VALIDATION-BASED DEVELOPMENT OF DEPENDABLE SYSTEMS

EARLY VALIDATION OF FAULT TOLERANCE IS ESSENTIAL IN DEVELOPING DEPENDABLE COMPUTER SYSTEMS. THE AUTHORS HAVE DEFINED A STRATEGY FOR TESTING FAULT TOLERANCE MECHANISMS, INTEGRATED IT INTO THE DESIGN PROCESS, AND DEVELOPED FAULT INJECTION TECHNIQUES FOR VHDL MODELS THAT ARE SUPPORTED BY A GENERIC TOOL.

Jean Arlat,
Jérome Boué,
and Yves Crouzet
LAAS-CNRS, France

Because fault tolerance is important in justifying our reliance on critical computing systems, validating fault tolerance is essential. Historically, certain techniques have helped designers in this process: axiomatic methods including formal verification, analytic evaluation and empirical methods based on fault injection, including injecting faults on prototypes, either by forcing voltage levels on IC pins or perturbating software layers. (For more information, see the "Background and related works" sidebar.)

Here, we focus on fault tolerance modeling and the control of activity flow in fault tolerance mechanisms (FTMs). We propose a fault injection-based testing framework linked to the design of fault-tolerant systems described as VHDL models and illustrate the application of our strategy with a case study corresponding to a simple embedded real-time fault-tolerant system. We address four main issues relevant to the development and validation of dependable systems:

- Modeling of the main structural features and interactions of the FTMs using concise diagrams. Such modeling aims at the high-level characterization of the testing strategy (for example, the order of the FTM testing).
- Characterization of the predicates that specify the outcome of the test sequences. We use these predicates to discriminate the observations obtained during the fault injection experiments.
- Modeling of the dynamic behavior of the FTMs using detailed behavioral models. We use these models to derive the test patterns to be applied. Such a derivation is based on a statistical analysis of these detailed models.
- Development of a generic tool to implement fault injection into VHDL models; that is, supporting the validation process accounting both for fault removal and fault-forecasting objectives.[1]

Testing framework

Current FTMs are software protocols that are often complex and specifically developed

0272-1732/99/$10.00 © 1999 IEEE

Background and related works

The development and early validation of dependable systems encompasses two main issues, namely the development of a specific fault injection-based test strategy targeted to the removal of fault tolerance deficiences and the conduct of simulation-based fault injection experiments.

Removal of fault tolerance deficiencies

Due to the very negative impact of deficiencies affecting the design and/or implementation of FTMs, the early verification of these mechanisms is essential. As for any verification activity, formal methods based on a mathematical model allow for the level of trust to be significantly enhanced. However, despite continuous progress of these methods, the number of uncertainties attached to the characterization of a computer system's behavior in the presence of faults, is such that it prevents these methods from providing alone an answer to this problem.

In this context, the experimental approach is perfectly suited to help take up this challenge. The fault injection method, in which the observation of the behavior of the computer system in the presence of faults is explicitly forced by the deliberate introduction of faults, constitutes a privileged approach. Indeed, fault injection can be seen as a form of testing the FTMs with respect to a class of specific inputs they were specifically designed to cope with: the faults. In spite of this a priori favorable environment, in most work emphasis is mainly put on evaluating the efficiency of the FTMs, the issue of removing fault tolerance deficiencies only comes into play as a subproduct of fault injection experiments. More focused testing approaches are needed to devise cost-effective testing strategies. To date, few studies have addressed this topic. Nevertheless, two main pioneering investigations are worth pointing out.

The first study[2] details a deterministic software test approach involving fault tolerance protocols. It relies on the insertion of software probes into the program implementing the target fault tolerance protocol (after each conditional branching) to provide execution traces for each experiment. Its has been applied to simple redundancy management protocols (for example, of the Triple Modular Redundancy type). The second study[3] combines a formal description and simulation. It relies on the determination of a formal model (a state table) of the fault tolerance protocol from which, irrespective of the program implementing the protocol, a set of assertions characterizing the properties of the service expected from the program is expressed using a rigorous formalism. It has been applied on the replica treatment protocol developed within the ESPRIT Delta-4 Project.

Besides these advances, the problem of systematic testing by fault injection to remove deficiencies in the FTMs remains widely open. Recently, an increasing number of studies have been reported that aim at facilitating the systematic testing of fault tolerance by considering a wide variety of approaches: error propagation analysis,[4] increasing stress,[5] monitoring the activity flow in the fault tolerance mechanisms,[6] application-specific safety criteria.[7] We based our approach in this article on the modeling of fault tolerance and the control of the activity flow in the FTMs.

Simulation-based fault injection

Simulation-based fault injection is intended to impact as early as possible the development process. Simulation-based fault injection covers a wide spectrum of activities ranging from the detailed study of the effects of specific types of faults to more general objectives by using more comprehensive supporting tools.[8-10]

FOCUS[7] supports mixed-level simulation (both at electrical and logical levels). The targets are the electrical nodes where transient current sources are applied. The effect of the injection is analyzed in details at the electrical level by considering transistor networks at the vicinity of the injection point, while the rest of the circuit is simply simulated at gate level.

The ASPHALT tool[9] is aimed at analyzing the faulty behavior and the error propagation process of complex digital devices. The tool considers an RTL language description of the target IC. Accordingly, the results obtained can provide objective error patterns to be applied during fault injection experiments carried out using the SoftWare Implemented Fault Injection (SWIFI, for short) technique (see Carreira, Madeira, and Silva[11]). The experiments aimed at determining to what extent such an RTL fault model covers actual hardware faults on the basis of a comparison with a gate-level simulation. The target system considered was the IBM RISC-oriented microprocessor.

DEPEND[9] is presented as a system-level functional simulation tool. Using the properties of the object-oriented paradigm, complex objects are built from elementary objects made available in a library (CPU, communication link, voter, server, and others). To speed up the simulation, DEPEND supports acceleration techniques, encompassing hierarchical modeling and variance reduction. Besides the dependability analysis of the Tandem Integrity S2,[10] two other recent applications of the tool concern: the analysis of a complex RAID storage system[11] and the study of a high-speed network system using both simulated injection and SWIFI.[13]

The previous studies feature dissimilar simulation languages at different phases of the target system's development. We advocate that the development of an integrated and coherent design environment for fault-tolerant systems based on a single language seems to be achievable when considering the emergence of hardware description languages. In this respect, VHDL (Very High Speed Integrated Circuits Hardware Description Language)[14] has been identified as a suitable language as it presents many useful features:

- the ability to describe both the structure and behavior of a system in a unique syntactical framework;
- widespread use in digital design and inherent hierarchical abstraction description capabilities;
- recognition as a viable framework (1) for developing high-level models of digital systems (block diagrams, Petri nets), even before the decision between hardware or software decomposition of the functions takes place, and (2) for supporting hybrid (mixed abstraction levels) simulation models; and
- the capability to support testing activities.

