EARLY VALIDATION OF DEPENDABLE SYSTEMS BY FAULT INJECTION INTO VHDL MODELS*

Jean Arlat

arlat@laas.fr

LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 4, France

Abstract: The early validation (encompassing both testing and evaluation) of the fault tolerance mechanisms is an essential task in the design of dependable computing systems. Towards this ends, the presentation focuses on the integration of the fault injection methodology within the design process of fault-tolerant systems. Due to its wide spectrum of application and hierarchical features, VHDL has been selected as the simulation language to support such an integration. Although most fault injection works address the evaluation issue, we focus rather in this paper on the testing issue. After having identified and discussed the possible means to inject faults in a VHDL model, we propose a fault injection-based testing strategy that can be explicitly integrated into the development process of a fault-tolerant system; special emphasis is put on: (i) the determination of the test inputs (fault and activity sets), and (ii) the characterization of the observations to be made to fully specify the fault injection experiments. We subsequently describe the main features of the MEFISTO-L (Multi-level Error/Fault Injection Simulation TOol developed at LAAS) environment supporting the proposed strategy. Finally, some preliminary results obtained when applying the testing strategy with MEFISTO-L are presented and discussed, along with the currently on-going work.

1. Introduction

Fault injection, in which the observation of the system behavior in presence of faults is explicitly forced by introduction of faults, has emerged as an invaluable means to support the dependability validation of fault-tolerant systems (e.g., see [1-3]).

Two main trends characterize most recent work on fault injection: (i) apply fault injection as early as possible in the design process of fault-tolerant systems, i.e., into simulation models of the target system [4-8] and (ii) when dealing with the implementation of the target system, favor software-implemented fault injection (SWIFI), i.e., based on the mutation of the executing software or of the data [9-13].

Classically, simulation-based fault injection approaches address different abstraction levels by using distinct description languages. Clearly, a coherent environment should be provided: (i) to favor interoperability between the successive abstraction levels and (ii) to integrate the validation in the design process. SWIFI is primarily motivated to avoid the difficulties and cost inherent to the implementation of physical fault injection approaches, such as pin-level or heavy-ion, (e.g., see [14]). It is also intended to cover errors that can be generated by both software and hardware faults. However, this is only achieved to some extent. Still, further work is needed to validate such abstract error models before it can be confidently considered that SWIFI alone can adequately embrace the consequences of hardware faults.

Another important characterization of the work on fault injection has to deal with the objective of the fault injection experiments, either fault removal or fault forecasting (e.g., see [15, 16]).

---

* This work was partially supported by the ESPRIT Projects: *Predictably Dependable Computing Systems* (PDCS-2 — 6362) and *Design for Validation* (DeVa — 20072).

Concerning the *fault removal* objective, fault injection is explicitly aimed at reducing, by *testing*, the presence of faults in the design/implementation of the fault tolerance mechanisms (FTMs) whose consequences would be *deficiencies* in their expected behavior when faced with faults they are explicitly intended to handle: faults are injected to uncover such potential *fault tolerance deficiencies faults* (in short, *ftd-faults*) and thus, to determine the most appropriate actions for improving the FTMs. In the case of *fault forecasting*, the main issue is to rate, by *evaluation*, the *efficiency* of the *operational behavior* of the FTMs: this type of experiment is aimed mainly at providing estimates for the parameters that usually characterize the operational behavior of the error processing and fault treatment, that is, *coverage* and *latency*.

Besides a few exceptions (e.g., see [17-20]), until recently, fault injection has been essentially used for evaluating coverage and/or latency of error detection (e.g., see [21-23]) or directly dependability measures (e.g., see [8, 24]). The problem of the systematic testing of fault tolerance by means of fault injection still remains a challenge. Indeed, several important aspects related to such an objective concern:

- conducting more efficiently fault injection experiments that increases the sensitization of the fault tolerance mechanisms,
- developing a method for the determination of the fault-activity sets to be applied in a fault injection campaign,
- allowing for the removal of fault tolerance deficiency faults,
- implementing and assessing a supporting prototype environment.

The work summarized in this paper attempts to contribute to these issues. It is part of a research effort initiated within the framework of the ESPRIT project PDCS2 (Predictably Dependable Computing Systems). The supporting tool MEFISTO-L, that is currently being developed at LAAS, is the second prototype of MEFISTO (Multi-level Error/Fault Injection in Simulation Tool) that was developed jointly with Chalmers University of Technology [5]. Fault forecasting of fault tolerance was the main objective of this first prototype. While this objective is also supported by MEFISTO-L, the main functional enhancements concern the provision of capabilities specifically aimed at facilitating the removal of ftd-faults, thus implementing the method proposed for fault tolerance testing by means of fault injection into VHDL simulation models.

The remainder of this paper is composed of four sections. Section 2 describes and analyzes the techniques that have been identified for injecting faults into VHDL models. Section 3 presents the global framework for testing fault tolerance. Section 4 gives an overall description of MEFISTO-L in terms of its objectives and main functional blocks. Section 5 briefly discuss the preliminary results obtained by applying the proposed testing strategy and concludes the paper by giving perspectives for the further development of MEFISTO-L.

2. Fault Injection into VHDL Models

The development of an integrated and consistent design environment for fault-tolerant systems based on a single language seems to be achievable when considering the emergence of hardware description languages. With this respect, VHDL has been identified as a suitable language as it presents many useful features:

- 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 [25],
- recognition as a viable framework (i) 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 [26], and (ii) for supporting hybrid (i.e., mixed abstraction levels) simulation models [27],
- capability to support testing activities [28].

