MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-10239-PRE/5757

A TOOL FOR FAULT INJECTION AND CONFORMANCE TESTING OF DISTRIBUTED SYSTEMS

Maria de Fátima Mattiello Francisco
Eliane Martins*

*UNICAMP – Instituto de Computação (IC)

Paper presented at the Latin-American Dependable Computing Symposium: LADC'2003, São Paulo, 21 a 23 de outubro de 2003.

INPE
São José dos Campos
2004

A tool for fault injection and conformance testing of distributed systems

Eliane Martins[1]
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 - São Jose dos Campos -12227-010 - SP - Brazil
FAX: +55-12-345-6625,
fatima@dss.inpe.br

Abstract. This paper presents an approach for conformance testing and fault injection of distributed systems supported by a tool named FSoFIST (Ferry-clip with Software Fault Injection Support Tool). The approach extends the ferry-clip concept to cope with fault injection. The ferry-clip concept was aimed at providing a highly modular, flexible and configurable architecture for protocol conformance testing. Due to these qualities, this architecture can be used for testing different protocol implementations with reduced effort. The work presents the design issues employed to achieve the ferry-injection architecture and describe the components of the proposed architecture. The capabilities of the approach are demonstrated in a case study used to validate the FSoFIST tool.

1 Introduction

On the last decades a continuous increase in the use of distributed systems has been observed in several applications: industry, offices, research and education institutions, banks, commercial organizations, and hospitals. The Internet extended the use of these systems to the domestic environment, supporting applications such as "home banking" and e-business among others.

Dependability becomes then an important property of such systems, as even more activities depend on their good performance. For that reason, dependability validation is more and more important, in order to guarantee that the developed system has the expected properties. In this paper our concern is with testing, aimed at revealing faults in a system's implementation.

Testing distributed systems usually requires various types of tests [2]: conformance, interoperability, quality of service testing. *Conformance testing* is aimed to determine if an implementation meets its functional specification. The *interoperability tests* aim to determine whether various components of a distributed system are

able to cooperate with each other to perform the specified services. In the *quality of service (QoS) testing*, the purpose is to assess the behavior of a distributed system to determine whether attributes such as performance, reliability or availability meet the expected standards. Complimentary to these techniques, fault injection allows validating the system in the presence of a variety of faults or errors, even those not anticipated in the specification.

Distributed systems consist of a number of independent components running concurrently on different machines that are interconnected by a communication network. Testing distributed systems is a difficult and challenging task for several reasons. One problem is the generation of adequate test cases that present good potential for uncovering faults. Another problem is the generation of the expected results for each test case. Generally these tasks are performed manually, but in the case of distributed systems the number of potential input sequences that the system can handle is infinite, which means that a subset of the possible input sequence represents indeed too much testing. This makes the cost of having people generating test cases and evaluating test data unfeasible. Although an important issue, test case generation and results analysis are not discussed further in this text. These were the concerns of two other tools developed by the group and can be seen in [16, 22].

Here our concern is with test execution support. One challenge is the intrinsic non-determinism of distributed systems. Non-determinism may occur when the implementation under test (IUT) cannot be accessed directly. For example, a protocol implementation is tested via an underlying communication service, or it is embedded in other protocol layers. This limits the view that the test component has of the system and the ability to reproduce the same IUT behavior when repeating the same input conditions. To reduce non-determinism, one has to introduce various test components into the System Under Test (SUT), which increase intrusion in the original system to be tested. Those test components may alter either the system structure (e.g., some fault injectors are implemented as extra code inserted either at the IUT or into libraries it access) or the behavior of the IUT, for example, by altering its execution speed, which may be a problem when testing real-time systems. Moreover, this requires distributed test components and, consequently, some mechanism to synchronize them.

Typically, distributed systems are heterogeneous in terms of communication networks, operating systems, hardware platforms and also the programming language used to develop individual components. This represents another challenge, in that a test system must be able to run in a wide variety of platforms and has to access different kinds of interfaces.

This text presents a tool built to support conformance testing, with the aim to give an answer the question: does the system comply with its functional specification? Fault injection is used as complement, in that it allows answering questions such as: how does the system react when whenever faced with unexpected or invalid behavior of its environment? The tool design is based on the *ferry architecture* [28] proposed in the context of protocol testing. The tool, named FSoFIST (Ferry-clip with Sofware Fault Injection Support Tool) [1], extends this architecture in order to support fault injection by software. The ferry architecture was adopted because of the following features:

- It offers the means to introduce test instrumentation inside the IUT with low intrusiveness;
- It provides the means to synchronize the test components;
- It is portable for different platforms and easy to adapt to different kinds of interfaces
- It is highly modular, which eases modifications;
- It can be easily extended to cope with testing of multiple IUT.

The paper is organized as follows: Section 2 presents basic notions about the two types of test currently supported. In Section 3 we present the proposed test architecture. Section 4 presents an implementation of the proposed test architecture, performance evaluation of the tool and also a case study for its validation. Section 5 describes some related works. Finally, section 6 concludes the paper. The appendix contains a list of abbreviations used in this text.

2 Types of Tests Supported

2.1 Conformance Testing

Conformance testing aims to determine whether an implementation meets the specification [12]. In the context of the Reference Model for Open Systems Interconnection (OSI), it was developed a particular standard for protocol conformance testing, the IS 9646: "OSI *Conformance Testing Methodology and Framework*" (CTMF) [14]. The standard defines the methodology, structure and specification of test sequences as well as the procedures to be followed. 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 (designated as abstract test methods) to support test execution (ISO IS-9646, 1993, Parts 1 and 2). A test architecture describe how an IUT (which represents a protocol entity) is to be tested, i.e., what inputs can be controlled and what outputs can be observed. As can be seen in figure 2.1 a, the points at which inputs and outputs to/from the IUT can be controlled and observed are called 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 access points (SAP), respectively named (N)-SAP and (N-1)-SAP.