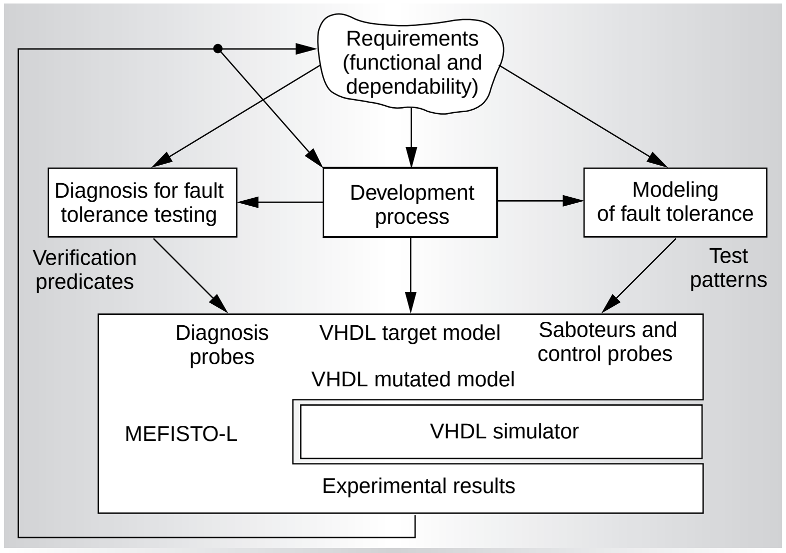


Figure 1. Framework for testing fault tolerance.

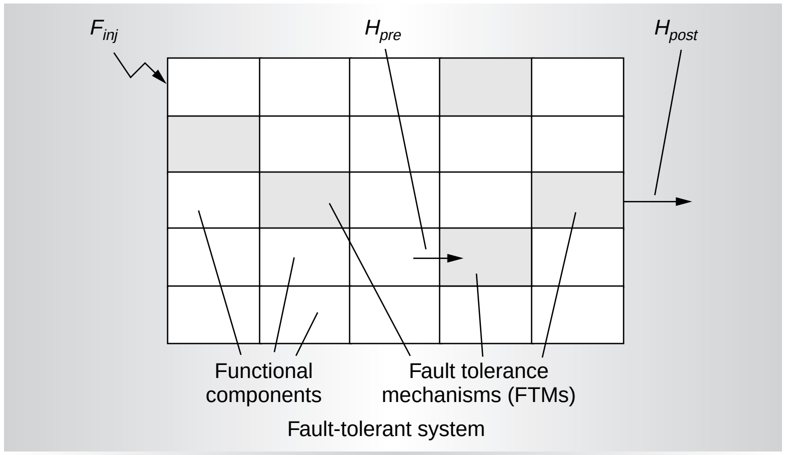


Figure 2. The F_{inj}, H_{pre} and H_{post} predicates.

to ensure error processing in fault-tolerant distributed systems. The underlying idea is to make fault injection experiments less "blind" by promoting a "glass box" approach that draws upon information about the implementation or relies on abstract models. (This is in contrast to the frequently used black-box notion that relies simply on the input-output characteristics of the mechanisms tested.) The main objective is to facilitate, or even guide, the test of these mechanisms by identifying testing criteria and ensuring test input generation (that is, errors combining faults, and functional activity) that facilitate the coverage of the criterion retained. Therefore, it appears that these experiments are close to the testing problem and that of the software testing, in particular.

Our proposal aims at integrating fault tolerance testing together with system design. The goal is to identify and implement relevant testing criteria for activating fault tolerance and adequate readouts for observing test outcomes, so as to guide the specification of the fault injection experiments. These experiments are then applied to a simulation model of the target fault-tolerant system to adequately exercise the FTMs. Figure 1 illustrates the proposed framework for testing fault tolerance and its links with the development process. The framework identifies two main facets of the proposed strategy: diagnosis of fault tolerance for specifying verification predicates and modeling of fault tolerance for test pattern generation. The figure also depicts the role played by MEFISTO-L (Multilevel Error/Fault Injection Simulation Tool developed at LAAS) in supporting the strategy.

Diagnosis for fault tolerance testing

Diagnosis of fault tolerance relies on the identification of suitable predicates for characterizing the behavior of FTMs, as well as their potential deficiencies.

Behavioral predicate of an FTM. Figure 2 provides an abstract view of a fault-tolerant system combining functional components and FTMs. The testing strategy proposed here is meant to test individually each specific FTM embedded in the system by injecting faults into the system. The figure also illustrates the three predicates whose combinations are used to identify the situations that may reveal deficiencies in the tested FTMs. These predicates are defined as follows:

- F_{inj} is true when a fault is injected into the system (thus, simulating the presence of an operational fault);
- H_{pre} is true when the error patterns observed on the input of the FTM match the failure mode assumptions against which its has been designed;
- H_{post} is true when the failure mode assumptions specified for the system are satisfied by the delivered service.

The injection of a fault into the system leads to the assertion of F_{inj}. The behavior of the

Table 1. Characterization of fault tolerance deficiencies.

$\downarrow F_{inj}$	H_{pre} — $H_{post} \longrightarrow$	False — False	False — True	True — True	True — False
	False	Nonsense $\neg F_{inj} \Rightarrow H_{pre}$	Nonsense $\neg F_{inj} \Rightarrow H_{pre}$	Nominal in absence of fault 1	Fault tolerance deficiency 6
	True	Expected behavior 3	Bonus 4	Nominal in presence of fault 2	Fault tolerance deficiency 5

components may verify H_{pre} on the inputs of the considered FTM. Only under this condition is the FTM expected to correctly handle the fault or its manifestations. The proper behavior of the FTM is assessed through the observation of system service (observation of H_{post}). Thus, the combined consideration of the three predicates, F_{inj}, H_{pre}, and $H_{post.}$ provides a means of revealing potential fault tolerance deficiencies.

Strictly speaking, the verification of fault tolerance necessitates only the removal of design faults affecting FTMs. This corresponds to checking whether the implementation of the FTMs matches their specification. The objective is then to ensure that an FTM activated by inputs satisfying H_{pre} delivers a service such that the outputs of the system satisfy H_{post}.

However, the verification of fault tolerance may lead designers to question the specification itself; that is, the representativeness of faults and errors handled by an FTM (and formalized by H_{pre}) with respect to actual operational faults. In case of a mismatch, one might discuss the choice of a particular FTM, for two reasons. The FTM may make stronger assumptions than needed on the failure modes of the components and thus needs to be enhanced, or the FTM is "overabundant" (that is, assumes weaker assumptions than necessary), while a simpler and more cost-effective mechanism would be sufficient.

Deficiencies of an FTM. Table 1 depicts all the combinations of the F_{inj}, H_{pre}, and H_{post} predicates.