Two categories of fault injection techniques can be identified to inject faults into VHDL models: (i) using the built-in commands of the VHDL simulator, or (ii) modifying the VHDL model.

2.1. Using the Built-in Commands of the VHDL Simulator

The main reason for using the built-in commands of the simulator for fault injection is that this does not require the modification of the VHDL code; however, the applicability of these techniques depends strongly on the functionalities offered by the command languages of the simulators. Two techniques based on the use of simulator commands have been identified and experimented: *signal* and *variable* manipulation. These elements characterize respectively structural and behavioral features of a VHDL model.

VHDL Signal Manipulation. In this technique, faults are injected by altering the value of the *signals* that are used to link the *components* that made up the VHDL model. For example, for a temporary stuck-at fault, this can be achieved simply by simulating the system until the time of injection is reached and all processes have stopped at a `WAIT` statement, then the signal is disconnected from its driver(s)[1] and forced to a new value. The system is simulated for the duration of the fault, and, finally, the signal's driver(s) is (are) reconnected, and the normal simulation is resumed until the end of the observation time.

VHDL Variable Manipulation. This technique allows injection of faults into behavioral models by altering values of variables defined in VHDL processes. In the simulation phase, the execution of the sequential code in a process takes no time as viewed by the simulator since time is incremented only when a process stops at a `WAIT` statement.

2.2. Modifying the VHDL Model

In this category, two techniques can be distinguished. The first one is based on the addition of dedicated fault injection (FI) components acting as *probes* or *saboteurs*. The second one is based on the *mutation* of existing component descriptions in the VHDL model, which generates modified component descriptions called *mutants*. Here again, these techniques apply respectively to either structural or behavioral features of the target VHDL model.

Insertion of Probes/Saboteurs. Five main means of connecting an FI component to a signal can be identified (Figure 1).

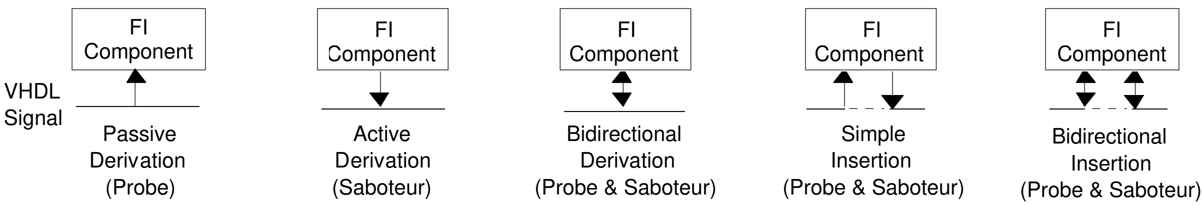

Figure 1 - The five means to connect a FI component to a signal

The passive derivation has only an observation role (the FI component simply acts as a *probe*). A probe can be used to collect data on the behavior of the target system (e.g., for monitoring error detection signals). The active derivation allows for a *resolved signal* to be altered (the FI component acts as a *saboteur*). A *saboteur* may alter the value or timing characteristics of one (or several) signal(s) when activated. It is usually inactive during normal system operation and activated only to inject a fault. The bi-directional derivation scheme implements both the probing and sabotage features. Indeed, the association of a probe to a saboteur may be useful to precisely control the fault injection process.

The insertion mode is more complex to handle. Indeed, the original signal has to be cut into two pieces and it follows that the placement of the saboteur on the signal will impact the resulting fault model being implemented: next to an emitter, the saboteur will affect all the receivers, next to a

---

[1] VHDL allows for a signal to have many *drivers* (signal sources), provided that a *resolution function* is supplied to resolve the values generated by the multiple sources into a single value for the signal; such a signal is called a *resolved signal*; it can be used to model buses and wired-and logic.

receiver, only the latter will be affected. The bi-directional case is even more difficult to implement, in particular due to the possible loops that can be created by the alternating emitter/receiver mode of the component that are connected to the target signal.

Mutation. A *mutant* is a component description that replaces another component description. When inactive, it behaves as the component description it replaces and, when activated, it mimics the component's behavior in presence of faults. It is easy to implement this replacement technique in VHDL using the configuration mechanism.

Here we consider that any change made to an existing component description is regarded as a mutation; thus, mutation encompasses the integration of fault injection components (see previous paragraph), but also includes the following forms of alterations:

- mutating structural component descriptions by replacing subcomponents (e.g., a `NAND`-gate replaced by a `NOR`-gate),
- automatically mutating statements in behavioral component descriptions, e.g., by generating wrong operators or exchanging variable identifiers; this is similar to the mutation techniques used by the software testing community,
- manually mutating behavioral component descriptions to achieve complex and detailed fault models.

Both signal and variable manipulations (cf. Section 2.1) can be used for controlling, i.e., activating and deactivating, probes, saboteurs and mutants. In this way, the injection of faults can be controlled by the built-in commands of the simulator also when mutants, probes and saboteurs are used. This can be achieved by means of global signals or shared global variables as supported by the most recent version of VHDL, called VHDL'93 [29].