Ideally each SAP is a PCO that is directly used for the communication with the IUT. But generally one or more SAPs are not directly accessible during testing. An IUT may be embedded in other applications or may be only accessed via underlying services. For these reasons, test architecture might be described in terms of [23]:

(i) **the PCOs**,

(ii) **the test context**, that is, the environment in which the IUT is embedded and that is used during testing to interface the IUT (the test context, also called test interface, is supposed to be correct);

(iii) **the testers** – there is a tester associated to each PCO, they are named upper tester (UT) and lower tester (LT) respectively connected to (N)-SAP and (N-1)-SAP. Whenever both testers are used, synchronization between them during testing is required; this is achieved by the Test Coordination Procedure (TCP).

Since, in real world, the IUT and the testers can reside on different machines, the following 4 architectures were proposed [14, 23] that can be classified as local or external. Figure 2.1 illustrates these architectures.

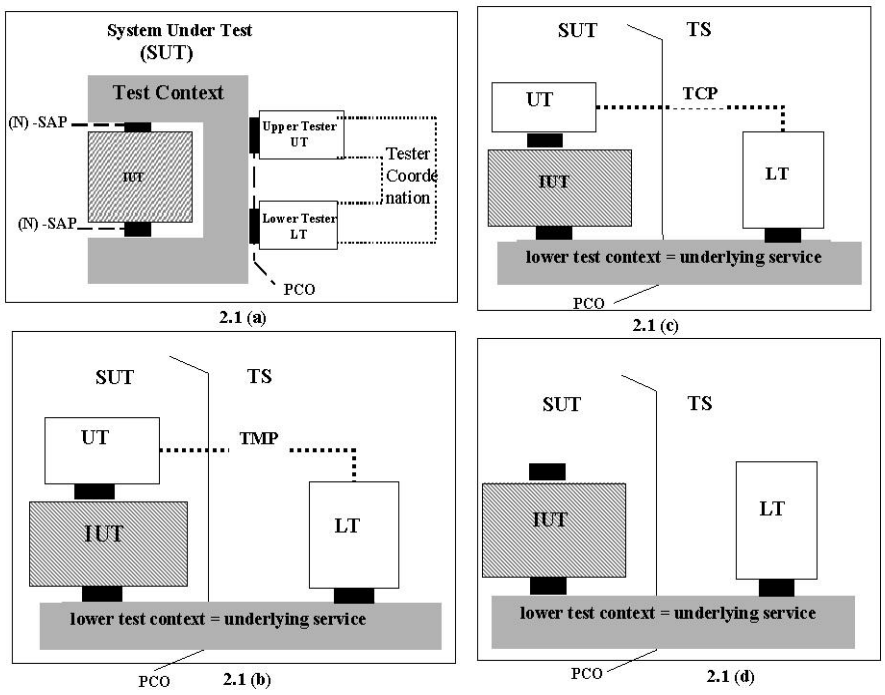

Figure 2.1. Conformance testing architectures.

In the local architecture (figure 2.1(a)), the IUT interacts with both testers, UT and LT, intercepting the SAP (Service Access Point) in the upper and lower interfaces of IUT, which are the tests PCOs.

The other three architectures are external, in which case the LT has remote access to the IUT. The LT resides remotely in the Test System (TS) and its interaction with the IUT is through the (N-1) layer. Test Coordination Procedure (TCP) shall implement a communication protocol between the TS and the LT. In the distributed architecture (figure 2.1(b)), the UT is physically close to IUT, directly accessing its upper interface. The Coordinated Method (Figure 2.1(c)) uses a Test Management Protocol

(TMP) as TCP. In the remote architecture, (Figure 2.1(d)), the UT does not access directly the IUT upper interface

Each test architecture has three variant forms [14]: single-layer, multi-layer and embedded. In single layer-methods, the IUT represents a single layer of a protocol stack and is tested without reference to the layers above it. Multi-layer methods are designed for testing a multi-layer IUT as a whole. In the embedded method a single-layer is tested within a multi-layer IUT; the SAP of the single-layer being tested is accessed through the layers above it, which constitutes the upper test context.

The architectures described so far are aimed at testing an IUT that communicates, in a point-to-point connection, with a single peer. This is the single-party testing context. A generalization of these architectures was proposed for the case in which an IUT participates in a multi-point connection with several peers at the same time. This constitutes the so-called multi-party testing context, in which several LT and zero or more UT are used to control and observe the IUT. Since our concern in this paper is single-party testing context, we will not discuss multi-party testing issues any further. Interested readers can consult references [6, 24] for further details on this topic. Also, [27] presents a good survey on distributed test architectures for protocol conformance testing according to CTMF and beyond.

2.2 Fault Injection

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

There are several approaches for fault injection [2, 7, 13]. These work addresses fault injections by software, in which changes in the state of the system under test, are performed under control of special software. In this way both hardware and software failure modes might be emulated. The mechanism consists in 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; as a routine started by a trace mechanism; as an extra code inserted into the IUT or in its context (operating system, underlying communication layer, for example). A pioneering work in [21] implements some of these mechanisms.

The fault injector implemented in ATIFS aims to mimic communication faults, which represent typical faults of distributed systems like message lost, duplication, corruption and delay. Communication fault injectors are generally inserted by a probe/fault injection layer between the IUT and the underlying service [9, 10, 11, 19, 20], as shown in Figure 2.2.

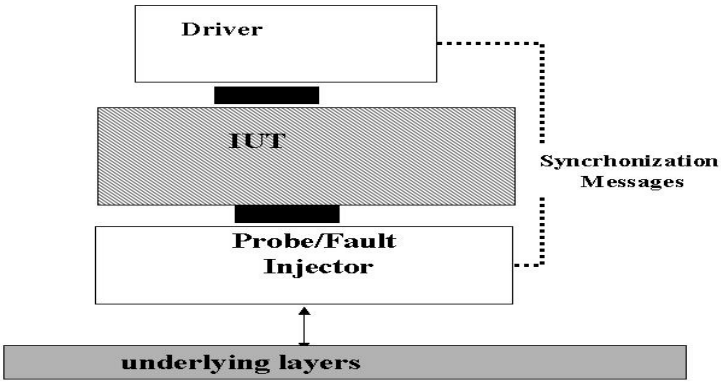

Figure 2.2. The probe/fault injection approach

Typically, fault injection is used later in the test process, mainly to obtain dependability measures. Here we are concerned with the testing of one IUT, and fault injection is used to ensure that erroneous messages can be generated as required by the test case being run.