According to the definitions of F_{inj} and H_{pre}, two out of the eight combinations in Table 1 are impossible. Indeed, a failure mode assumption includes the correct service of a component, that is, in the absence of fault. The six other combination definitions (as represented by the numbers in Table 1) are

1. Representation of the nominal behavior of the system in the absence of a fault.
2. When a fault is injected, the components provide FTM inputs satisfying H_{pre}, and the mechanism acts correctly so that the system delivers a service satisfying H_{post}. This is the expected behavior of the system in the presence of fault.
3. The case when a fault is injected and leads to a violation of the failure mode assumptions defining H_{pre}. The cause may be found in the specification or the type of injected fault is sufficiently rare or of limited consequence to be worth considering. For our concern here, the expected behavior of the considered FTM is that it will not be able to handle such a case.
4. Again, the case when a fault is injected and leads to a violation of the failure mode assumptions defining H_{pre}. However, the FTM can handle this unfavorable case, and the system delivers a service satisfying H_{post}. The FTM thus offers a bonus coverage for the system. As a simple example of such a case, consider a parity error detector; although specifically designed to detect single errors, it also can detect every odd order error.
5. The case when a fault is injected and the components provide inputs to the FTM satisfying H_{pre}, but the FTM is unable to act so that the system delivers a service satisfying H_{post}. This reveals a fault tolerance deficiency.
6. The case when no fault is injected and the components provide inputs to the FTM satisfying H_{pre}, but the mechanism, only by its presence, prevents the system from delivering a service satisfying H_{post}. In this case, the FTM is harmful to the dependability of the system, because it constitutes an additional source of possible failures. This case also reveals a fault tolerance deficiency.

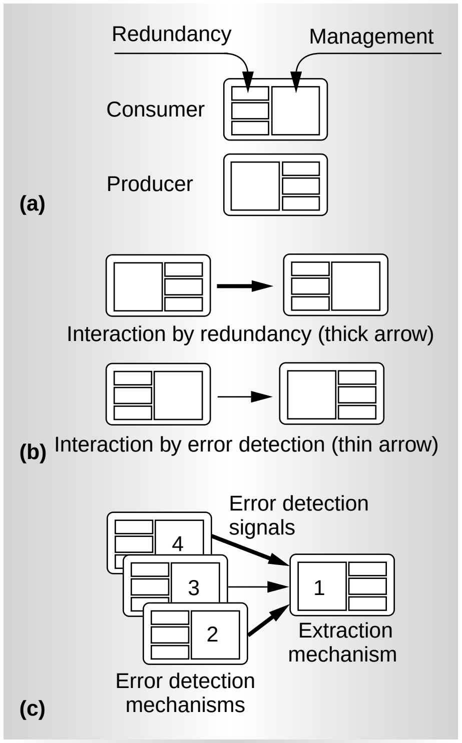

Redundancy
Management
Consumer
Producer
(a)
Interaction by redundancy (thick arrow)
Interaction by error detection (thin arrow)
(b)
Error detection signals
4
3
1
2
Extraction mechanism
Error detection mechanisms
(c)

Figure 3. Symbols used in DRIFTs: vertices (a), interactions (b), and an example (c).

Removal of fault tolerance deficiencies then consists in identifying experiments corresponding to cases 5 and 6. Their common characteristics are that the H_{pre} predicate is true, while the H_{post} predicate is false. The difference between cases 5 and 6 is that in the former faults are injected while in the latter no fault needs to be injected.

Test pattern generation

The modeling of fault tolerance that supports the generation of the test patterns is carried out in two steps using two models. The models describe 1) the overall organization of fault tolerance by means of Diagrams of the Redundancies and Interactions of Fault Tolerance (DRIFTs), 2) and the behavior of the individual FTMs.[16] A formalization of the proposed testing strategy is also given.

Modeling of fault tolerance using DRIFTs. We can identify fault tolerance chains that are composed of several mechanisms in series such as coding and decoding devices, error detection and error recovery, and broadcast and vote. The mechanisms in the chains exchange data by means of the underlying redundancy via a producer-consumer relationship or via specific error detection signals.

A DRIFT gathers within a single graph several fault tolerance chains that may share some mechanisms. The vertices of the graph correspond to FTMs while the arcs show the data exchanges (the interactions) between the mechanisms. Two distinguishable main types of vertices are the redundancy producers and redundancy consumers (Figure 3a). Also, two forms of interactions are identified via redundancy or error detection signals (Figure 3b).

As an example, consider the DRIFT representation of a compound block of FTMs including error detection and extraction mechanisms (Figure 3c). Such a block is often used to set the system state to a naturally safe position (for example, the idle state). In that case, the fail silence property is often desired. This property is implemented by one mechanism whose function is to isolate a failed module. This mechanism (labeled FTM1) receives several signals from the error detection mechanisms (labeled FTM2 to FTM 4) monitoring the module state.

The role of a DRIFT is triple. It

- represents a synthetic view of fault tolerance,
- identifies precisely the FTMs and their interdependencies, and
- contributes to the organization of the testing strategy.

Distinguishing between functional components and FTMs is not always an easy task. Nevertheless, the DRIFT formalism offers an objective support for describing the overall fault tolerance of a system. The construction of a DRIFT guides the analysis of the system specification to extract relevant information related to fault tolerance. Moreover, this selective reading of the specification reveals potential incompleteness, inconsistency, or inaccuracy. The few details required for this operation permit a relatively early action. The general principle of the testing strategy is to proceed block by block, in an order that follows the information flow in the DRIFT. As a consequence, the FTMs are tested according to increasing interdependencies between fault tolerance blocks (detection, recovery, reconfiguration). Such a progression follows the increased complexity attached to the organization of the FTMs and of their interactions within the DRIFT (for example, see the later Figure 7). In addition, this progression toward more interactions between blocks also allows for activating the latter mechanisms while the former are being tested.

Testing FTM behavioral models. Models describing the behavior of the individual mechanisms are needed to complement the overall description provided by DRIFTs. Several suitable formalisms have been identified: Petri nets, finite-state machines, control flow graphs, and so on. A common type of behavioral model is a state graph. Its ability to describe simple digital circuits as well as more complex algorithms makes it a privileged model. The associated testing criteria usually considered to guide the selection of the test patterns include state, transition, or path coverage.

Furthermore, we rely on statistical testing to compensate for the imperfect match between the testing criteria and the fault tolerance deficiencies to be uncovered.[15] The difficulty in deciding whether the system output is correct for each test input (known as the oracle problem) is the drawback generally opposed to statistical testing. In the framework of the verification of fault tolerance, this problem is more easily tractable. Indeed, it consists mainly in observing predicates on failure mode assumptions at the FTM inputs and outputs.

