Fault Injection into VHDL Models: The MEFISTO Tool[1]

Eric Jenn*, Jean Arlat*, Marcus Rimén, Joakim Ohlsson** and Johan Karlsson****

*** LAAS-CNRS, 7, Avenue du Colonel Roche, 31400 Toulouse, France**
**** Laboratory for Dependable Computing, Chalmers University of Technology, S-412 96 Gothenburg, Sweden**

Abstract

This paper 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. Suitable techniques for injecting faults into VHDL models are identified and depicted. Then, the main features of the MEFISTO environment aimed at supporting these techniques are described. Finally, some preliminary results obtained with MEFISTO are presented and analyzed.

Index Terms: *Experimental Validation, Fault Injection, Simulation, VHDL, Fault/Error Models.*

1 Introduction

In the last decade, fault injection has emerged as an invaluable means to support the dependability validation of fault-tolerant systems (e.g., see [1-3]).

Two main trends characterize 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 [3-6] 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 [7-10].

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 (e.g., pin-level [11] or heavy-ion [12]). 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.

The paper presents the main objectives and the preliminary results of a research aimed at (i) providing an integrated environment — called MEFISTO (Multi-level Error/Fault Injection Simulation TOol) — for applying fault injection into simulation models encompassing various levels of abstraction and thus (ii) helping to identify and validate an abstraction hierarchy of fault/error models.

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. With this respect, VHDL has been recognized 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 [13],
- 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, and (ii) for supporting hybrid (i.e., mixed abstraction levels) simulation models [14],
- capability to support test activities [15].

MEFISTO can be used: (i) to estimate the coverage of fault tolerance mechanisms, (ii) to investigate different mechanisms for mapping results from one level of abstraction to another and (iii) to validate fault and error models applied during fault injection experiments carried out on the implementation of a fault-tolerant system (e.g., SWIFI or pin-level fault injection).

The remainder of this paper is made up of six sections. Section 2 presents a critical analysis of four techniques for injecting faults into VHDL models. Section 3 describes the architecture and the main features of the MEFISTO system that supports these techniques. Section 4 details the user interactions necessary to set up a fault injection campaign. Section 5 presents a case study where MEFISTO is used to

[1] This work was supported in part by the CEC ESPRIT Basic Research Project nº6362 PDCS2 (Predictably Dependable Computing Systems) and by the *Midi-Pyrénées* Regional Authority under contract RECH/90078306.

inject faults in two VHDL models (structural and behavioral) of a simplified 32-bit processor. Abstract error models are extracted from the resulting data and compared. Section 6 concludes the study with a summary and some remarks.

2 Fault injection into VHDL models

Two categories of fault injection techniques are identified: the first one covers two techniques that require *modification of the VHDL model* and the second one covers two other techniques that instead use the *built-in commands* of the simulator. Before describing these two categories, we provide a summary of the characteristics of the VHDL language used in the subsequent sections [16].

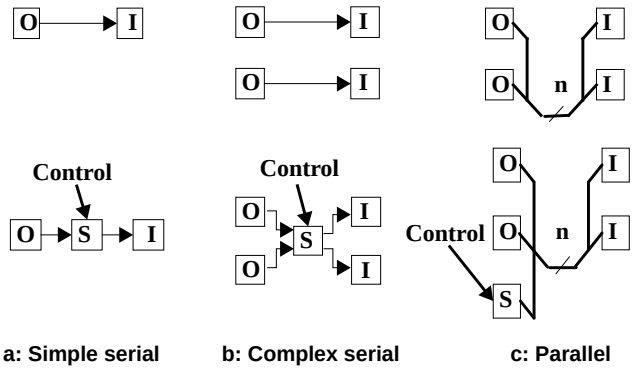
2.1 Characteristics of the VHDL language

A system modeled in VHDL is made up of *components* linked by *signals*. VHDL allows 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.

A component can have many implementations. In VHDL, each implementation is expressed by a separate component description. To allow simulation of the system, a *configuration* mechanism assigns a single component description to each component. A component description is represented by both an entity declaration and an architecture. The entity declaration specifies the interface, i.e., the set of inputs and outputs, while the architecture describes the internal organization of the component. A component description can be *structural* or *behavioral* (or a mixture of both). A structural description is a composition hierarchy consisting of subcomponents and signals, while a behavioral description is an algorithmic model of the component's function expressed by means of *processes*. A VHDL *process* is a set of sequential statements. The process can communicate with other processes by reading and assigning values to *signals*. *Variables* are used inside processes and are not accessible from the outside. Processes are executed repeatedly in a loop fashion, and stopped only at synchronization statements, called *wait statements*.

2.2 Modification of the VHDL model

In this category, two techniques can be distinguished. The first one is based on the addition of dedicated fault injection components, called saboteurs, to the VHDL model. The second one is based on the mutation of existing component descriptions in the VHDL model, which generates modified component descriptions called mutants. In this paper, any change made to an existing component description is regarded as a mutation.



(original structures at the top modified at the bottom)

Figure 1: Insertion of saboteurs

A *saboteur* is a VHDL component that alters the value or timing characteristics of one or several signals when activated. It is usually inactive during normal system operation and activated only to inject a fault. A *serial saboteur* breaks up the signal path between a driver (output) and its corresponding receiver (input) (Figure 1-a) or a set of drivers and its corresponding set of receivers (Figure 1-b). A parallel saboteur is simply added as an additional driver for a resolved signal, as shown by Figure 1-c. Another way to achieve the same effect is to assign a modified resolution function to signals that are potential fault targets.