3 FSOFIST design overview

3.1. The proposed ferry-injection architecture

The *ferry* architecture was proposed in [28, 29] to support protocols test architectures defined under ISO9646 standard. This architecture has a high modular structure, which allows its use in different platforms requiring few modifications. Basically, it consists of transporting the test data from the Test System (responsible to the test coordination) to the SUT, which contains the IUT, in a transparent way to allow those both upper and lower testers reside in the Test System. This approach simplifies the synchronization between the UT and LT, minimizing the amount of test software to be aggregated to the SUT. The study in [6] has shown the flexibility of the ferry architecture in supporting other types of testing. The architecture proposed here, designated as ferry-injection, extends the ferry-clips to support fault injection.

Figure 3.1 illustrates the distributed architecture proposed. 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 by the IUT, so the Test Sequence Controller is not modified to each new IUT.

As shown in figure 3.1, the test data are transported from the Test System (responsible for the 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 and stores data collected by the latter. The Fault Injection Module (FIM) intercepts the messages received by the IUT and inserts the faults determined by FIC.

The current version of FSOFIST is aimed at single-party testing, in which there is a single IUT that participates in a point-to-point communication with its peer. In this context, fault injection objective is only to ensure that errors are generated as required by the test case being run. The IUT can represent a single layer or multiple layers of a protocol stack, considered as a whole.

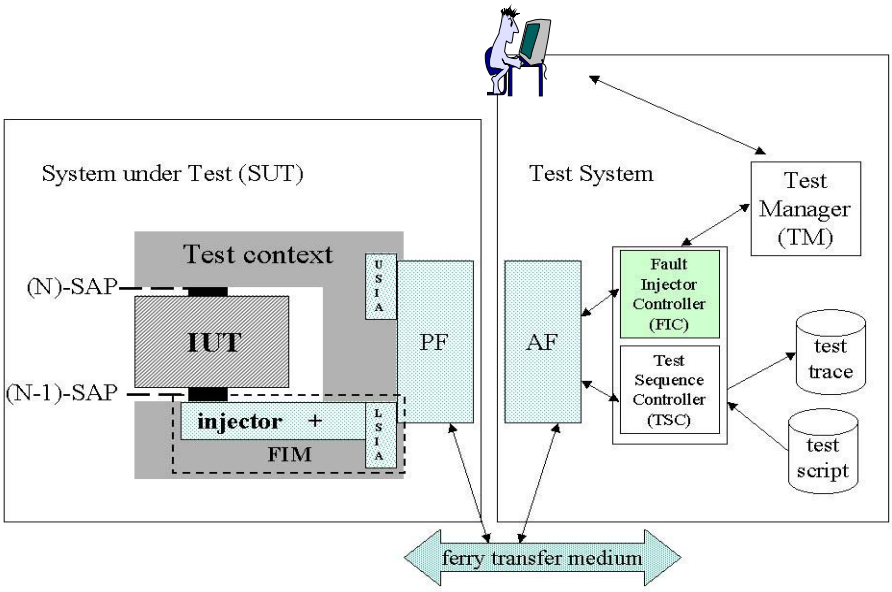

Figure 3.1. FSoFIST architecture

An important aspect concerning software fault injection tools is intrusiveness: the instrumentation introduced for injection and monitoring purposes may affect the structure (structural intrusiveness) or disturb the execution of the target system (behavioral intrusiveness). Our approach attempts to minimize intrusiveness in the following ways: (i) only the IUT dependent part of the fault injection features resides on the System Under Test (SUT), in that way reducing space overhead on this system; (ii) the FIM is part of the PF test component, affecting messages passing through the

IUT lower interface. In this way, IUT code is not necessarily modified, which avoids structural intrusiveness.

In the following, the major components of this architecture are described. Unless otherwise stated, the components functions are identical to that indicated in the literature [5, 6, 29] which can be consulted for further details.

3.2. The ferry protocols

The transfer of data between the AF and PF is achieved through the Ferry Control Protocol (FCP). Since the FCP is implemented in each ferry-clip, it should be as simple as possible, to simplify their implementation. The FCP defines abstract service primitives (FCP-ASP) that allow open and close a connection and the transfer of data between the two ferries. It also defines ASPs for the communication among the test components and the ferries. They are not presented here, for sake of space. Interested readers can consult [1] for further information The FCP uses services provided by the Ferry Transfer Medium Protocol (FTMP). To simplify matters to the FCP, this protocol has the following requirements: (i) guarantees end-to-end error free delivery of data and (ii) deliver packets in the same sequence they were sent. The SUT must provide such a protocol. In the present implementation of FSoFIST, the TCP/IP is used as the FTMP.

3.3. The ferry-clips

The structure of both ferry-clips is quite similar. They are structured into three modules:

- Ferry Engine (FE), which implements the core functions of the ferry-clip. It contains all functions that are independent of the IUT and FTMP being used. It interacts with the other test components by means of FP-ASP.
- Lower Mapping Module (LMAP), which maps the FP-ASP into the FTMP-ASP. Thus, this is the module to change for different implementations of the FTMP.
- Service Interface Adapter (SIA) which provides interface to the IUT. Since in the implemented test architecture the PF has access to both IUT interfaces, this module was not implemented in the AF. In the Passive Ferry this module converts data received through the ferry transfer medium into a format used by the IUT, and conversely, i.e., it converts IUT outputs into a format that the Test Suite Controller (TSC) understands. The PF-SIA is then logically structured in two sub-modules: the U-SIA, associated with the IUT upper interface, and the L-SIA, associated with the lower interface, as shown in Figure 3.3. Thus, the PF-SIA module is IUT-dependent, hence, it is the only module to be replaced if a different IUT is to be tested.