We determine the testing profile by transforming the behavioral model of the FTM to be tested into a probabilistic model. Three steps are necessary:

1. *Identification of the independent variables of the behavioral model.* These independent variables constitute the degrees of freedom on which fault injection testing takes place.
2. *Attribution of distributions to these independent variables.* For easier tractability, here, we consider exponential distributions, thus each variable is labeled by its transition rate.
3. *Selection of the optimum values of these transition rates.* This corresponds to maximizing as much as possible the activation of every element of the model as defined by the considered testing criterion (state coverage, transition coverage).

For an example, see the "Formalization" sidebar on the next page.

Fault injection into VHDL models

The interest in developing fault injection techniques for VHDL models is relatively recent. Powerful VHDL simulators are available on the market. However, we still need automated support to efficiently define and run fault injection experiments on a complex VHDL model. This has given rise to work for developing prototype tools aimed at supporting the user in carrying out fault injection test sequences on a VHDL model.[17] MEFISTO,[18] ADEPT,[19] and VERIFY[20] are significant examples of such efforts.

The development of MEFISTO was initiated in cooperation with Chalmers University of Technology (Gothenburg, Sweden) within the framework of the ESPRIT project PDCS. This tool used Synopsys Optium VHDL simulator commands to inject faults in variables and signals of the target VHDL model. Besides ease of implementation, these techniques strongly depend on the simulator features, which led to consideration of more generic injection techniques based on the integration of injection modules into the VHDL model. In a later section, we summarize the main features of this new version of MEFISTO developed at LAAS (MEFISTO-L). Another improved version, using the same principles as the initial tool, but featuring parallel simulation on a network of workstations, was developed at Chalmers (MEFISTO-C).[21]

ADEPT provides an integrated environment that features both analytical techniques and VHDL simulation to support performance and dependability analysis.[19] Fault injection techniques for VHDL were extensively investigated: the studies encompass the characterization of techniques for various levels of abstractions of the simulation. The technique described in DeLong et al.[17] defines a new data type that adds two additional fields (`control` and `mask`) to the usual `data` field of a VHDL signal. The `control` field indicates whether the signal is read from or written to, while the `mask` field specifies the fault model to be applied.

Formalization

Consider the case in which the behavioral model of the FTM is characterized by an asynchronous state machine. The corresponding probabilistic model is a continuous time Markov chain, for which transitions are exponentially distributed. We thus assume the following notation:

- $\Omega = [\omega_j]$ denotes the vector of the rates associated to the input variables of the state machine describing the FTM's behavior.
- $\Lambda(\Omega) = [\lambda_{jk}]$ is the transition rate matrix associated to the Markov chain derived from the state machine; these rates are indeed a function of the elements of the input rate vector.

Let C denote a testing criterion. On a finite-state machine, the two main criteria of interest are relative to the transitions' activation and to the visit of the states. On an ergodic Markov chain, these criteria can be assessed by the mean frequency of firing a transition, and the mean visit frequency of a state, respectively.

Assume that

- $\Pi = [\pi_j]$ is the vector of the steady-state probabilities of the Markov chain,
- ν_{jk} denotes the mean frequency of activation of transition from state j to state k, and
- ν_j denotes the mean frequency of visit (either arrival or departure) of state j,

leads to

$$\nu_{jk} = \pi_j \times \lambda_{jk} \tag{1}$$

and

$$\nu_j = \nu_j^{arr} = \nu_j^{dep} = \sum_{k \neq j} \pi_k \times \lambda_{kj}$$

$$= \pi_j \times \sum_{k \neq j} \lambda_{jk} = \pi_j \times |\lambda_{jj}| \tag{2}$$

In this context, the statistical test corresponding to a given test quality q_T[15] and optimizing the input distribution for the testing criterion considered is thus characterized by two theorems.[16]

Theorem 1. The following relation links the quality of statistical test q_T to the duration of test T

$$\exp(-\nu_{\min} \times T) = 1 - q_T \text{ with } \nu_{\min} = \min(\nu_C)$$

where $\nu_C = f_C(\Pi)$ are the mean frequencies of the activation of the elements of criterion $C(\{\nu_C\} = \{\nu_{jk}\}$ or $\{\nu_j\}$, as defined by Equations 1 and 2). They are functions of steady-state probability vector Π, which solves:

$$\Pi \times \Lambda(\Omega) = 0 \text{ and } \sum_j \pi_j = 1$$

Theorem 2. Solving the nonlinear programming problem, maximize $\nu_{\min} = \min(\nu_C)$, gives us the optimal profile for an asynchronous finite-state machine under constraints $\omega_j \geq 0$, where the ω_j are the rates of the input variables.

We illustrate this formalization by an example of its application to a specific FTM in the Case Study section.

VERIFY relies on an extension of VHDL that defines a comprehensive fault model describing not only classical fault model parameters (for example, fault type and duration), but also the fault occurrence process (mean time to fault occurrence). Libraries of basic components (logic gates) featuring this comprehensive fault model are defined. This makes it possible to directly obtain dependability measures such as MTTF (mean time to failure), in addition to coverage and latency measures. Besides requiring that the individual failure rates be known, the price to pay for this extra refinement results in significant simulation time overhead.

Injection techniques

We can identify three classes of techniques when considering the features of VHDL. It is a

- *Simulation language.* The supporting simulation environments allow the state of a model to be modified during the execution of the simulation. This leads to the development of techniques based on the direct perturbation of the simulation state.
- *Hardware description language.* The injection techniques are derived from classical hardware testing methods, which lead to the perturbation of VHDL signals.
- *Programming language.* Considering soft-

ware mutation-testing techniques leads to the alteration of one (or several) VHDL statements.

Hereafter, we provide a brief description of these techniques; Boué provides a detailed comparison of the advantages and drawbacks.[16] The first class of techniques was implemented in the initial MEFISTO tool, while the current version of MEFISTO-L supports the second. We plan to investigate the mutation technique in further versions of the tool.

Perturbation of the state of the VHDL simulation. VHDL simulators feature built-in commands that allow access to the state of the VHDL model being simulated. To a large extent, the principles used here can be related to the mechanisms applied by the SWIFI technique. (See the "Background" box again.) Some examples of useful capabilities that are most often offered are

- stopping of the simulation (at a given time or upon occurrence of a specific event),
- reading and modifying a VHDL signal or variable, and
- recording a snapshot of the state of the simulation (to carry out comparisons of successive simulation runs or to start the simulation in a predefined state).

Two main types of modifications of the system state can be distinguished:[18] signal and variable manipulation. Signal manipulation is suited for implementing simple fault models (permanent or temporary stuck-at faults) on the structural description of the model. Variable manipulation offers a simple way to inject behavioral faults.

Perturbation of a VHDL signal. A VHDL model is made up of components linked by signals. Fault injection is based on the addition of dedicated fault injection components acting as probes or saboteurs. Both forcing and insertion techniques that are widely used in physical injection[22] have their equivalent here. Figure 4 identifies the five means for connecting a fault injection component to a VHDL signal.