2.3. Comparison of the Techniques

The fault injection techniques considered are compared first in terms of three main criteria: fault modeling capacity, effort required for setting up an experiment and simulation time overhead; more details on this comparison can be found in [30]. Then, a more focused comparison of the techniques using the built-in commands of the simulator will be presented on the basis of the results of experiments carried out using the first prototype tool MEFISTO on two VHDL models (structural and behavioral) of a simplified 32-bit processor. The technique allowing for the direct alteration of the signals as supported by MEFISTO-L is currently being evaluated (see Section 4).

Fault Modeling Capacity. Mutants can be designed using the full strength of the VHDL language, and are thus well suited for implementing any behavioral and structural fault models provided they can be expressed within the VHDL semantics. This is also the case for the probes and saboteurs, although they have a very restricted view of the system due to their limited number of input and output ports. Signal manipulation is suited for implementing simple fault models (e.g., permanent or temporary stuck-at faults). Variable manipulation offers a simple way for injecting behavioral faults.

Effort Required for Setting up an Experiment. The signal and variable manipulations do not require any modification of the VHDL code, whereas much effort is needed for probes, saboteurs, and mutants as they require (i) creation and generation of probes, saboteurs and mutants, (ii) inclusion of probes, saboteurs and mutants in the model and (iii) recompilation of the VHDL model. The creation of probes and saboteurs, and the automatic generation of mutants are relatively easy tasks, provided that simple fault models are considered. It is also worth noting that probes and saboteurs are reusable components, while a mutant has to be specifically generated for each target component; on the other hand, the inclusion of probes and saboteurs requires the modification of the component description while mutants are easily included in the model by means of the VHDL `configuration` mechanism.

Simulation Time Overhead. Signal and variable manipulations impose a model-independent overhead due to the fact that the simulation has to be stopped and started again for each fault injected. The simulation time overhead imposed by probes, saboteurs and mutants depends on several factors such as (i) the amount of additional generated events (signal changes), (ii) the amount of code to execute per event (e.g., a complex behavioral mutant may require many statements to be executed per event), and (iii) the complexity of the injection control.

Comparison of signal and variable manipulation techniques. The target system considered for this study is a model of a very simple 32 bits processor, the DP32, described in [31], for which two architectures, one behavioral and one structural, are available. In either cases, a set of single faults were randomly injected on the basic elements of the considered architecture: transient faults (one clock cycle) on the internal signals for the structural model, bit-flips on the variables, for the behavioral model. All output ports of the DP32 (i.e., *address bus*, *data bus*, *read*, *write*, *fetch*) were chosen as observation points for the fault injection experiments to identify error occurrences. Although the type of errors observed on these output ports (data and control flow errors) did match for the workloads considered ("Heapsort" and "Bubblesort" programs), the latency of these errors did vary significantly (Figure 2).

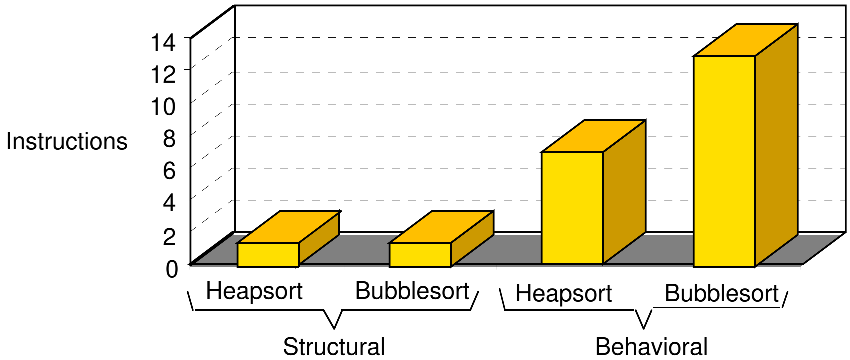

Figure 2 - Comparison of the error latency

The figure shows the mean latency observed (expressed in number of instructions executed) after the fault has been injected before an error appears on the output ports of the processor. For both workloads, it was shown that the latency is mainly determined by the access delay to the faulted memory elements that was found to be shorter for errors caused by signal manipulation faults than variable manipulation faults [5]; basically, the probability that error affects a register likely to be reused (read) is higher in the first case. This shows that the dynamic behavior has not to be neglected when identifying equivalent error classes at different abstraction levels of representation.

3. A Framework for Testing Fault Tolerance

The objective of simulation-based fault injection is to assess (verify and evaluate) fault tolerance at the early steps of the development process. The proposed framework aims at integrating fault tolerance testing activities together with system design activities. The aim is to identify relevant testing criteria for activating fault tolerance and adequate readouts for observing test outcomes, that will guide the specification of fault injection experiments. These fault injection experiments are then applied with MEFISTO-L to a simulation model of the fault-tolerant system to adequately exercise the fault tolerance mechanisms.

Figure 3 depicts the framework proposed for testing fault tolerance and its links with the development process. Further details can be found in [32].

Around the steps devoted to the design process — dependability requirements (a), global specification (b), detailed specification (c), simulation model (d) — the two main following activities for the testing of fault tolerance mechanisms take place: test pattern generation and test diagnosis.