An important concern when designing the ferry clips is to keep the PF as small and simple as possible. This can be achieved, on the one hand, by choosing a simple, yet reliable, ferry-transfer medium protocol. On the other hand, the conversion

functions performed by the PF-SIA should be reduced to a minimum. One possible solution consists in externally specifying the IUT interface formats instead of coding the conversion into the PF-SIA. This is the solution we adopted for FSoFIST. The primitives (or PDUs), their respective parameters, as well as parameter length and the allowable range for each parameter are input to another tool, ConDado [16], that automatically generates test cases in the adequate format.

3.4. The Fault Injection Module

The Fault Injection Module (FIM) is an extension of the PF-SIA to incorporate fault injection capabilities. More precisely, it extends the L-SIA, given that communication fault injectors typically resides in some layer bellow the IUT (see section 2.2). When no fault injection is used for testing, the user can configure the PF to use the L-SIA instead of the FIM. It contains the Injection Logic, which performs fault-injection functions, and a message buffer to store messages to/from the IUT.

The FIM first processes the fault descriptor which was sent by the FIC. Then, it runs through a loop, intercepting messages from/to the IUT and checking whether a message should be injected or not, according to the fault descriptor. A message counter is used by the FIM to trigger fault injection. In this way, the FIM is able to inject various fault types such as: message delay (a message from/to the IUT is retained until a certain delay is up), message dropping (a message from/to the IUT is deleted), message duplication (a copy of a message from/to the IUT is maintained in the buffer and sent after a certain delay) and message corruption (the content of a message is modified). It can also inject faults permanently (affecting every message), intermittently (affecting certain messages according to a pre-specified frequency) or transiently (affecting only one message in a test run).

These fault types are applied only on incoming messages. When no faults are to be introduced, FIM only transfers messages directly to the IUT.

3.5. The Fault Injection Controller

The Fault Injection Controller (FIC) implements all fault injection functions that are IUT independent. Hence, changing the IUT has no effect on this module. It uses a script defined by the user determining: when to start fault injection, what faults to inject and for how long. This information is used to pass the appropriate commands to the FIM using the Ferry Protocol primitives. This module also stores data sent back by the FIM for post-mortem analysis.

3.6. Test Specification

The CTMF defines a test notation for test case specification: it is the Tree and Tabular Combined Notation (TTCN) [14, 18]. However we decided to use TCL for that purpose, for the following reasons: (i) at the time the tool was designed, TCL was a popular interpreted language, used in some fault injection tools (e.g, [9]); (ii) TCL

syntax is quite similar to C, which reduces the burden of learning a new language for users already familiar with this programming language and (iii) TCL allows users to write their own extensions, in C or C++, so that the script can invoke user defined procedures.

3.7. Requirements on the System Under Test

It is important to point out that there are also requirements on System Under Test (SUT) in order that our test architecture could be used. First of all, the SUT must allow the insertion of test components to control and observe the IUT interfaces. Another point is that it should allow the installation of a tester in a layer bellow the IUT to perform fault injection. Furthermore, it must provide an IUT-independent communication medium for the transfer of information between the Test System and the SUT.

4 Implementation of FSoFIST

FSoFIST was developed according to the ferry-injection architecture described in the previous Section. It presents a user interface allowing the tester to control and monitor the tests step by step, allowing the user to start, stop and continue a test session any time.

FSoFIST was developed under the Solaris 2.5 operating system, where it was first used. It was further ported to Linux. Its PF component ran initially in Solaris but was transferred to Windows platform for the case study presented in section 4.4.2. The AF is written in C++ language and the PF in Perl. The communication between them uses the socket library and the TCP/IP protocol, which guarantees portability, since socket libraries are available in various Operating Systems.

In the following we present the experiments performed to validate FSoFIST. The first experiments were aimed at evaluating the tools performance. The second one illustrates the use of FSoFIST in a simple, real world application.

4.1 Performance measurements

The SUT used in these experiments was a dual Xeon 300MHz CPU running a Linux Operating System. The PF and the IUT resided both in the SUT. The IUT in this case does not matter, since we were only interested in measure the overhead introduced by the test components. The Test System was implemented on an Ultra SPARC 366 MHz running Solaris. The two hosts were connected through a FastEthernet network.

The Test Manager implements a TCL Interpreter, which manages the execution of the testers (FIC and TSC). The Test System components were implemented as objects, and were written in C++.

The performance evaluation experiments are summarized bellow.

Measuring script interpretation overhead. These experiments were aimed at evaluating the tool's execution delay caused by the TCL Interpreter. For those purposes, four scripts were generated: a null script, and three scripts that executed a loop for 1000 times, 10000 times and 100000 times, respectively. The script is shown bellow and the results obtained are presented in Table 4.1.

```
set x 0
while ($x < 1000 ) { incr x }
```

The time taken to process the null script indicates the TCL Interpreter overhead. By subtracting this overhead from the times taken to execute the loops we obtain that each assignment operation takes about 7μs (e.g., 208 (the time taken to process the loop 10000 times) - 137 = 71 ms /10000 assignments).

Table 4.1. Script interpretation delays

Script	Execution time
Null script	137 ms
Loops 1000 times	141 ms
Loops 10000 times	208 ms
Loops 100000 times	684 ms

Measuring TSC-AF communication delay. These experiments were aimed at evaluating the execution delay caused by the communication internal to the Test System. For that purpose, we vary the size of the data exchanged between two modules: Test Suite Controller and AF. Three buffer sizes were considered: 10, 100 and 1000 bytes respectively. The script is shown bellow and the results obtained are presented in Table 4.2.

```
set data_buffer "xxxxxxxxxx"
time {
       senddata Lower data_buffer
     } 1000
```

Table 4.2. Internal communication delays

Number of bytes	Execution time
Empty data	13,353 s
Data with 10 bytes	13, 458 s
Data with 100 bytes	19,993 s
Data with 100 bytes	26,550 s

The internal communication is through pipes, and we considered that each pipe has the same execution speed, so only one pipe was observed. The connection with the PF was not opened, which causes the AF to respond to each request with a disconnect indication.