Mutation of VHDL statements. We can describe the behavior of a VHDL component by a series of software instructions very much similar to Ada. The possible alterations are as diverse as the ones encountered in the case of software mutation testing.[23,24] Armstrong, Lam, and Ward[25] identifies eight types of alterations. We can also consider more sophisticated types of alterations, in particular, that of the syntactic tree of the VHDL model.[26] Rules are proposed for the modification of tree branches (for example, permutation of the clauses `then` and `else` in an `if-then-else` construct or the substitution of an operator by another).

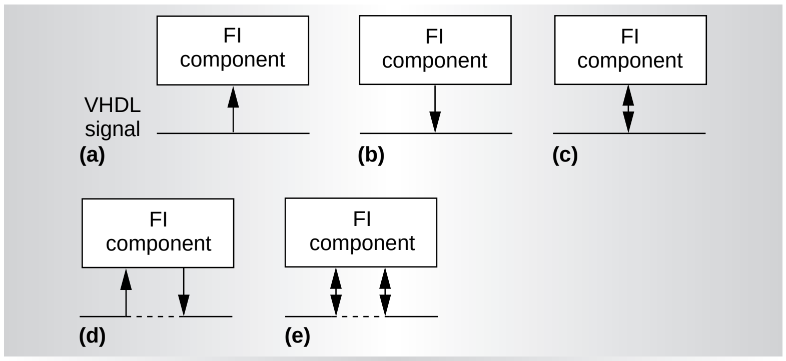


Figure 4. The five means to connect a fault injection component to a VHDL signal: passive derivation, a probe (a); active derivation, a saboteur (b); bidirectional derivation (c); simple insertion, and bidirectional insertion (d). These last three are both probes and saboteurs.

MEFISTO-L

We considered three main guidelines for the design of MEFISTO-L:

- support both fault forecasting and fault removal objectives, and thus contribute objectively to the testing framework depicted earlier,
- offer portable fault injection capabilities (independent of the underlying VHDL simulator), and
- implement rather simple fault models that are representative of the VHDL descriptions. This has led to a preference for the signal perturbation technique mentioned earlier in this section.

We only focus here on a brief description of the tool and on issues related to the support of the testing strategy; refer to Boué, Pétillon, and Crouzet[27] for a more detailed description of the tool's features.

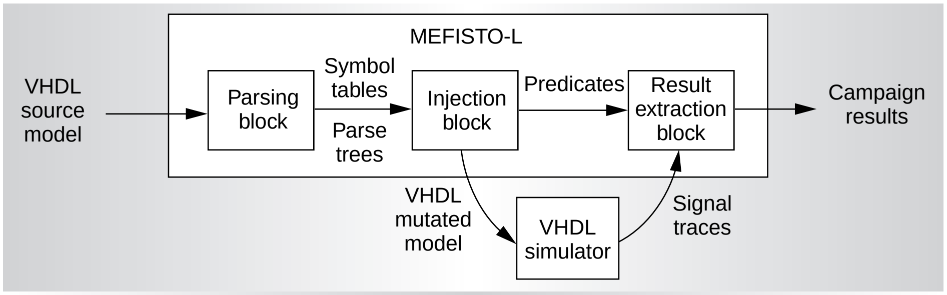


Figure 5. Architecture of MEFISTO-L.

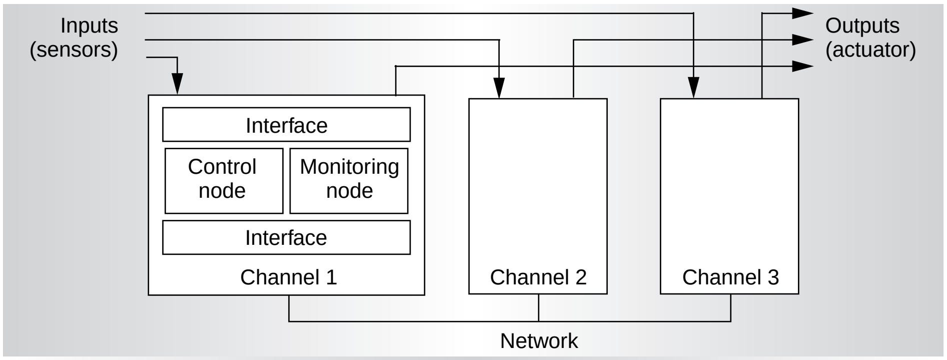


Figure 6. The target fault-tolerant distributed system.

Three distinct functional blocks make up MEFISTO-L: parsing, injection, and result extraction (Figure 5).

The parsing block extracts the data needed for the injection campaign from the VHDL source code of the target model. The injection block deals with the whole specification of the campaign, and the generation of the mutated model. The tool automatically generates a mutated model by manipulating the different parse trees of the source model. All probes and saboteurs necessary to carry out the injection experiments defined by the testing strategy are incorporated into the model; accordingly, only one compilation is necessary. The fault injection campaign is then carried out by executing the mutated model with any standard VHDL simulator. We are currently using the Optium simulator from Synopsys. The result extraction block analyzes the traces obtained from the simulation of the mutated model to produce the results of the campaign.

Case study

To assess the usefulness of the proposed testing strategy, in particular when using MEFISTO-L, we developed a VHDL model of a simple fault-tolerant system. This target system gathers several FTMs issued from actual fault-tolerant systems for a control process in industrial or embedded real-time applications (Figure 6).

The architecture features three channels interconnected by a network supporting a time-slice protocol.[28] The system interfaces with the controlled process via redundant sensors (one for each channel) and a single actuator incorporating analogical voting. The behavior of the target system is organized in cycles during which sensor sampling, median voting on sensor values, computing of the regulation function, and output emission are scheduled.

Each channel is designed to be fail-silent:[29] in case of an error detection, the channel should extract itself from the network and wait for repair. More precisely, each channel implements a dual-node architecture, adapted from Brière and Traverse.[30] Each node contains a processing unit (PU) and two interfaces (to the controlled process and to the other channels via the network). Each processing unit is described both by a behavioral model (algorithm of the protocol of execution) and by a structural model (microprocessor, memory, decoding logic, and interface). The microprocessor is the DP32[31] already considered in various related studies.[18,20] It is activated by a program describing a low-level execution protocol written in assembly language. Except for their identification number, all channels are identical. Each channel is connected to an independent sensor, and all are connected to the actuator. The corresponding VHDL model represents about 2,500 lines of code, divided in 15 entities, 16 architectures, 8 configurations, and 2 packages (plus `STD` and `IEEE` packages).