3.1. Test pattern generation

A general scheme of the organization of fault tolerance of the system (e) is derived from the global specification (b). It mainly makes an inventory of the various fault tolerance mechanisms and their interactions; it thus forms a Diagram of the Redundancies and Interactions of the Fault

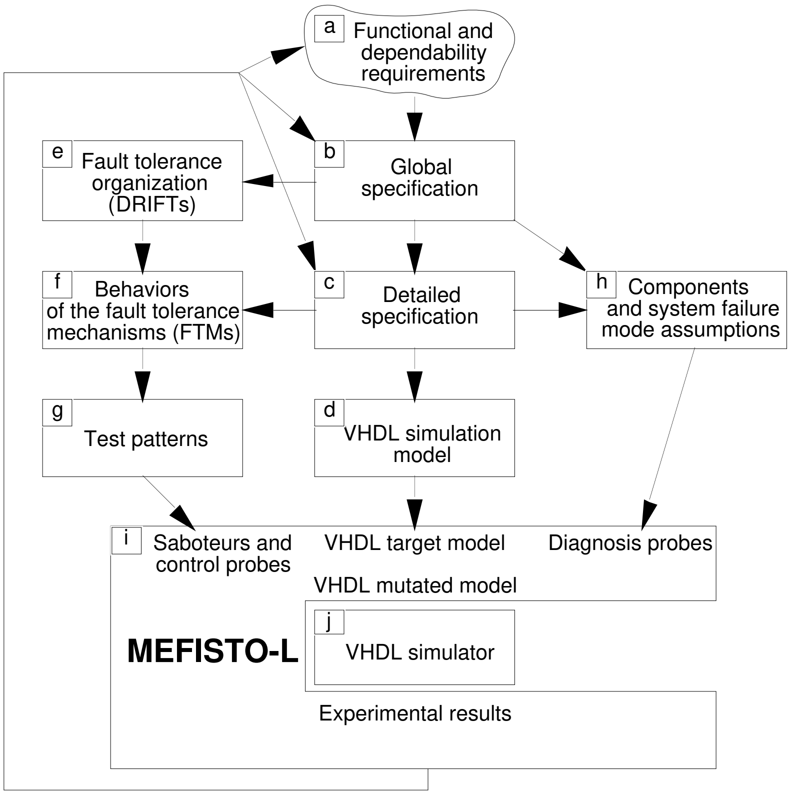

Figure 3 - The framework for testing fault tolerance with MEFISTO-L

Tolerance (DRIFT, for short) [33]. This scheme provides a data flow representation of fault tolerance. The concise and explicit identification of what are the fault tolerance mechanisms and their interactions provided by the DRIFT facilitates the elaboration of the overall testing strategy.

The behavior of the individual fault tolerance mechanisms (f) identified in the DRIFT can be derived from the refinement of the specification (c). This behavior can be described by means of models such as Petri nets, algorithms, logical expressions, etc. They are used to identify the relevant testing criteria whose coverage is the condition for deriving the test patterns (g), encompassing both activity (system inputs and states) and fault spaces, for assessing their efficiency.

As for software, statistical testing (e.g., see [34]) has been also identified as a valuable means for generating test patterns due to its ability to produce easily a high number of test cases, thus to overcome the imperfection of testing criteria with respect to the design faults to be removed (i.e., the ftd-faults here).

3.2. Test diagnosis

The specification steps (b and c) define via failure mode assumptions [35] on the internal components and the system itself (h) (e.g., "are the clocks synchronized?" or "is there only one master in the system?") what is to be observed to assess the results of a single test run.

Basically, a fault tolerance mechanism receives its input from the other components of a fault-tolerant system. These inputs exhibit some properties, called error patterns, relative to the failure mode assumption of these components. Some of these failure mode assumptions are individual, e.g., a fail-silent component. Others are collective, e.g., there is only one faulty component at a time for a subset of the components. The fault tolerance mechanism is designed upon these failure mode assumptions and more precisely on the input error patterns resulting from these assumptions. A predicate H_{pre} is defined which is true when the failure mode assumption, as reflected on the inputs of the fault tolerance mechanism, are satisfied.

The considered fault tolerance mechanisms are such that the entire system in which they are included, delivers a service exhibiting some properties relative to its failure mode assumptions. A predicate H_{post} is defined which is true when these failure mode assumptions are satisfied in the delivered service. This includes also the reaction of the fault tolerance mechanisms to the fault being injected.

4. MEFISTO-L

The design of MEFISTO-L benefited from recent works on VHDL-based fault injection techniques and tools [5, 8, 23, 24, 36-39]. The main innovative features of the tool include:

- the embedded VHDL code analyzer, that facilitates the identification of the signals to be injected at different levels of the model hierarchy;
- the observation and injection mechanisms, their synchronization and their automatic placement in the target VHDL model to derive the mutated model (i.e., including saboteurs and probes).

4.1. Organization

MEFISTO-L is composed of three distinct functional blocks: the parsing block, the injection block and the results extraction block (Figure 4).

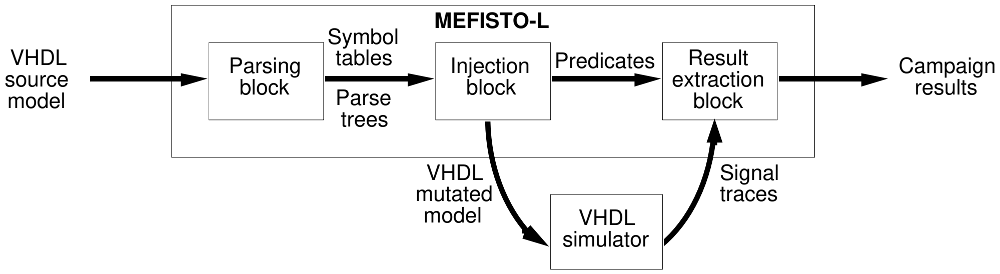

Figure 4 - Organization of MEFISTO-L

The parsing block is meant to extract the information 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. All probes and saboteurs necessary for the fault injection experiments to be carried out are incorporated into the model; accordingly only one compilation is necessary. The results extraction block analyzes the traces obtained from the simulation of the mutated model to produce the results of the campaign. In the sequel, we focus on the injection block; refer to [32] for more details on the other blocks.