Measuring AF-PF communication delay. These measurements were aimed at assessing the response time in the ferry transfer medium. The PF was executed in loop back mode, that is, all data received was sent back to the AF. The script used in these experiments is presented bellow, and the results are in Table 4.3.

```
# opening the connection with AF:
openconn <PF#>  <host>  <port>
openconn 1 lua 2345

# putting PF in loopback mode
sendcmd 1 Loopback

# sending nb bytes buffer and awaiting to receive it
# back for 1000 times:
senddata <PF #> <SAP> <data>
time {
    senddata 1 Lower "xxx ...<nb bytes> ... xxx"
    recvdata 1 Lower
     } 1000

# closing the connection with the PF
closeconn 1
```

The measures in table 4.3 show the time taken by the TCL Interpreter to process the script as well as the internal communication overhead and the data transfer using TCP/IP as Ferry Transfer Management Protocol (FTMP).

The overhead in the communication between PF and IUT depends on the interfaces between these components, and for that reason it is not presented here.

Table 4.3. Ferry transfer delays.

Number of bytes (nb)	Execution time
nb = 0	20.89 s
nb = 100	57.75 s
nb = 1000	63.89 s
nb = 100000	86.10
nb = 1000000 bytes	244.6 3 s

4.2. Experiments with MASCO

The Telemetry Reception Software (TMSTATION) of the MASCO Telescope [25] implements a ground entity of the ground-board communication protocol which receives in real-time the data of the sky imaging in X-rays got by the MASCO telescope (a Brazilian space project). The data are acquired during approximately 30

hours during the telescope flight on board of a balloon mission. The experiment was developed by the Astrophysics Division at the National Institute of Space Research (INPE) and will be launched yet in 2003. The imaging data acquired by a hardware detector are organized in frames, stored in files on board and transmitted in real time to the Ground Station, where they are received by the TMSTATION software [17], as illustrated in figure 4.1.

The main function of TMSTATION is to separate the frames, sequentially received through a serial channel RS422, in distinct files, like they are stored on Hard Disk on board. The separation is based on the identification of a pattern, e.g. a string of at least 5 occurrences of the hexadecimal word "AA55", which shall be presented at the end of each frame. In the rest of the text we will refer to this word as the end-of-file (EOF) pattern. The software specification also required the TMSTATION application to report the status of each frame stored on ground according to the frame length. The TMSTATION was implemented in C language under LabWindow/ CVI [15] environment.

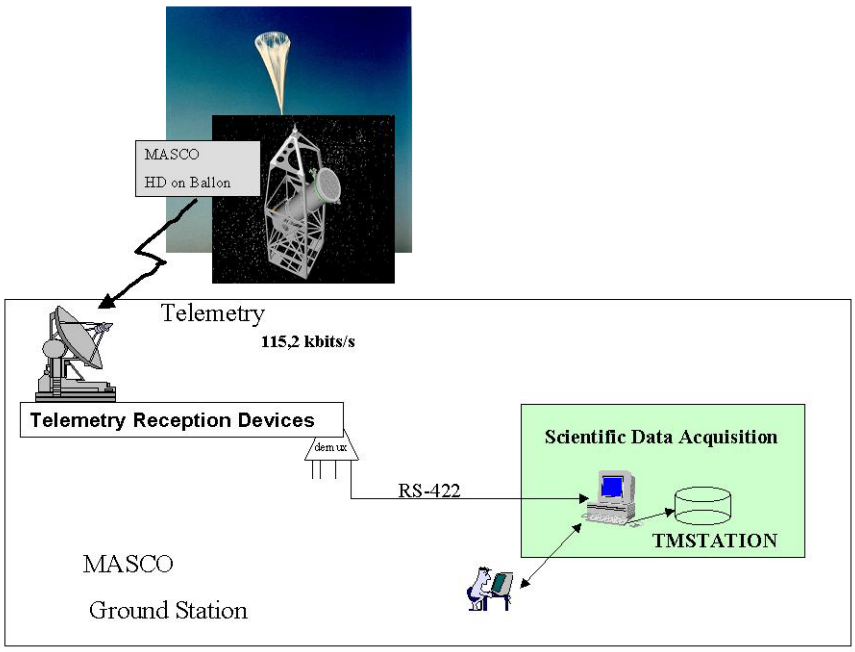

Figure 4.1 The TMSTATION software operation context for MASCO scientific data reception on ground.

Under normal transmission conditions from board to ground, the valid frame to be recorded at ground station has the following format: first, the word "EB90", followed by "937 x 146 words" (data field), finally the "end of file" pattern. In case of communication failure, part of the frame might be missed resulting:

(i) **Truncated frame**: a frame with reduced length is generated whenever part of the frame data field is lost, without affecting the EOF pattern
(ii) **Extended frame**: a frame with bigger length might occur if part of the EOF pattern is corrupted or lost. In this case, the TMSTATION aggregates the next frame to the current frame data field.

In order to validate whether the TMSTATION implementation deals correctly with the reception of valid frames, we used conformance testing. To emulate invalid frames, as described above, we used fault injection with the aim to mimic communication failures on the radio frequency downlink.

The FSoFIST architecture, presented in figure 3.1, was configured to achieve the TMSTATION testing needs as shown in figure 4.2. The physical serial channel RS-422 connects the two processes, PF and IUT, both residing on the same machine, on an Windows environment. The Lab-CVI library was used for the communication through the serial line, acting then as the test context. Faults are injected to emulate ground-board link failures. For that reason, we assumed that messages transferred to the IUT through the serial line are delivered in sequence, without modification. So, the FIM was not introduced inside the test context, as it should be (c.f. Section 3.1); instead, it uses this context to interface with the IUT. This configuration was useful given that our main interest was to validate FSOFIST module's behavior.