We applied the testing strategy to the target system using 1) DRIFTs to determine the main interactions between the FTMs, and 2) behavioral models of each elementary FTMs

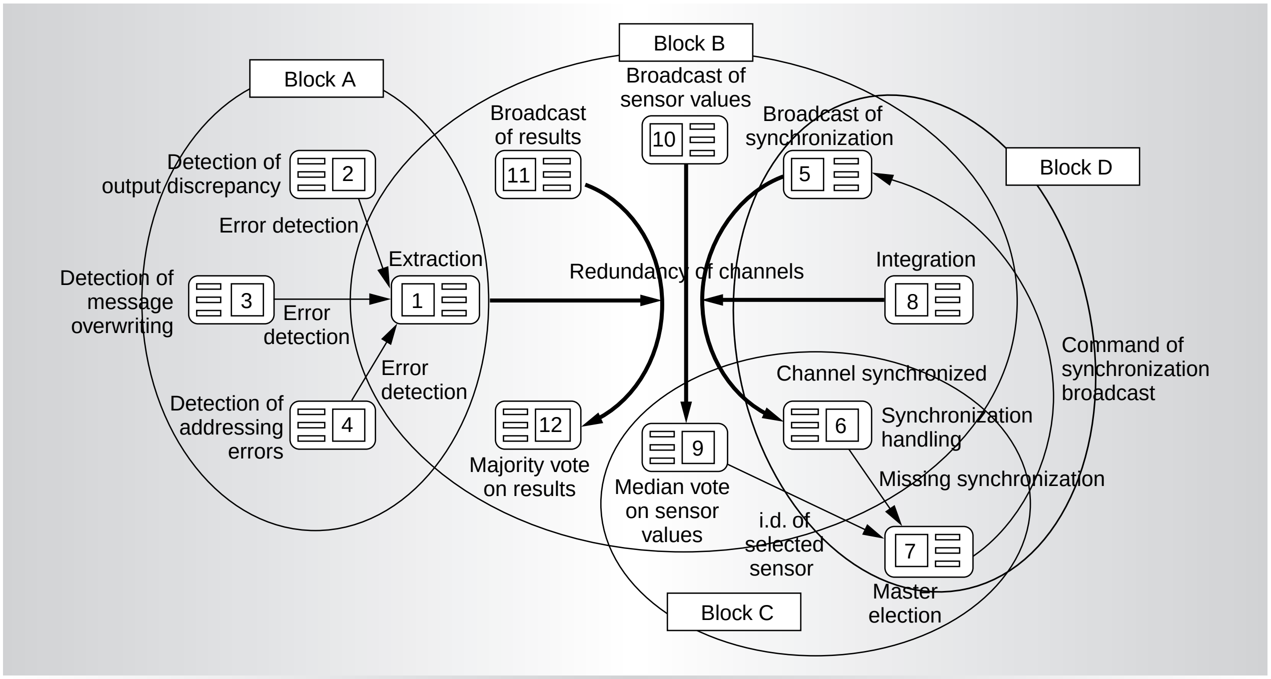


Figure 7. DRIFT of the target system.

to derive the suited test patterns (fault and activity). So far, the study of simple error detection mechanisms (for example, a detector of loss of a queued message received via the network—message overwriting) has shown the pertinence and feasibility in that context of the application of statistical testing on state machine graphs. Indeed, the application of the strategy helped reveal a deficiency in the design of the target VHDL model.[16]

Modeling the fault tolerance in the target system

Figure 7 shows the DRIFT of the target system. The FTMs are grouped into four blocks:

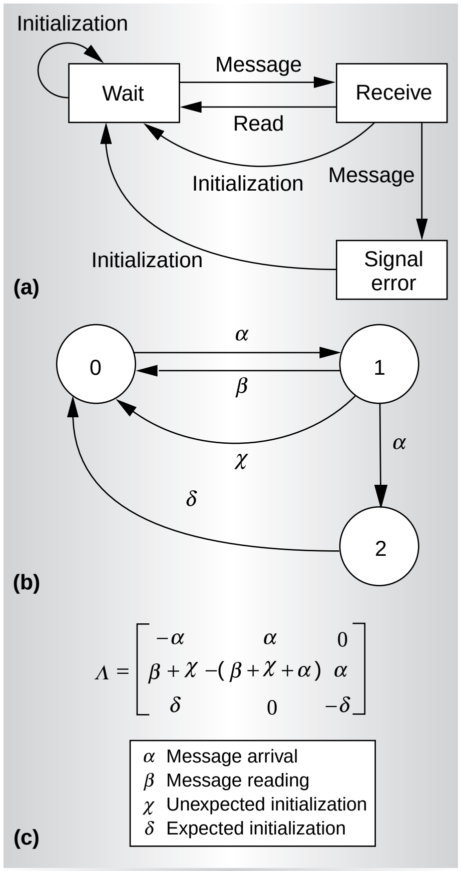
- A: error detection and extraction of a channel;
- B: redundancy of the channels, involving several broadcasts and associated voting;
- C: master election; and
- D: synchronization of the channels and group membership protocol.

The decomposition of the fault tolerance into blocks is a natural step that aims at grouping the FTMs performing a common function or sharing the same redundancy. This decomposition provides a means for structuring the testing process. The fact that blocks overlap indicates that unit testing of the individual FTMs is not sufficient and must be complemented by additional verifications carried out on a more global level.

Block A aims at implementing the fail-silence property. Toward this end, each channel includes a set of error detection mechanisms aimed at disconnecting the channel upon the signaling of an error. The block is composed of four FTMs (see Figure 7):

- FTM 1 gathers the intrachannel error detection signals and causes the extraction of the channel,
- FTM 2 is a comparator for checking the value calculated by each node and the output delivered,
- FTM 3 is a detector of loss (overwriting) of a queued message received via the network, and
- FTM 4 is a detector for illegal addressing of the microprocessor peripheral devices.

Block B refers to the hardware redundancy of a channel. It is composed of eight mechanisms: two FTMs for activating and inhibiting (extracting) the channel, three in charge of broadcasting different kinds of data, and



$$\Lambda = \begin{bmatrix} -\alpha & \alpha & 0 \\ \beta + \chi & -(\beta + \chi + \alpha) & \alpha \\ \delta & 0 & -\delta \end{bmatrix}$$

α	Message arrival
β	Message reading
χ	Unexpected initialization
δ	Expected initialization

Figure 8. Behavioral and probabilistic models of the overwritten message detector.

three associated voting mechanisms for selecting the “correct” value among those provided by the active channels. The extraction mechanism was already presented in the previous paragraph as FTM 1. The other FTMs are

- FTM 8, which supports the integration of one channel into the group of active channels; this function is activated at the beginning of every cycle (that is, a new group is formed at each cycle);
- FTM 10, which broadcasts the values of the sensors acquired by each channel;
- FTM 9, which carries out a median vote on the inputs read from the sensors by each channel;
- FTM 11, which broadcasts the results processed by each channel to be delivered to the actuator;
- FTM 12, which carries out a majority vote on the outputs to be delivered to the actuator;
- FTM 5, which provides synchronization signals; and
- FTM 6, which processes these signals to provide relevant inputs to the integration (FTM 8) and master election (FTM 7) mechanisms.

