This is a draft version. A complete version is published in the Proceedings of *Workshop of Dependable Software Tools and Methods (DSN-2005)*, Yokohama, Japan, June-July 2005

A Methodology for Designing Fault Injection Experiments as an Addition to Communication Systems Conformance Testing

Ana Maria Ambrosio[1]
Fátima Mattiello-Franscisco[1]
Nandamudi L. Vijaykumar[1]
Solon V. de Carvalho[1]
Valdivino Santiago [1]

[1]*National Institute for Space Research (INPE), Av. Dos Astronautas, 1758, São José Campos, SP, 12227-010, Brazil* ana@dss.inpe.br;

Eliane Martins[2]
[2]*Institute of Computing - Campinas State University (UNICAMP) P.O. Box 6176 - Campinas, 13083-970, SP, Brazil*
eliane@ic.unicamp.br

Abstract

Conformance testing aims at validating if an unit under test accepts the normal-expected inputs; and if it properly rejects erroneous-inputs such that consistency with the original specification is maintained. Methods for automatically generating test cases for protocol conformance testing have long been progressed. Most of them depend on state-based specifications and may detect transition faults. However, real protocol faults go beyond the state transition faults. Based on previous research in building testing tools for conformance and for fault injection, we propose a testing methodology which guides the test personnel to model the exceptional behavior of a unit under test using UML-diagrams and an external fault model as well. The modeling starts from a textual specification towards the fault cases definition. The fault cases are based on the communication fault-model and executed with SWIFI. The fault cases are deterministic fault injection experiments once they are derived from specification models. The methodology has been used to validate INPE's in-house implementation of the communication system between the On-Board Data Handling (OBDH) computer and a payload experiment of Brazilian scientific satellites. The methodology and the experimental results of testing a real communication system are discussed and analyzed.

Keywords: testing methodology, communication faults, conformance testing, SWIFI.

1. Introduction

The International Standard Organization (ISO) and the International Telecommunication Union (ITU-T) have standardized procedures for communication protocols with practical guides for conformance testing, stated in the IS-9646 standard [8]. This standard describes the structure of an abstract test suite to be obtained from a standard protocol specification and used to certify that a corresponding implementation conforms to the specification.

In addition to the ISO practical testing guides, protocol conformance testing using formal methods have been researched. This test consists in checking the conformance of an implementation with respect to a specification written in a formal language. If the specification is in a formal notation, such as a finite-state machine, one may use graph algorithms to automatically generate cases for such specification [5], [15]. Automated test methods lead to time and effort reduction with tests as well as to assure great coverage of the specification with the tests. However, there are difficulties in obtaining a formal representation of a specification and also in executing and analyzing the results of the huge test suite (generated by exponential combination of input events) of real protocols [9].

According to Holzmann, a set of conformance test cases aims to establish that a given unit under test: (i) performs all functions of the original specification, over the full range of parameter values and (ii) can properly reject erroneous inputs in such a way that it is consistent with the original specification [7]. This definition has motivated us to model the system behavior under correct and erroneous inputs in separate models, thereby deriving abstract test cases and fault cases from distinct models. So, it reduces the difficulty of handling the complexity of the specification as a whole and allowing the execution of the fault cases with the Software-Implemented Fault-Injection (SWIFI) technique. Moreover, fault injection has been invaluable in space application testing due to its capability to model space environment faults which commonly occur in the system's operational phase.

Once the domain of real protocol faults is much broader than faults related to state transitions [13] this approach allows augmenting the fault classes for evaluating communication systems.

In order to master the fault case derivation, which may be executed with SWIFI a systematic testing

methodology, named conformance and fault injection (CoFI), is presented. This methodology addresses the issue on "*how to deduce faults to inject, starting from a specification?*" and makes a comprehensive use of the ATIFS project' tools [11].

In CoFI, normal and exceptional scenarios are mapped into separated models; consequently, each model is smaller than the complete model. The "normal" models allow one to suitably apply algorithms to generate test cases. The "exceptional" models guide a fault injection process in: (i) monitoring the fault cases coverage against to the test criteria *all-exceptional-scenarios*, (ii) facilitating the detection of design/implementation faults of fault tolerance mechanisms when dealing with erroneous inputs caused by the medium.