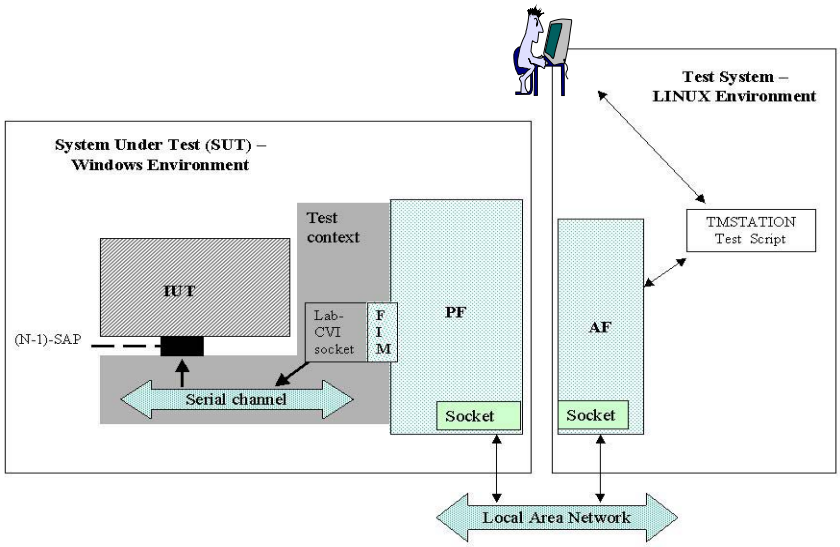

Figure 4.2. FSoFIST configuration for TMSTATION application testing

A TELEMETRY.dat file, containing a sequence of valid frames, was used during the tests. Various scripts were developed: one for conformance testing, and the others for fault injection. The conformance testing script merely reads frames on the

TELEMETRY.dat file and passes them to the IUT through the ferry clips. For the fault injection tests, the scripts were generated according to the types of faults to be injected (fault cases). Faults were selected deterministically to meet the frame violations bellow:

(i) omission of words in the data field – to emulate the reception of truncated frames.

(ii) change of the EOFpattern - to emulate failure in the transmission of this pattern which results in extended frames.

Figure 4.3 shows an example of FSoFIST window during test execution. The upper part of this window shows the script used to inject faults according to (ii) above. In this script, the last three words of the EOF pattern are changed from (AA55) to (9A05), intermittently on the 2nd, 3rd and 4th frames of a sequence composed of five frames. The lower part of Figure 4.3 shows part of the log generated during the tests.

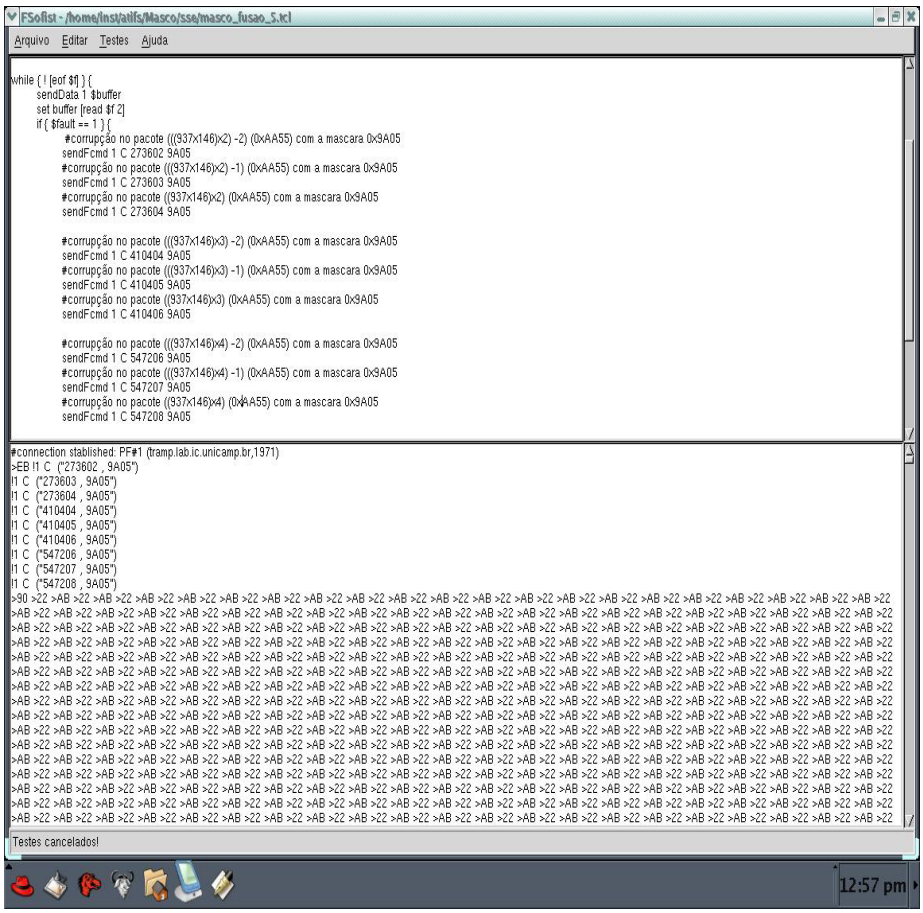

Figure 4.3. presents the script and test results associated to FSoFIST execution for extended frames test case.

Six scripts were created in order to validate fault injection capabilities of the FIM module. These scripts implement the prescribed error models according to (i) and (ii) above, and for each one we varied the repetition pattern (transient, intermittent and permanent). Table 4.4 describes them as well as the related results:

Table 4.4. Summary of the results of FSoFIST validation using TMSTATION software as case study.

Number of frames in input file	Error model	FSoFIST technique	Status of the stored frames (*)
6	Transient at 3rd frame	corruption at EOF pattern	1valid 2valid 3/4extended 5valid 6 valid
7	Intermittent at 2nd, 4th, 6th frames	corruption at EOF Pattern	1valid 2/3extended 4/5extended 6/7extended
5	Intermittent at 2nd, 3rd, 4th frames	omission of EOF pattern	1valid 2/3/4/5extended
3	Permanent	byte omission on the data field	1 truncated 2 truncated 3 truncated
5	Intermittent at 2nd, 4th, 5th frames	byte omission on the data field	1 valid 2 truncated 3 valid 4 truncated 5 truncated
4	Transient at 2nd, 3rd frames	byte omission on the data field	1 valid 2 truncated 3 truncated 4 valid

*Test Results: Status of the data files (frames) generated by TMSTATION

5 Related Works

Numerous approaches have been proposed for the validation of distributed 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 test architectures used in these areas. Here we discuss some previous work.