Block C elects the master channel. C is composed of three FTMs of which FTM 6 and FTM 9 have already been presented. FTM 7 elects the master channel for the next cycle by collecting the data provided by the median vote (FTM 9) and by the synchronization handling (FTM 6).

Block D is in charge of synchronizing the channels and of their integration at the beginning of each cycle. D is composed of four mechanisms that also belong to blocks B or C (or both).

Overwritten message detection mechanism models

Figure 8 depicts the behavioral and probabilistic models of the FTM aimed at detecting message overwriting.

When idle, the mechanism awaits the arrival of a message. When a message is received, the interface warns the processing unit by an interruption signal, and the mechanism moves to the receive state. The processing unit then reads the message received, and the mechanism returns to the wait state. An error is detected when a second message is received before the first has been read by the processing unit (initial message has been overwritten). The detector then enters the signal error state that activates the error detection signal. An initialization of the channel is necessary for the channel to quit the error state. An unexpected initialization also makes the detector return to the wait state.

As message arrival frequency a is under control of the overall protocol, we identify only three independent variables for fault injection:

- message-reading frequency β,
- unexpected initialization frequency χ, and
- expected initialization frequency δ.

The operational profile of the input events of the overwritten message detector is characterized by the rates shown in the left column of Table 2. The selected testing profile aims at maximizing the transition activation. The resolution of the optimization problem (see the "Formalization" box) leads to a profile that reduces the message-reading frequency and increases the unexpected initialization. The values obtained appear in the right column of Table 2.

Table 2. Operational and testing profiles.

Operational profile (nominal)		Testing profile
8 messages during each 750-ms-long processing cycle	$\alpha \approx 10$ kHz	$\alpha \approx 10$ kHz
1 message reading within 1 ms following its arrival	$\beta \approx 1$ MHz	$\beta \approx 10$ kHz
0 unexpected initialization	$\chi \approx 0$ Hz	$\chi \approx 10$ kHz
1 expected initialization within 1 ms following an error signaling	$\delta \approx 1$ MHz	$\delta \approx 1$ MHz

Placing a saboteur on the initialization signal of the target channel ensures the modification of the initialization rates. This saboteur is synchronized with the state of the overwritten message detector so as to exhibit a different behavior in a nominal case (unexpected initialization) and after error detection (expected initialization). This is achieved by placing a probe on the error detection signal of the overwritten message detector. The saboteur thus generates pulses exponentially distributed, with a rate of 10 kHz or 1 MHz, according to the value of the error detection signal.

Placing a saboteur on the interruption signal of the overwritten message detector indirectly modifies the message reading rate. Indeed, the direct perturbation of the read signal could affect more peripheral devices than the network interface. On the contrary, filtering the interruption signal affects only the overwritten message detector. The saboteur is thus in charge of simulating a delay before a message is actually read by the processing unit.

The simulation of the platform with the operational profile did not exhibit a deficiency, even with test lengths corresponding to good test qualities q_T. (See "Formalization" box.) However, the simulation with the testing profile maximizing the activation of the transitions in the Markov chain describing the target FTM revealed one fault tolerance deficiency. This fault is manifested by a crash of the simulation environment caused by a runtime error of the VHDL model. The Synopsys Optium simulator issues the following error message: `**Runtime Error: Negative time in process timeout.`

Diagnosis, carried with the aid of debugging facilities of the simulation environment, showed that bad handling of the processing unit's initialization caused this failure. The VHDL process modeling this unit calls procedures that are insensitive to an unexpected initialization. Thus some state and timing variables could not be updated. Used in a `WAIT` instruction, they give a negative time in the process timeout, which is forbidden in VHDL. Thus, in summary, the test has revealed that a transition in the behavioral model of the overwritten message detector is absent in the VHDL simulation model.

As illustrated by our case study, the application of the proposed testing strategy has helped reveal a deficiency in the design of the VHDL model describing the fault-tolerant architecture used as a target system. Nevertheless, further work is needed. From the conceptual point of view, we believe that the use of a more formal representation of the FTMs would help address the high-level and detailed aspects of the testing strategy at the same time. On a more practical aspect, we are adding new fault injection capabilities to MEFISTO-L. These include the ability of injecting faults in other VHDL objects than signals. The most promising target is constituted by the variables of sequential descriptions in VHDL. MICRO

Acknowledgments

ESPRIT projects 6362 (PDCS: Predictably Dependable Computing Systems) and 20072 (DeVa: Design for Validation) partially supported our work. We acknowledge the essential contributions to the early phases of this work made by Eric Jenn (now with Technicatome, Aix-en-Provence, France), and by our colleagues at Chalmers University of Technology, Gothenburg, Sweden: Johan Karlsson, Joachim Ohlsson (now with SAAB Ericsson Space, Gothenburg), and Marcus Rimén (now with Carlstedt Research & Technology, Gothenburg). Our thanks go also to Philippe Pétillon (now with Aerospatiale Matra Airbus, Toulouse, France) and to Jean-Etienne Doucet at LAAS-CNRS, for their key role in the development of MEFISTO-L.