The paper is organized as follows. Section 2 presents concepts used in this article. In section 3 the CoFI methodology is explained. Section 4 illustrates the approach with the OBDH-Apex communication software testing; which provides the communication between an intelligent scientific experiment (the Apex) with the on-board data handling computer of a Brazilian satellite. Section 5 discusses similar solutions. Finally, section 6 concludes the paper.

2. Some Concepts and Assumptions

A ***finite state machine*** (FSM) $M = (\Sigma, O, Q, \delta, \lambda, q_0, F)$ where: Σ -alphabet of input symbols, O - alphabet of output symbols, Q - finite set of possible states, δ -transition function $\delta: Q \times \Sigma \rightarrow Q$, $\lambda: Q \times \Sigma \rightarrow O$, q_0 -initial state, $q_0 \in Q$, F -set of ending states, $F \subset Q$; allows to represent a system behavior specification. The diagrammatic representation of a FSM is a directed graph, thus algorithms may be used to traverse the graph and generate test cases (a sequence of input and their corresponding output symbols = $\{<i_1,o_1>, <i_2,o_2>, ..\}$) [15].

The ***fault injection technique*** means to submit a unit under test (UUT) to faults in order to detect and remove design and implementation faults. The fault, representing hardware mal-functioning, may be random or deterministically chosen [2], [6].

Experiments of fault injection are executions of fault cases, i.e., sequences of correct and faulty inputs. They are characterized by the FARM attributes, where: F={faults to be injected}; A={test data aimed at exercising the injected faults}; R={observed outputs generated during the experiments}; M={measure obtained from the readouts, that allows the verdict for fault removal}[1]. A fault case is a sequence of inputs, corresponding outputs and an fault indication = $\{<i_1,o_1f_0>, <i_2,o_2,f_1>, ..\}$, where: $i \in$ A, $o \in$ R and $f \in$ F, and f_0 means no fault.

In CoFI, the FARM attributes mean: F={faults caused by the communication medium}; A={observable inputs arriving in the UUT that lead it to a state in which faults may occur}; R={observed outputs obtained during the fault cases execution}; M={verdicts obtained on comparing the expected output with the observed outputs}.

A ***fault model*** is a class of faults. A communication fault model comprises the typical *faults* caused by the communication medium. A communication fault is characterized by the following parameters:

- *when* - time or message identification, when the fault should be injected,
- *what* - fault type: lost, delay, corruption or duplication [7],
- *how* – the way a message should be corrupted,
- *where* - fault location to where the message flows.

We have considered the communication fault model as a set of the following fault primitives: f.corrupt(*when, how, where*); f.delay(*when, how, where*); f.duplic(*when, where*); f..lost(*when, where*).

A ***fault injector*** is a mechanism, external to the UUT, able to inject the faults during the test execution which supports the fault primitives.

A ***test architecture*** establishes the testing elements and the visible points of the UUT available for testing. In this work the *ferry-injection* test architecture [18] is taken into account. The general elements of the ferry-injection are illustrated in Figure 1. This architecture allows an upper tester (UT) and a lower tester (LT) to observe and to monitor the UUT via points of control and monitoring (PCO) available in the UUT. PCOs are communication channels. The fault injector (FI) is able to inject faults via the lower PCO.

For the given architecture, the fault parameters are: *when*- the ordinal number of the messages passing through the PCO in which the fault is to be injected, i.e., the first, the second, etc.; *what*- one of the fault primitives that may be applied; *how*- the corrupted field values of the massage or the delay time according to the fault primitive; *where*- a PCO.

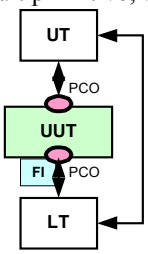

Figure 1: Testing architecture.

3. CoFI: Conformance-and-Fault-Injection testing methodology

CoFI consists in building, step-by-step, scenarios designed in UML-based diagrams (use-case, sequence and state diagrams), from which *abstract test* and *fault cases* are derived. The steps allow one to map every UUT-specified observable inputs and outputs. In short, the methodology main steps are:

1. identify inputs, outputs, points of control and observation and the available elements of test architecture, including the injectors;
2. describe the functions of the UUT, in use case notation, identifying only the *normal scenarios* based on the system behavior in presence of the valid inputs,
3. describe the normal scenarios in a normal sequence diagram. At least the UUT, its peer and the communication medium must be the entities represented in vertical lines of the diagram;
4. translate the sequence diagram into a FSM. One may find in literature some algorithm guiding this translation [14]. Then, automatically derive test cases;
5. identify the *fault model* in agreement with the communication medium features and the faults the UUT is supposed to tolerate;
6. create *exceptional scenarios* based on the fault model and write them in *exceptional sequence diagrams*:
   a. identify the entities interacting with the unit under test, including the fault injectors,
   b. describe the scenario as a sequence of interactions arriving and leaving the UUT as horizontal lines,
   c. for each normal scenario defined in step 3, combine a fault type of the fault model, identify what should be the system reaction under such a fault, then create a new exceptional scenario,
   d. in that exceptional scenario, draw, firstly, a fault interaction, an interaction between the fault injectors representing the fault to be injected to origin such a scenario;
   e. repeat steps 6.c and 6.d for every fault of the fault model (step 5) applied to every normal input of every normal scenario of the step 4; (repetitions should be avoided);
7. check if the scenarios created in step 6 cover all exceptions defined in the specification. If not complete them;
8. derive a *fault case* from each exceptional scenario. The inputs are obtained from every normal interaction arriving in the UUT, the expected outputs are obtained from the interactions leaving the UUT and the faults are obtained from the fault interactions (the first interaction drawn in an exceptional scenario).
9. optionally, exceptional scenarios may be translated to one or more FSM. In this translation, the fault interaction information is used to mark the transition representing the fault. Once the scenarios are modeled in FSM other combinations of normal and exceptional inputs may be obtained with algorithms that traverse the machine and create fault cases as the test cases are created.

The *abstract test and fault cases* are derived from state diagrams (FSM) in such a way that a path through the state machine [15] is a test or fault case. This derivation is automatically performed by using Condado [10]. The test and fault cases are abstract as they are implementation independent and have a tool-independent description.

The generation of faults to be injected is deterministic as the faults are derived from the exceptional scenarios. The test criterion for the fault cases is to cover every exceptional scenario established in the specification basis.

The UML-models used in this methodology contain enough information to allow automation. Presently, steps 4 and 9 are fully automated and a tool for supporting the fault cases execution with reusable fault injectors has been already underway. The automation for creating exceptional scenarios on combining normal scenarios with the fault of the fault model (step 6) has been intended to be designed as well as, the derivation of fault cases from sequence diagrams to script that may be directly executed by the FIC.

4. Case study

In order to evaluate the CoFI approach in a realistic application we have used the *commands recognition software*, part of the communication protocol developed at the National Institute for Space Research (INPE). This protocol is in charge of providing the communication between on-board data handling computer (OBDH) and an intelligent satellite payload for Brazilian scientific satellites. The UUT is the command recognition implementation of the payload experiment named APEX – *Alpha, Proton and Electron Monitoring Experiment in the Magnetosphere*, to be included in the EQUARS – *Equatorial Atmosphere Research Satellite*.

The APEX sends data only under OBDH request, in a master-slave relationship. The physical communication is the RS-422 serial interface in asynchronous mode. Whenever the experiment fails, the OBDH timeouts. The command message sent by the OBDH to APEX is shown in Figure 2.

This is a draft version. A complete version is published in the Proceedings of *Workshop of Dependable Software Tools and Methods (DSN-2005)*, Yokohama, Japan, June-July 2005

SYNC	EXPID	TYPE	SIZE	DATA	CKSUM
			Optional Field		

Figure 2: Command message sent to APEX.

The fields SYNC (synchronism) and EXPID (experiment identification) must have the value EB 92H, meaning a communication starting. TYPE contains commands such as resetting the micro-controller, transmitting scientific data, memory load, memory dump, load parameters, send clock, initiate and stop data acquisition. SIZE identifies the amount of bytes in DATA field and CKSUM is the checksum of the ongoing message.

The APEX accepts a command message byte-to-byte. If a field value is not in agreement with the specification or if it does not arrive within 500ms the full message is discarded and the APEX waits for a new command. The starting of a new message is identified by the value EBH.

4.2. Applying the CoFI

Step 1: for testing the command recognition, the inputs being considered are the bytes arriving in APEX; the outputs are null if everything goes correctly; the PCOs and the testing architecture are shown in Figure 3, which includes the fault injector controller (FIC) and the fault injector module (FIM) The option was to avoid as much intrusiveness as possible in flight software, so the FIM is not into the UUT.