From protocol conformance testing field we borrowed the ferry-clip approach to build our protocol and fault injection tool. The ferry-clip approach is not new. Zeng et al introduced the concept in 1985 [28], where the PF was intended to replace the UT in the System Under Test, and an enhanced UT was moved to the same machine as the LT. As a result, the amount of test code in the SUT is reduced, and also, the synchronization between the UT and LT is easier. In their approach both the test channel and the ferry channel pass through the IUT. Many refinements were applied to this architecture since then. In [29] it is proposed a reduction in the PF by removing the interface region from it. 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, introduced in the Test System to convert data from to/from the upper IUT interface. Another part constitutes 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. Chanson et al proposed the Ferry Clip based Test System (FTCS) [5], where a Ferry Control Protocol was introduced to provide 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-injection tool. 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 availability of the connection between the Test System and the System Under Test in case of crashes that can occur as consequence of fault injection.

Another related work is the one presented by A.W.Ulritch et al, where two test architectures for testing distributed systems were proposed: one based on a global tester that controls and observes all distributed components of a SUT in a central manner, another based on a distributed tester that consists of a number of distributed, concurrent tester components, each of them observing a partial behavior of the IUT [24]. The latter must provide a test coordination procedure to assure a consistent global view of the SUT, which comprises various IUT. A tool, TMT (from Test and Monitoring Tool) to support both architectures was implemented in Java. TMT can be applied to systems implemented in C++ (for Unix-like or Windows NT platforms) or Java. The distributed test components are implemented in a test library. Call for functions in this library must be inserted into the IUT source code. The source code is then necessary since TMT implements a grey-box testing approach, in which instrumentation is introduced inside the IUT to observe its internal interactions. The architecture of FSOFIST can also be easily extended to incorporate multiple ferry clips inside the SUT. Grey-box testing can then be used, for instance, for a multi-layer IUT, allowing the observation exchanges at various layer boundaries. However, the fact that FSOFIST does not take for granted that source code is available, its implementation is not dependent on any specific programming language.

Also closely related to our work is the approach called on-the-fly testing, which combines test derivation and execution in an integrated manner. Instead of deriving complete test cases (comprised of test events, each test event representing an IUT interaction), the test derivation process only derives the next test event from the specification and this test event is immediately executed [26]. In this case, besides the Driver, that controls and observes the IUT during testing, there is another component, the Primer, responsible for the generation of a valid input derived from the specification, as well as for checking whether the output generated by the IUT is valid according to the specification. TORX [23] is an example of a tool developed to support this approach. The tool also supports batch test derivation and execution. A drawback of this approach is that the Test System is dependent on the specification formalism, since it is also responsible for test case derivation. In our case, this task can be performed either by another tool or by the user that can manually write her/his test cases using the editor available at FSoFIST interface.

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 by hardware or through fault simulation. These approaches have been used even for software validation (e.g. [3; 8]). More recently, software-implemented fault injection is been used for that purpose. SFI [20] and its successor, Doctor [19], use different approaches to inject different types of faults. To inject communication faults, library routines are altered so as to provide erroneous communication services. EFA [11] introduces communication faults by inserting a fault injection layer in the protocol stack. This extra layer is introduced on top of the network link layer. Fault injection control is a program compiled into the fault injection layer. This program can be automatically generated from a formal model of the system. VirtualWire [10] also introduces fault injection and monitoring features on top of physical layer, more precisely, between the network interface card's device and the IP protocol stack. VirtualWire is aimed at testing any network protocol operating within a local area network. Fault injection control is programmed in a declarative language specially developed for that purpose. PFI and its successor, ORCHESTRA [9] use a fault injection layer between the IUT and the layer below on the network stack, as the previous works. But differently from EFA and VirtualWire, this layer moves around the protocol stack. It is based on the concept of probes, introduced to network monitoring purposes. Similar to FSoFIST, the experiments are programmed by the user in TCL and C. This work is rather close to the approach used in this paper. They also defined an abstract fault-injection architecture and develop a tool based on it. Their architecture aims at protocol-independence, although their tool is devoted to protocols that use sockets on Unix-like operating systems. Fault injection capabilities were introduced at the routines which comprise the socket interface provided by a library. The IUT source code is not modified, but it must be re-linked with the new library.

Loki [4], as ORCHESTRA, is another tool that uses probes monitoring as well as fault injection purposes. Probes are inserted in each node to monitor state changes and to inject faults as required. Fault injection on a node is triggered according to state changes information, either local or from remote nodes. Probes can be inserted into the source code, whether it is available. In case the source code is not available, monitoring and fault injection is performed from outside the IUT. The users can either select a probe among the pre-implemented ones, or can develop their own probes.

Similar to Loki, in FSoFIST the users can also develop their own PF, more specifically, the SUT-dependent part of the PF (LMAP and SIA/FIM modules), which gives the tool more flexibility and applicability. The user can choose a suitable location to the SIA/FIM modules according to her/his test purposes. These modules can also be implemented using probes, as in ORCHESTRA and Loki. For the moment we do not have pre-implemented modules as in Loki, but this is envisaged for the near future.

6 Conclusion and Future Work

This paper presents the ferry-injection architecture for conformance testing and fault injection of distributed systems. We also presents a tool, FSOFIST, that implements the proposed architecture. Some experiments were performed to validate the tool and demonstrate the capabilities of the ferry-injection architecture. Preliminary results of performance evaluation are also reported. A small and simple, but real-world application was used as case study. The advantages of the proposed architecture include protocol independence and portability to different platforms. The architecture is also ease to use, since test cases for conformance testing, as well as fault injection experiments, might be specified using user-defined scripts in TCL, whose syntax is very similar to the C language. Ongoing activities on this project are considering the following aspects: (i) use of other distributed applications in the aerospace field as case studies; (ii) extension of the ferry-injection architecture for multi-party and interoperability testing. In the long term we plan to use other commercial or prototype distributed applications to evaluate the tool's capabilities.