4.2. The Injection Block

To set up the fault injection campaign, the injection block relies on the constructs generated by the parsing block. Figure 5 depicts the structure of the injection block.

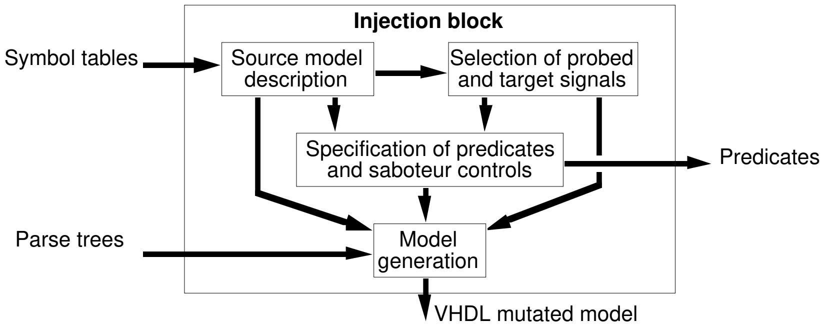

Figure 5 - Structure of the injection block

The main objectives of the injection block are: (i) to help the user in specifying the injection campaign and, (ii) to automatically generate the mutated model. The first objective encompasses the description of the source model, the selection and specification of the target and probed signals, the specification of the control of the injection process, and the definition of the error, detection and failure predicates. The second objective relies on the manipulation of the different parse-trees of the source model to achieve the mutation of the model. We briefly illustrate these successive steps in what follows.

Description of the source model. This description gives an overview of the source model to the user, so that he can efficiently determine the target signals. As shown on the left side window of Figure 6, each level of the model hierarchy can be browsed, and described by means of the following information.

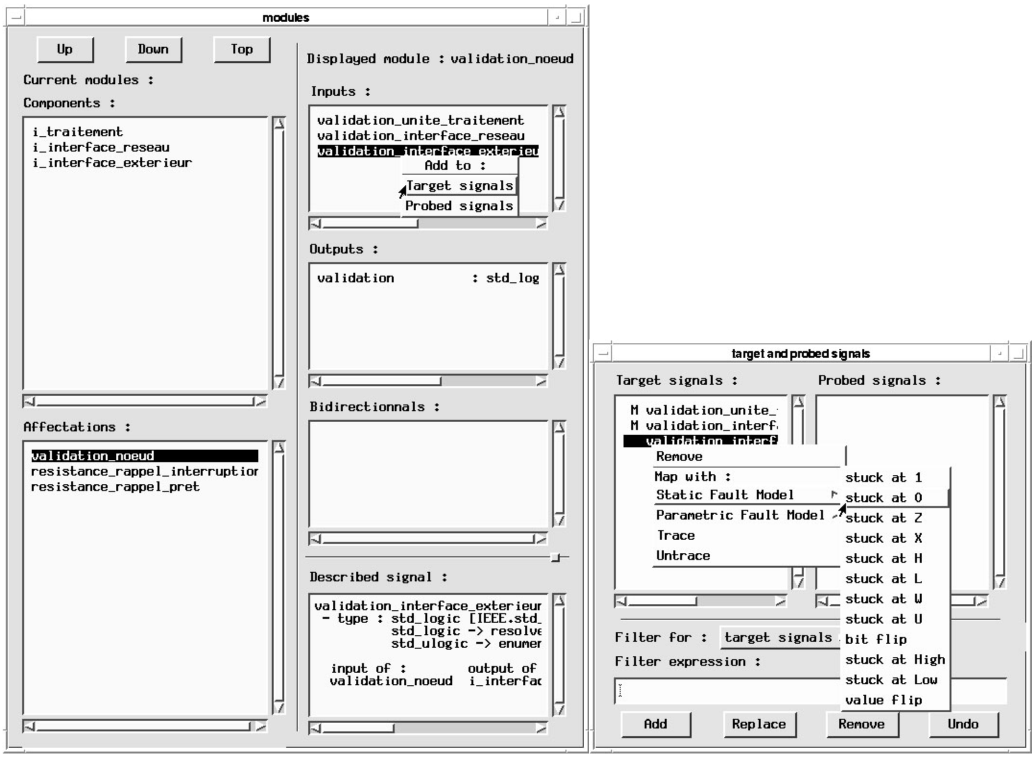

Figure 6 - Selecting a target signal and applying a fault model

The interface signals of the currently selected module are listed, including the type of the signal, and its communication mode (in, out or in/out). The description of each of these signals can be refined to provide more detailed information, such as the location of the type declaration, and the different types it is derived from. The internal signals of the currently displayed module are listed, along with their type, and may also be described as the interface signals.

Construction of the list of target signals. The description of the model hierarchy allows the user to select directly the target signals in the displayed lists. The target signals of interest to the user can be selected from each level of the model hierarchy. This simple form of selection provides precision and simplicity.