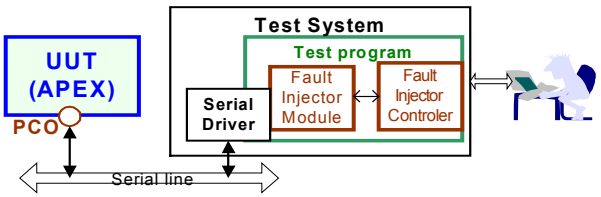

Figure 3: Test architecture for the OBDH-APEX.

Step 2: use cases were omitted here for brevity.

Step 3: two of nine *normal scenarios* in sequence diagram are shown in Figure 4. The entities are the OBDH, the communication medium and the APEX which were divided into 3 components: command recognizer, clock and protocol.

Step 4: the FSM translated from the normal sequence diagram (see Figure 5) was submitted to Condado, generating 10 conformance test cases.

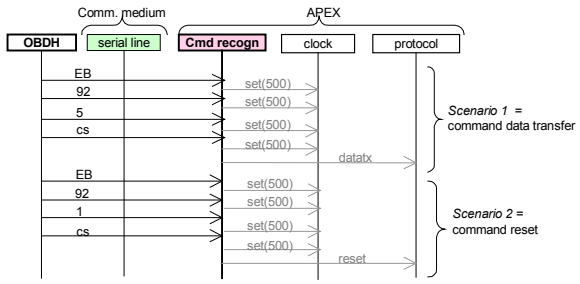

Figure 4: N*ormal scenarios* in sequence diagram.

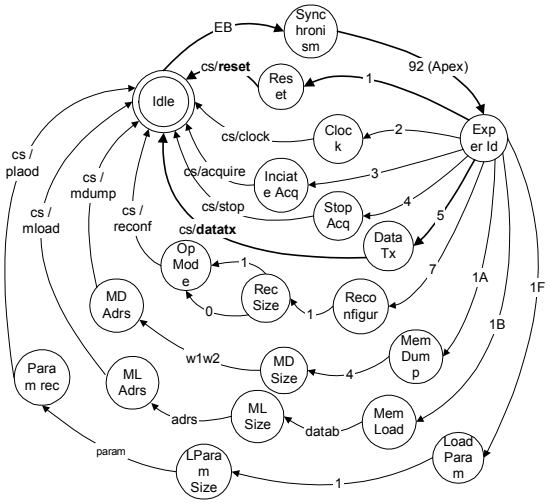

Figure 5: Normal scenarios in FSM.

Step 5: the communication medium between the APEX and the OBDH is the RS-422, in which Byzantine faults are possible, so bytes may be lost, duplicated, delayed, and corrupted.

Step 6: for the exceptional scenarios two extra entities are included: the FIC and the FIM. The interactions between the injectors, illustrated in the sequence diagram, represent the communication fault in such a scenario. On applying the four fault types to each correct byte arriving to the APEX in the normal scenarios (without repeating it for EB and 92 as they are equal in all scenario), 116 exceptional scenarios were obtained. Figure 6 shows only three *exceptional scenarios*, two of them are based on the corruption fault in the incoming byte of the command and one is based in the lost fault. Note that, in Figure 6, the first scenario (first msg corrupted) is obtained when one applies a corruption fault to the SYNC byte transmitted by the OBDH. In the third scenario the third byte is lost and do not achieve APEX. Duplication and delay are created in the same way. In all of these scenarios the command is not recognized by APEX and no reaction is expected.

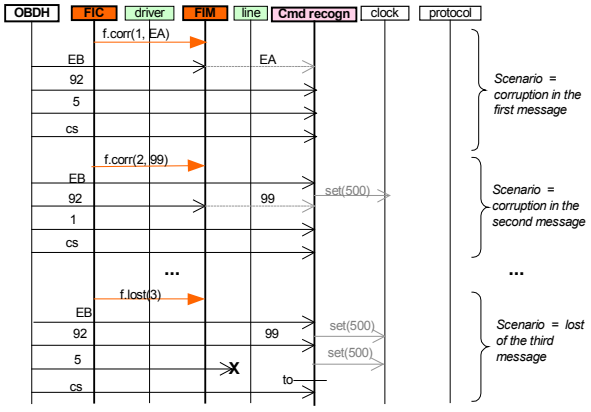

Figure 6: *Exceptional scenarios* in sequence diagram.