Acknowledgment

The authors would like to acknowledge the financial support from the National Research and Development Council (CNPq) of Brazil and the team of the Astrophysics Division at the National Institute of Space Research (INPE) headed by Dr. João Braga, responsible for the MASCO project. We also thank Anderson Nunes de Paiva Moraes for executing the experiments with FSoFIST, and Ana Maria Ambrósio for the technical review.

References

1. Araújo, M. R. R. FSoFIST- A Tool for Fault Tolerant Protocols Testing. MSc Dissertation. Institute of Computing – State University of Campinas (UNICAMP). (October/2000). (In Portuguese)
2. Arlat, J., Crouzet, Y., Laprie, J.-C., Fault Injection for Dependability Validation of Fault Tolerant Computing Systems. Proc Int'l Symposium on Fault-Tolerant Computing (FTCS-19), Chicago, USA, (1989)
3. 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
4. Chandra, R., Lefever, R.M., Cukier, M., Sanders, W.H., Loki: a State-Driven Fault Injector for Distributed Systems. Proc. Int'l. Conference on Dependable Systems and Networks (DSN'00), New York, USA, (2000) 237-242
5. 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, Enscchede, The Netherlands. (1989) 101-118
6. Chanson, S.T., Vuong, S., Dany, H., Multi-Party and Interoperability Testing using the Ferry Clip Approach. Computer Communications, 15 (3), (April/1992)
7. Clark, J. A., Pradhan, D. K., Fault Injection: A Method for Validating Computer System Dependability. IEEE Computer, (June/1995) 47-56

8. Czeck, E., Siewiorek, D., Effects of Transient Gate-Level Faults on Program Behavior. Proc Int'l Symposium on Fault-Tolerant Computing (FTCS-20), (1990) 236-243
9. Dawson, S., Jahanian, F., Mitton, T., ORCHESTRA: a Fault Injection Environment for Distributed Systems. Available on site: http://www.ecs.umich.edu.
10. 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)
11. 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)
12. Holzmann, G.J. Design and Validation of Computer Protocols, Prentice Hall, (1991)
13. Hsueh, M., Tsai, T. K., Iyer, R. K., Fault Injection Techniques and Tools. IEEE Computer, (April/1997) 52-75
14. ISO TC97/SC21, IS 9646. OSI Conformance Testing Methodology and Framework, ISO/1991.
15. LabWINDOWS/CVI-C for Virtual Instrumentation-User Manual, *National Instruments Corporation Technical Publications*, 1996
16. Martins, E., Sabião, S.B., Ambrosio, A.M., ConData: a Tool for Automating Specification-based Test Case Generation for Communication Systems. Software Quality Journal, 8 (4), (1999) 303-319
17. Mattiello-Francisco, M.F. Software Requirements Specification for MASCO Telemetry Ground Reception. Internal Report MASCO-SRS-001, INPE, (05/2000)
18. Probert, R., Monkevic, O. , TTCN: The International Notation for Specifying Tests of Communication Systems. Computer Networks & ISDN Systems, 23 (1992) 111-126
19. Rosenberg, H.A., Shin, K.G., DOCTOR: an Integrated Software Fault Injection Environment. Technical Report, University of Michigan n° CSE-TR-192-93 (1993)
20. Rosenberg, H.A., Shin, K.G., Software Fault Injection and its Application in Distributed Systems. In Int'l Symposium on Fault-Tolerant Computing (FTCS-23), (1993), Toulouse, France
21. 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 Int'l Symposium on Fault-Tolerant Computing (FTCS-18), Tokyo, Japan, (1988) 102-107.
22. Stefani, M.R., Trace Analysis and Diagnosis Generation for Testing the Behavior of a Communication Protocol in the Presence of Faults. MSc. Dissertation, Institute of Computing, State University of Campinas, (May/1997)
23. Tretmans, J., Belinfante, A., Automatic Testing with Formal Methods. Proc. 7th. European Conference on Software Testing, Analysis and Review. EuroSTAR´99, (November, 1999)
24. Ulrich, A W., Zimmerer, P., Chrobok-Diening, G., Test Architectures for Testing Distributed Systems. Proc. of Software Quality Week, (1999)
25. Villela, T., Braga, J., Mejá, J., D'Amico, F., Alves, A., Silva, E., Rinke, E., Fernandes, J., Corrêa, R., Preflight Tests of the MASCO Telescope, Advances in Space Research, 26(9), (2000) 1411--1414
26. Vries, R.G., Tretmans, J., On-the-fly Conformance Testing using SPIN. In G.Holzmann, E.Najm, A.Serhrouchni, (ed.), 4th. Workshop on Automata Theoretic Verification with the SPIN Model Checker, Paris, France, (November/1998) 115-128
27. Walter, T., Shieferdecker, I., Grabowski, J., Test Architectures for Distributed Systems – State of the Art and Beyond. In, Testing of Communicating Systems, 11, A. Petrenko, N.Yevtuschenko (ed.), Kluwer Academic Publishers, (September/1998). Obtained in 2002 at URL: http://citeseer.nj.nec.com/walter98test.html

28. Zeng, H.X., Rayner, D., The Impact of the Ferry Concept on Protocol Testing. In Proc. V Protocol Specification, Testing and Verification, (1986) 533-544
29. Zeng, H.X., Li, Q., Du, X.F., He, C.S., New Advances in Ferry Testing Approaches. Computer Networks and ISDN Systems, 15 (1988) 47-54

Appendix

AF	Active Ferry	PCO	Point of Control and Observation
ASP	Abstract Service Primitive	PDU	Protocol Data Unit
CTMF	Conformance Testing Methodology and Framework	PF	Passive Ferry
FCP	Ferry Control Protocol	SAP	Service Access Point
FE	Ferry Engine	SIA	Service Interface Adapter
FIC	Fault Injection Controller	SUT	System Under Test
FIM	Fault Injection Manager	TCP	Test Coordination Procedure
FTMP	Ferry Transfer Medium Protocol	TM	Test Manager
ISO	International Standard Organization	TMP	Test Management Protocol
IUT	Implementation Under Test	TS	Test System
LMAP	Lower Mapping Module	TSC	Test Sequence Controller
LT	Lower Tester	UT	Upper Tester