtual Instrumentation-User Manual, *National Instruments Corporation Technical Publications*, 1996

[Mat00] Mattiello-Francisco, M.F. "Software Requirements Specification for MASCO Telemetry Ground Reception". Tech. Rep. MASCO-SRS-001, INPE, May 2000.

[MSA99] Martins, E.; Sabião, S.B.; Ambrosio, A.M. - "ConData: a Tool for Automating Specification-based Test Case Generation for Communication Systems" – *Software Quality Journal*, Vol. 8, No.4, 303-319, edited by Anna Liu and Paddy Nixon - Kluwer Academic Publishers, 1999.

[Pre97] Pressman, R.S. “Software Engineering – a practitioner’s approach” fourth edition. McGraw-Hill, 1997.

[RS93] Rosenberg, H.A.; Shin, K.G.. Software Fault Injection and its Application in Distributed Systems. In *Proc. FTCS-23*, Toulouse, France, 1993.

[Sab99] Sabião, S.B. A test case generation method based on Extended Finite State Machines combining black-box testing techniques. Master dissertation, Computing Institute – Campinas University, Brazil, Jan. 1999.

[Ste97] Stefani, M.R. Trace analysis and diagnosis generation for behavior testing of a protocol implementation in the presence of faults. MSc dissertation, Computing Institute – Campinas University, Brazil, May 1997. (in Portuguese)

[SVS+88] Segall, Z.; Vrsalovic D.; Siewiorek, D.P.; Yaskin, D.; Kownacki, J.; Barton, J.; Dancey, R.; Robinson, A.; Lin, T. FIAT-Fault Injection Based Automated Testing Environment. In *Proc. FTCS-18*, Tokyo, Japan, pp 102-107, 1988.

[SW97] Sotoma, I.; Weber, T. S. AFIDS – Arquitetura para Injeção de Falhas em Sistemas Distribuídos, *Anais do XV Simpósio Brasileiro de Redes de Computadores*, São Carlos-SP, 1997.

[Tre99] Tretmans, J. PhD Thesis In: Web:http://www.fmt.cs.utwente.nl/publications/files/280-TrBe99.ps.gz

[TB99] Tretmans, J.; Belinfante, A.. "Automatic testing with formal methods". Proc. EuroStar'99: 7th. European Conference on Software Testing, In *Proceedings of the Conference on Software Testing, Analysis and Review. EuroSTAR´99*, Nov. 1999.

[TPB96] Tan, Q.M.; Petrenko, A.; Bochmann, G.V. A test generation tool for specifications in the form of finite state machines. In Proc. Int'l Communications Conference (ICC), USA, Jun/1996, pp225-229. Obtained on Web site: http://www.iro.umontreal.ca/labs/teleinfo/ReprPubTest.html.

[Ube01]. Uber, F.R. "Integrating Domain Testing into Extended Finite State Machine Testing," Master dissertation, Computing Institute – Campinas University, Brazil, Oct. 2001. (in Portuguese)

[UY91] Ural, H.; Yang, B. "A test sequence generation method for protocol testing". *IEEE Trans. On Communications*, 39 (4), pp 514-523, Apr. 1991.

[VBF+01] Villela, T.; Braga, J.; Fonseca, R.; Mejla, J.; Rinke, E.; D'Amico, F. "An Overview of the MASCO Balloon-Borne Gamma-Ray Experiment", *Advances in Space Research*, 2001.

[ZR86] Zeng, H.X.; Rayner, D. The Impact of the Ferry Concept on Protocol Testing. *Proc. V Protocol Specification, Testing and Verification*, pp 533-544, 1986.

[ZLD+88] Zeng, H.X.; Li, Q.; Du, X.F.; He, C.S. New Advances in Ferry Testing Approaches. *Computer Networks and ISDN Systems*, 15, pp47-54, 1988.