Figure 6 shows the interaction with MEFISTO-L to select a target signal and apply a fault model on it. On the model hierarchy browsing window (left), the signal `validation_interface_exterieur` (external interface validation) is chosen from the input signals of a concurrent signal assignment `validation_noeud` (node validation), and added to the list of target signals for injection. The right side window gives the list of injectors and probes. Here, a `stuck-at-0` fault-model is applied on the signal chosen in the left side window.

A set of filter mechanisms is available to help build the list according to some properties on the signals of the model. These filters can be combined with set manipulations functions (union, intersection and difference of signals lists).

Placement of the saboteurs. The fault injection method implemented in the tool relies on the insertion of saboteur components onto VHDL signals. Although the signal consists of a single VHDL entity, it may in practice contain several fault injection locations (especially in the case of signal connected to multiple components, as busses). A saboteur may be placed either on the output of a component driving the signal (thus disrupting all the components consuming the signal) or on the input of a component consuming the signal (thus disrupting only this component).

The saboteur placement can be seen as the last step of the target signal selection, by selecting a specific location on a single signal. The tool provides a simple way to precisely specify the placement of the saboteur: a specific function, implemented as a special filter `#apply`, has been added, and the semantics of the set manipulations has been modified to take the saboteurs placement into account. The `#apply` filter uses the previously built signal list and tries to apply each signal to the specified component: if the considered signal is linked to the component, the saboteur is specified to corrupt the related part of the signal, otherwise the saboteur is maintained at its "default" position. This default position consists in applying the saboteur between the group of modules using the signal as an output and the group of modules using the signal as an input.

Assigning fault models to saboteurs. Each saboteur has to be assigned a specific fault model. MEFISTO-L provides a library of saboteurs, including several fault models for main standard types (see Figure 6). Some fault models are specific to a type, and others are common to different types. The tool allows the user to map a fault model to a group of saboteurs, if the fault model is included in the intersection of the fault model lists of the saboteurs of the group.

Specifying probes for either observation or control. The specification of the fault injection campaign concerns the choice of the fault injection targets, but also the control of saboteur activations and observations points in the model to assess relevant predicates. Both objectives are achieved by the insertion of simple FI components acting as observation components that are called probes (cf. Section 2.2).

The list of the probed signals is built at each hierarchical level in the same way as the list of target signals. Each probed signal may be traced explicitly, or implicitly via a predicate expression.

The probe applied on a signal may be associated with specifications: these specifications consist in test expressions, that will be inserted in the code to communicate internal information to the injection control module of the mutated model. These tests allow dynamic control of the activations of the saboteurs.

A probe definition specifies a signal value or a range of signal values for which the probe is asserted as true, and false otherwise. Several probes may be defined for a single signal.

Specification of the control of the saboteurs. The control of the saboteur is specified at the highest level of the hierarchy. The specification consists in a logical expression on some selected parameters from the following list:

- the value of a probe, at every level of the model;
- the result of the comparison between the values of two probed signals;
- the value of static clock probe, defined by the user via their rising edge, their falling edge and their optional period.

These mechanisms allow the user to specify permanent or transient faults, with static or dynamic activation.

Definition of predicates. The definition of predicates allows for specifying conditions to assess the results of the fault injection campaign. These predicates are mainly used for observing the issue of the test according to the test diagnosis predicates *Hpre* and *Hpost* (see Section 3.2).

The predicates are defined as Boolean conditions on the faulted model status, or on the comparison of the status obtained during the simulations of the faulted model and of the non faulted model.

Besides the observation of *Hpre* and *Hpost* predicates for test diagnosis, several error, detection and failure predicates can be defined, enabling to precisely observe the behavior of the faulted simulation model, including the error propagation process.

5. Preliminary Results and On-going Research

To assess the usefulness of the testing strategy, in particular when using MEFISTO-L, a simple fault-tolerant system whose VHDL model has been developed [33]. The target system gathers several FTMs taken from actual fault-tolerant systems for control process in industrial or embedded real-time applications.

The nominal architecture features three channels interconnected by a network supporting a time-sliced protocol [40]. The system interfaces with the process by means of redundant sensors (one for each node) 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” [41]: in case of detection of an error, the channel should extract itself from the network and wait for repair. More precisely, each channel implements a dual-node architecture, adapted from [42]. Each node contains a processing unit and two interfaces (to the controlled process and to the other channels via the network). The 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] also considered in Section 2.3. It is animated 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 can be connected to the actuator.

The proposed testing strategy based on the use of (i) DRIFTs to determine the main interactions between the FTMs, and (ii) behavioral models for each elementary FTMs to derive the suited test patterns (fault and activity) has been applied to this target system. Furthermore, so far, the study of simple error detection mechanisms (e.g., a detector of loss (overwriting) of a queued message received via the network) 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 fault in the design of the target VHDL model [43].

Additional developments still need to be carried out on MEFISTO-L to fully support the proposed testing strategy. They concern in particular the addition of new fault injection capabilities. Several actions are involved, including:

- Integration of more fault models for standard signal types: the available fault models simply modify the value of a signal; it should be interesting to also propose time-based fault models such as delay.
- Enhancement of clock types for synchronizing injectors: in particular, random event generators are a need to support the statistical testing framework.
- Incorporation of an injector customization module: the aim is to give the user the ability to define his own injectors for some user-defined signal types associated to a specific system.
- Addition of capabilities for injecting faults in other VHDL objects than signals: the most promising target is constituted by the variables of sequential descriptions in VHDL.