Step 7: the scenarios created with the fault types covered all the specified exceptions. These scenarios are important to uncover the cases to validate the implemented fault tolerance mechanisms.

Step 8: the *fault case* were derived from each exceptional scenario. Table 1 illustrates six fault cases, whose inputs are shown in column *Valid Inputs* and the fault parameters are in the next column. This translation was manually performed. The fault parameter location requires information about the point of control and observation (PCO) where the faults will be injected. This is implementation related information, which will be obtained at a later stage.

Table 1: Fault cases.

	Valid Inputs	**Fault Parameters**			
		when	*what*	*how*	*where*
1	EB9205cs	1st byte	corrupt	EA	PCO
2	EB9205cs	2nd byte	corrupt	99	PCO
3	EB9205cs	3rd byte	corrupt	99	PCO
…					
7	EB9205cs	3rd byte	delay	600	PCO
8	EB9205cs	4th byte	delay	600	PCO
…					
17	EB9201cs	3rd byte	corrupt	99	PCO
…	…	…	…	…	…

4.3. Fault injection experiment results

The defined fault cases covered every exceptional scenario, which mapped the system behavior in presence of the set of defined communication faults. Every 20 test cases plus 116 fault cases were applied to the APEX command recognition implementation. The tests were executed with the help of a testing program developed by the APEX development team. This program was adapted for supporting delay, lost, duplication and corruption of the bytes according to values chosen by the user, via a graphical user interface. Table 2 summarizes the observed outputs of some executed fault cases.

Table 2: Fault injection experiments results.

Input	***Observed Output***	***Comments***
EA9205cs	timeout	SYNC not recognized
EB(f.delay)9205	no reaction	No reaction
EB(f.delay)92	timeout	EXPID - not received
EB9905cs	timeout	EXPID - unknown
EB9299cs	timeout	TYPE - invalid
EB92(fdelay)01	timeout	TYPE - not received
...	...	...
EB921A04FFFFFFFF	timeout	Incorrect address
EB921F0131CC	timeout	CKSUM - incorrect

22,2% of the cases covered *delay*, 22,2% covered *duplication* and the others covered data *corruption* and *loss*. The fault cases covering delay and loss had not been previously designed by the development team. Therefore, neither the fault cases nor the test cases uncovered new errors in APEX implementation.

This may be explained by the fact that APEX implementation, used in this case study, had already been tested by experts when we applied the cases created with CoFI.

Although, the CoFI tests did not detect any error, this approach was well-accepted by the test personnel as it allows them to plan the test effort based on a model-based covered test strategy. Moreover, it guides the definition of scenarios leading them to face not only with the internal defects (software bugs) but also focus on the environment (and its faults) under which the system will be running. It provides the test personnel with a new vision of conformance by adding a systematic way of generating fault cases based on the faulty inputs caused by the medium. Besides, CoFI leads test and fault injection design to be performed early in the development process.

5. Related solutions

The CoFI methodology is based on two research areas: protocol conformance testing and experimental evaluation by software-implemented fault-injection. In this article we have focused on the definition of fault injection experiments.

The fault injection technique has been mostly applied to experimental evaluation in which the system is stressed with randomly inputs selected from fault/error models that are intended to simulate. However, deterministic fault injection has shown good results in [2], [6].

Echtle and Chen [6] showed analytically that deterministic fault selection offers certain benefits over random injection in fault-tolerant protocol testing. They inject the faults at message level based on the knowledge of the program control-flow graph. The fault injection is deterministically guided by coverage criteria on such a graph. In CoFI the faults are applied in the messages but the program control flow is unknown. The faults are defined on system exceptional behavior basis and on the coverage criteria *all-exceptional-scenarios*.

In [2] a graph, representing the execution tree, is used to map all the possible executions of fault tolerant mechanisms. Sequences of deterministic fault injection for testing the corresponding implementation are automatically derived from the execution tree by static analysis. Paths from the root to the leaves are formally defined as set of assertions (or formula). The formulas allow characterizing the fault injection-based test sequence. In CoFI, the test sequence, including the fault parameter values, are derived on the analysis of the exceptional UML-sequence diagrams.

In [4] a state-driven fault injection tool that controls fault injections in concurrent systems is defined. CoFI does not treat concurrence issues. The fault injection experiments consist in executing each

fault case at once: as soon as an experiment finishes, the system returns to its initial state.