References

1. J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," *Proc. 25th Int'l Symp. Fault-Tolerant Computing (FTCS-25),* Special Issue, IEEE CS Press, 1995, pp. 42-54.
2. K. Echtle and Y. Chen, "Evaluation of Deterministic Fault Injection for Fault-Tolerant Protocol Testing," *Proc. 21st Int'l Symp. Fault-Tolerant Computing (FTCS-21),* 1991, IEEE Computer Society Press, Los Alamitos, Calif., pp. 418-425.
3. D. Avresky et al., "Fault Injection for the Formal Testing of Fault Tolerance," *IEEE Trans. on Reliability,* Vol. 45, 1996, pp. 443-455.
4. J. Christmansson and P. Santhaman, "Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms—Criteria for Error Selection Using Field Data on Software Faults," *Proc. Seventh Int'l Symp. Software Reliability Engineering (ISSRE'96),* 1996, IEEE CS Press, pp. 175-184.
5. T.K. Tsai, R. K. Iyer, and D. Jewitt, "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems," *Proc. 26th Int'l Symp. on Fault-Tolerant Computing (FTCS-26),* 1996, IEEE CS Press, pp. 314-323.
6. T. Tsai et al., "Path-Based Fault Injection," *Proc. Third ISSAT Int'l Conf. Reliability and Quality in Design,* 1997, pp. 121-125.
7. K. Echtle, "Safety Testing by Fault Injection," *Proc. Eighth European Workshop on Dependable Computing (EWDC-8),* Chalmers Univ., Gothenburg, Sweden, 1997.
8. G.S. Choi and R.K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis," *IEEE Trans. Computers,* Vol. 41, Dec. 1992, pp. 1515-1526.
9. C.R. Yount and D.P. Siewiorek, "A Methodology for the Rapid Injection of Transient Hardware Errors," *IEEE Trans. Computers*, Vol. 45, Aug. 1996, pp. 881-891.
10. K.K. Goswami, R.K. Iyer and L. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis," *IEEE Trans. Computers*, Vol. 46, Jan.1997, pp. 60-74.
11. J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Trans. Software Engineering,* Vol. 24, Feb. 1998pp. 125-136.
12. M. Kaâhniche et al., "A Hierarchical Approach for Dependability Analysis of a Commercial Cache-Based RAID Storage Architecture," *Proc. 28th Int'l Symp. Fault-Tolerant Computing (FTCS-28),* IEEE CS Press**,** 1998, pp. 6-15.
13. D.T. Stott et al., "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection," *IEEE Trans. Computers,* Vol. 47 Jan. 1998, pp. 108-119.
14. *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1993, IEEE, Piscataway, N.J., 1993.
15. P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "Software Statistical Testing," *Predictably Dependable Computing Systems,* B. Randell et al., eds., Springer Berlin, 1995, pp. 253-272.
16. J. Boué, *Fault Tolerance Testing by Means of Fault Injection in VHDL Simulation Models,* doctoral dissertation, National Polytechnic Inst., Toulouse, France, 1997. (Also LAAS Report 97-503 in French).
17. T.A. DeLong, B.W. Johnson, and J.A. Profeta III, "A Fault Injection Technique for VHDL Behavioral-Level Models," *IEEE Design & Test of Computers,* Vol. Winter, 1996, pp. 24-33.
18. E. Jenn et al., "Fault Injection into VHDL Models: The MEFISTO Tool," *Proc. 24th Int. Symp. Fault-Tolerant Computing (FTCS-24),* IEEE CS Press, 1994, pp. 66-75.
19. A. Ghosh, B.W. Johnson, and J.A. Profeta III, "System-Level Modeling in the ADEPT Environment of a Distributed Computer System for Real-Time Applications," *Proc. Int'l Computer Performance and Dependability Symp. (IPDS'95),* IEEE CS Press, 1995, pp. 194-203.
20. V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions," *Proc. 27th Int'l Symp. Fault-Tolerant Computing (FTCS-27),* IEEE CS Press,1997, pp. 32-36.
21. P. Folkesson, S. Svensson, and J. Karlsson, "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection," *Proc. 28th Int'l Symp. on Fault-Tolerant Computing (FTCS-28),* IEEE CS Press, 1998, pp. 284-293.
22. J. Arlat et al., "Fault Injection for Dependability Validation—A Methodology and Some Applications," *IEEE Trans. Software Engineering*, Vol. 16, Feb. 1990, pp. 166-182.
23. R.A. DeMillo, R.J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* Apr.

1978, pp. 34-41.

24. Y. Crouzet, P. Thévenod-Fosse, and H. Waeselynck, "Validation of Software Testing by Fault Injection: The SESAME Tool," *Proc. 11th Conf. Reliability and Maintainability,* 1998, pp. 551-559. (*SEE,* in French)
25. J.R. Armstrong, F.-S. Lam, and P.C. Ward, "Test Generation and Fault Simulation for Behavioral Models," *Performance and Fault Modelling with VHDL,* J.M. Schoen, ed., Prentice-Hall, Englewood Cliffs, N.J., 1992, pp. 240-303.
26. E. Jenn, *On the Validation of Fault-Tolerant Systems: Fault Injection in VHDL Simulation Models,* doctoral dissertation, National Polytechnic Inst., Toulouse, France, 1994. (Also LAAS Report 94-361 in French).
27. J. Boué, P. Pétillon, and Y. Crouzet, "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance," *Proc. 28th Int. Symp. Fault-Tolerant Computing (FTCS-28),* IEEE CS Press,1998, pp. 168-173.
28. H. Kopetz, "The Time-Triggered Approach to Real-Time System Design," *Predictably Dependable Computing Systems,* B. Randell et al., eds., Springer, Berlin, 1995, pp. 53-66.
29. D. Powell et al., "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," *Proc. 18th Int'l Symp. Fault-Tolerant Computing Systems (FTCS-18)*, IEEE CS Press, 1988, pp. 246-251.
30. D. Brière and P. Traverse, "AIRBUS A320/A330/A340 Electrical Flight Controls—A Family of Fault-Tolerant Systems," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing (FTCS-23),* IEEE CS Press, 1993, pp. 616-623.
31. P.J. Ashenden, "The VHDL Cookbook," tech. report., Univ. of Adelaide, South Australia, 1990.

Jean Arlat is a member of the Dependable Computing and Fault Tolerance (TSF) Group at LAAS-CNRS, Toulouse, France. He also leads the Laboratory for Dependability Engineering (LIS), a cooperative laboratory hosted by LAAS, for TSF researchers and representatives from five French industrial companies. His research interests focus on the evaluation of hardware-and-software fault-tolerant systems including both analytical modeling and experimental fault injection approaches. Arlat received the Certified Engineer degree from the National Institute of Applied Sciences of Toulouse, the Doctor in Engineering and the Docteur ès-Sciences degrees from the National Polytechnic Institute of Toulouse. He currently chairs the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance. He has cochaired the Program Committee of the 28th IEEE Symposium on Fault-Tolerant Computing held in Munich and chaired the IEEE Computer Society's Technical Committee on Fault-Tolerant Computing. He is a member of the ACM, the IEEE, and the SEE Design and Validation for Dependability Group.

Jérome Boué works at COFRAMI as a dependability expert for the aerospace industry and is currently involved in the development of critical software for the European scientific module of the International Space Station. He joined the Dependable Computing and Fault-Tolerance (TSF) Group at LAAS-CNRS to prepare his master's and Doctorate degrees in computer science. His main research interests during this period were statistical software testing and its application on fault tolerance testing by means of fault injection in VHDL simulation models. Boué received his Doctorate from the National Polytechnic Institute of Toulouse.

Yves Crouzet is a member of the Dependable Computing and Fault-Tolerance (TSF) Group at LAAS-CNRS. He works on fault tolerance of human errors and has worked on the design and realization of self-checking VLSI circuits. His research interests have concerned the experimental validation of dependable systems by fault injection and the experimental validation of software testing methods by mutation analysis. Crouzet received the Certified Engineer degree from the Higher National School of Electronics, Electrical Engineering, Computer Science and Hydraulics of Toulouse and the Doctor in Engineering from the National Polytechnic Institute of Toulouse. He is a member of SEE Design and Validation for Dependability Group.

Direct questions concerning this article to Jean Arlat, LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse Cedex 4, France; arlat@laas.fr.