Acknowledgment. I am pleased to acknowledge the contributions made by several people to the work reported in this paper, and especially Jérome Boué, Yves Crouzet, Jean-Etienne Doucet, Eric Jenn and Philippe Pétillon at LAAS, and Johan Karlsson, Joakim Ohlsson and Marcus Rimén at Chalmers University.

References

[1] J. Arlat, “Fault Injection for the Experimental Validation of Fault-Tolerant Systems”, in *Proc. Workshop Fault-Tolerant Systems,* Kyoto, Japan, 1992, pp. 33-40 (IEICE, Tokyo, Japan).

[2] J. A. Clark and D. K. Pradhan, “Fault Injection — A Method for Validating Computer-System Dependability”, *Computer*, vol. 28, pp. 47-56, June 1995.

[3] M.-C. Hsueh, T. K. Tsai and R. K. Iyer, “Fault Injection Techniques and Tools”, *Computer*, vol. 30, pp. 75-82, April 1997.

[4] G. S. Choi and R. K. Iyer, “FOCUS: An Experimental Environment for Fault Sensitivity Analysis”, *IEEE Trans. on Computers*, vol. 41, pp. 1515-1526, December 1992.

[5] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson and J. Karlsson, “Fault Injection into VHDL Models: The MEFISTO Tool”, in *Proc. 24th Int. Symp. on Fault-Tolerant Computing (FTCS-24),* Austin, TX, USA, 1994, pp. 66-75 (IEEE Computer Society Press).

[6] C. R. Yount and D. P. Siewiorek, “A Methodology for the Rapid Injection of Transient Hardware Errors”, *IEEE Trans. on Computers*, vol. 45, pp. 881-891, August 1996.

[7] K. K. Goswami, R. K. Iyer and L. Young, “DEPEND: A Simulation-Based Environment for System Level Dependability Analysis”, *IEEE Trans. on Computers*, vol. 46, pp. 60-74, January 1997.

[8] V. Sieh, O. Tschäche and F. Balbach, “Comparing Different Models Using VERIFY”, in *Proc. 6th IFIP Int. Working Conference on Dependable Computing for Critical Applications (DCCA-6),* Grainau, Germany, 1997, pp. 59-76.

[9] J. H. Barton, E. W. Czeck, Z. Z. Segall and D. P. Siewiorek, “Fault Injection Experiments Using FIAT”, *IEEE Trans. on Computers*, vol. 39, pp. 575-582, April 1990.

[10] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, “FERRARI: A Flexible Software-Based Fault and Error Injection System”, *IEEE Trans. on Computers*, vol. 44, pp. 248-260, February 1995.

[11] J. Carreira, H. Madeira and J. G. Silva, “Xception: Software Fault Injection and Monitoring in Processor Functional Units”, in *Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5),* Urbana-Champaign, IL, USA, 1995, pp. 135-149.

[12] E. Fuchs, “An Evaluation of the Error Detection Mechanisms in MARS using Software-Implemented Fault Injection”, in *Proc. 2nd European Dependable Computing Conf. (EDCC-2),* Taormina, Italy, 1996, pp. 73-90 (Springer-Verlag).

[13] D. T. Stott, M.-C. Hsueh, G. L. Ries and R. K. Iyer, “Dependability Analysis of a Commercial High-Speed Network”, in *Proc. 27th Int. Conf. on Fault-Tolerant Computing (FTCS-27),* Seattle, WA, USA, 1997, pp. 248-257 (IEEE Computer Society Press).

[14] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber and J. Reisinger, “Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture”, in *Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5),* Urbana-Champaign, IL, USA, 1995, pp. 150-161.

[15] J.-C. Laprie, “Dependable Computing: Concepts, Limits, Challenges”, in *Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25). Special Issue,* Pasadena, CA, USA, 1995, pp. 42-54 (IEEE Computer Society Press).

[16] J. Arlat, Y. Crouzet and J.-C. Laprie, “Fault Injection for the Experimental Validation of Fault Tolerance”, in *Proc. Esprit Conference (Esprit'91), (CEC-DGXIII),* Brussels, Belgium, 1991, pp. 791-805 (Office of Official Publications of the European Communities, Luxembourg).

[17] K. Echtle and Y. Chen, “Evaluation of Deterministic Fault Injection for Fault-Tolerant Protocol Testing”, in *Proc. 21st Int. Symp. on Fault-Tolerant Computing (FTCS-21),* Montréal, Québec, Canada, 1991, pp. 418-425 (IEEE Computer Society Press).

[18] D. Avresky, J. Arlat, J.-C. Laprie and Y. Crouzet, “Fault Injection for the Formal Testing of Fault Tolerance”, *IEEE Trans. on Reliability*, vol. 45, pp. 443-455, 1996.

[19] 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”, in *Proc. 7th Int. Symp. on Software Reliability Engineering (ISSRE'96),* White Plains, NY, USA, 1996, pp. 175-184 (IEEE Computer Society Press).

[20] T. Tsai, S. J. Upadhayaya, H. Zhao, M.-C. Hsueh and R. K. Iyer, “Path-Based Fault Injection”, in *Proc. 3rd ISSAT Int. Conf. on Reliability and Quality in Design,* Anaheim, CA, USA, 1997, pp. 121-125.