Statecharts diagram are used in [12] to model the system behavior under explicit faults and their effects. Fault injection is not used, instead a formal analysis is performed on model for faulty state analysis. Error-free and the erroneous behavior are modeled into a single Statecharts. In CoFI, flattened Statecharts and other UML-diagrams help the modeling of the exceptional behavior for fault cases designing in several diagrams.

Binder [3] defines a test strategy based on state-based representation of a system, similar to CoFI, in which Statecharts represent object-oriented implementations. This strategy guides the definition of test cases. The completeness assumptions on the flattened Statecharts improve the test case effectiveness. However, neither fault injection nor test cases providing external faults coverage are used.

6. Conclusion

This paper has presented the CoFI testing methodology which guides the design of fault injection experiments based on exceptional scenarios of the system behavior. The methodology consists of a set of steps, starting from a textual specification, toward exceptional scenarios that fault tolerance mechanisms should support.

Normal and exceptional scenarios are designed in UML-diagrams, such as, use cases, sequence and state diagrams. Fault cases are derived from sequence diagram or optionally from the state diagram. For the latter, the Condado is used to automate this step. The fault cases complement the test cases created to cover the normal behavior. The use of UML-diagrams points out mainly advantages for testing as they are usually developed in early software development phases; the time spent with tests is reduced, moreover the automation of the steps are feasible.

The methodology is illustrated with a realistic application, an implementation of the On-Board Data Handling (OBDH)-APEX communication software developed at INPE. APEX is a scientific experiment to fly on board of a Brazilian satellite. The complete test suite for this example consisted of 126 elements from which 8 percent are test cases and 92 percent are fault cases, so, covering every specified scenario.

This methodology for designing tests has shown advantage in the sense that it provides a systematic way for generating test (conformance) and the fault cases (fault injection experiments). It has allowed fault injection experiments to be defined and applied to different units under test developed from a common specification. As CoFI does not require any knowledge of the source code fault cases can be re-used.

Further research has been performed in the test suite effectiveness evaluation and the use of other fault models especially useful in space applications.

7. References

[1] Arlat, J.; Aguera, M.; Amat, L.; Crouzet,Y.; Fabre, J.-C.; Laprie, J.-C.; Martins, E.; Powell,D. Fault Injection for Dependability Validation: A Methodology and Some Applications. IEEE Tr on SE, v. 16, n.2, p. 166-182, 1990.

[2] Avresky, D; Arlat, J; Laprie, J-C; Crouzet, Y. Fault Injection for the Formal testing of Fault Tolerance. IEEE Tr on Reliability, v.45, n.3, p.443-455, 1996.

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

[4] Chandra, R.; Lefever, R.M.; Cukier, M; Sanders, W.H. A global-state triggered fault injector for distributed system evaluation. IEEE Tr on Parallel and Distributed Systems – July 2004 p.593-605 v.15 n.7

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

[6] Echtle, K; Chen, Y. Evaluation of Deterministic Fault Injection for Fault-Tolerant Protocol Testing. IEEE 21th Annual International Symposium on Fault-Tolerant Computing, Montreal, June 1991, p.418-425.

[7] Holzmann, G. J. Design and validation of computer protocols**.** Prentice Hall, 1990.

[8] International Organization for Standardization ISO/IEC–IS9646 International standard conformance testing methodology and framework. Geneve, 1991.

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

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

[11] Martins, E. ; Mattiello-Francisco, F. A Tool for Fault Injection and Conformance Testing of Distributed Systems. Lecture Notes in Computer Science, v. 2847/2003, pp. 282-302, Brazil, oct 2003.

[12] Pataricza, A; Majzik, I; Huszer, G; Varnai, G. UML-based design and formal analysis of the safety-critical railway control software module. In G. Tarnai and E. Schnieder: Formal Methods for Railway Operation and Control Systems, Hungary. p.125-132, 2003.

[13] Riese, M.; Vijayananda, K. Characterization of protocol faults based on experience in interoperability testing. Interoperability in Broadband Networks,1994. p.71-80.

[14] Rumbaugh,J.; Blaha,M.; Premerlani,W; Eddy,F; Lorensen,W. Object-oriented Modeling and Design. USA: Prentice-Hall International, 1991. 500 p.

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