[21] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems”, *IEEE Trans. on Computers*, vol. 42, pp. 913-923, August 1993.

[22] D. Powell, M. Cukier and J. Arlat, “On Stratified Sampling for High Coverage Estimations”, in *Proc. 2nd European Dependable Computing Conf. (EDCC-2),* Taormina, Italy, 1996, pp. 37-54 (Springer-Verlag).

[23] P. Prinetto, A. Benso, F. Corno, M. Rebaudengo, M. Sonza Reorda, A. Amendola and L. Impagliazzo, "Fault Behaviour Observation of a Microprocessor System through a VHDL Simulation-Based Fault Infection Experiment", in *Proc. EURO-DAC'96,* Geneva, Switzerland, 1996.

[24] 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", in *Proc. Int. Computer Performance and Dependability Symp. (IPDS'95),* Erlangen, Germany, 1995, pp. 194-203 (IEEE Computer Society Press).

[25] H.-P. Juan, N. D. Holmes, S. Bakshi and D. P. Gajski, "Top-Down Modeling of RISC Processors in VHDL", in *Proc. EURO-VHDL'93,* Paris, France, 1993, pp. 454-459.

[26] A. A. Jerraya, I. Park and K. O'Brien, "AMICAL: An Interactive High Level Synthesis Environment", in *Proc. EDAC/EUROASIC'93,* Paris, France, 1993, pp. 58-62 (IEEE Computer Society Press).

[27] J. H. Aylor, R. Waxman, B. W. Johnson and R. D. Williams, "The Integration of Performance and Functional Modeling in VHDL", in *Performance and Fault Modelling with VHDL,* (J. M. Schoen, Ed.) pp. 22-145, Englewood Cliffs, NJ, USA: Prentice-Hall, 1992.

[28] A. Miczo, "VHDL as a Modeling-for-Testability Tool", in *Proc. COMPCON'90,* 1990, pp. 403-409 (IEEE Computer Society Press).

[29] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. no. 1076-1993, 1993.

[30] M. Rimén, J. Ohlsson, J. Karlsson, E. Jenn and J. Arlat, "Design Guidelines of a VHDL-based Simulation Tool for the Validation of Fault Tolerance", in *Proc. 1st Open Workshop ESPRIT Basic Research Project 6362: PDCS-2,* Toulouse, France, 1993, pp. 461-483.

[31] P. J. Ashenden, "The VHDL Cookbook", Tech. Rep. Univ. Adelaide, South Australia, 1990.

[32] J. Boué, P. Pétillon, Y. Crouzet and J. Arlat, "Early Experimental Verification of Fault Tolerance: The VHDL-based Fault Injection Tool MEFISTO-L", in *ESPRIT Project 20072: Design for Validation - 2nd Year Report,* 1997 (to appear).

[33] J. Boué, J. Arlat, Y. Crouzet and P. Pétillon, "Verification of Fault Tolerance by Means of Fault Injection into VHDL Simulation Models", *ESPRIT Project 20072: Design for Validation - 1st Year Report,* pp. 347-369, LAAS-CNRS, 1996.

[34] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "Software Statistical Testing", in *Predictably Dependable Computing Systems,* (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), vol. 1, pp. 253-272, Berlin, Germany: Springer, 1995.

[35] D. Powell, "Failure Mode Assumptions and Assumption Coverage", in *Proc. 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22),* Boston, MA, USA, 1992, pp. 386-395 (IEEE Computer Society Press).

[36] J. R. Armstrong, F.-S. Lam and P. C. Ward, "Test Generation and Fault Simulation for Behavioral Models", in *Performance and Fault Modelling with VHDL,* (J. M. Schoen, Ed.) pp. 240-303, Englewood Cliffs, NJ, USA: Prentice-Hall, 1992.

[37] Z. Navabi, N. Cooray and R. Liyanage, "Modeling for Fault Insertation and Parallel Fault Simulation", in *Proc. VHDL International Users Forum Spring Conference,* Santa Clara, CA, USA, 1993, pp. 75-89.

[38] B. Cohen, *VHDL Coding Styles and Methodologies*, Norwell, MA, USA: Kluwer Academic Publishers, 1995.

[39] T. A. DeLong, D. T. Smith, B. W. Johnson and J. P. Hanna, "Simulator Independent Fault Simulation Using WAVES", in *Proc. VHDL International Users Forum Fall 1996 Conference,* Durham, NC, USA, 1996, pp. 129-138.

[40] H. Kopetz, "The Time-Triggered Approach to Real-Time System Design", in *Predictably Dependable Computing Systems,* (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), pp. 53-66, Berlin, Germany: Springer, 1995.

[41] D. Powell, G. Bonn, D. Seaton, P. Veríssimo and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", in *Proc. 18th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18),* Tokyo, Japan, 1988, pp. 246-251 (IEEE Computer Society Press).

[42] D. Brière and P. Traverse, "AIRBUS A320/A330/A340 Electrical Flight Controls - A Family of Fault-Tolerant Systems", in *Proc. 23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23),* Toulouse, France, 1993, pp. 616-623 (IEEE Computer Society Press).

[43] J. Boué, Fault Tolerance Testing by means of Fault Injection in VHDL Simulation Models, Doctorate Dissertation, National Polytechnic Institute, Toulouse, France, 1997 (in French).