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 imitates the component's behavior in presence of faults. It is easy to implement this replacement technique in VHDL using the configuration mechanism.

Mutation may be accomplished in several ways by:

- adding saboteur(s) to structural or behavioral component descriptions,
- 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 can be used for controlling, i.e., activating and deactivating, saboteurs and mutants. In this way, the injection of faults can be controlled by the built-in commands of the simulator also when mutants and saboteurs are used. This can be achieved by means of global signals or shared global variables. The latter is a new capability introduced in the latest version of VHDL, called VHDL'93 (IEEE Std. 1076-1993).

2.3 Use of built-in commands of the 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 functionnalities offered by the command languages of the simulators.

Two techniques based on the use of simulator commands can be identified: *signal* and *variable* manipulation. For each technique, the required sequence of simulator pseudo-commands is described in Figure 2 and is explained below.

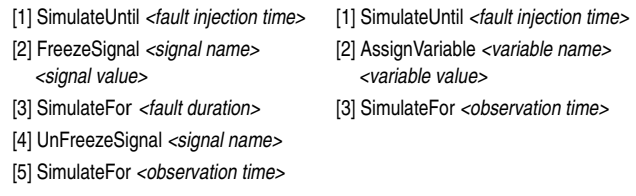


Figure 2: Fault injection using simulator commands

2.3.1 Signal manipulation: In this technique, faults are injected by altering the value of signals in the VHDL model. This is done by simulating the system until the time of injection is reached and all processes have stopped on a `WAIT` statement (Figure 2-a, step 1), then the signal is disconnected from its driver(s) and forced to a new value (step 2); the system is simulated for the duration of the fault (step 3); finally, the signal's driver(s) is (are) reconnected when the fault injection is completed (step 4) and the simulation is continued until the end of the observation time (step 5). For a permanent fault, steps 3 and 4 are skipped. Intermittent faults can be injected using a more complex command sequence.

2.3.2 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.

A variable is termed *atemporal* when it holds valid information over a number of sequential statements in the process, but never over any `WAIT` statements. Thus, the variable will not be susceptible to changes made to its contents when the process has stopped. A variable is termed *temporal* if it always holds valid information over `WAIT` statements. As a result, the variable will be susceptible to changes made to its contents when the process has stopped. In other words, a variable is temporal or atemporal depending whether it is used to transfer information through simulation time space or code space, respectively. Furthermore, a variable is termed *mixed-mode* if it is sometimes temporal and sometimes atemporal.

In the case of temporal variables, faults are injected by the same means as signals, i.e., when all processes are stopped on a `WAIT` statement (see Figure 2.b). By definition, this technique cannot be used for atemporal variables since they are only used between two `WAIT` statements, therefore some extra fine-grained synchronization is required. Figure 3 shows the simulator controls required to manipulate an atemporal variable. The main idea is to (i) simulate the system until one of the statements where the target variable is assigned a value is reached (steps 1-3) and (ii) then assign a faulty value to it (steps 4-7).

Figure 3: Variable manipulation of atemporal variables

Step 1 in Figure 3 is the same as step 1 in Figure 2. In step 2, breakpoints are set on all VHDL source code lines that follow an assignment to the target variable. Then the simulation is started in step 3 and continues until a breakpoint is reached. However, as the execution can follow paths in the VHDL code where no assignments are made to the target variable from one `WAIT` statement to the next one, it cannot be ensured that a source line breakpoint will be reached. Therefore it is necessary to specify a maximum time for the simulation, which is defined by the <*time window*> parameter in step 3. If a breakpoint is reached (step 4), a fault is injected (step 5), then all source line breakpoints are removed (step 6) and the simulation is continued until the end of the observation time (step 7). If a breakpoint is not reached within the time window (step 4) then steps 5-7 are skipped, i.e., no fault is injected.

The injection of faults into mixed-mode variables is achieved by a combination of the above methods; the sequence is the same as the one given in Figure 3, with an "AssignVariable <*variable name*> <*variable value*>" step inserted between the first and the second steps.

2.4 Comparison of the fault injection techniques

The fault injection techniques considered are compared in terms of *fault modeling capacity, effort required for setting up an experiment* and *simulation time overhead*.

Considering the *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 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.

Considering the *effort 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 mutants and saboteurs, as they require (i) creation/generation of saboteurs/mutants, (ii) inclusion of saboteurs/mutants in the model and (iii) recompilation of the VHDL model. The creation of saboteurs and the automatic generation of mutants are relatively easy tasks, provided that simple fault models are considered. It is also worth noting that a saboteur is a reusable component, while a mutant has to be specifically generated for each target component; on the other hand, the inclusion of saboteurs requires the modification of the component description while mutants are easily included in the model by means of the VHDL configuration mechanism.

Considering the *simulation time overhead* induced by the injection mechanisms, 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 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.

3 Overview of MEFISTO

Figure 4 gives an overview of MEFISTO and of the main user interactions for defining and executing a *fault injection campaign* (i.e., a series of *fault injection experiments)*. The fault injection campaign consists of three phases: a *Setup Phase*, a *Simulation Phase*, and a *Data Processing Phase*.

3.1 The Setup Phase

The two main objectives of the setup phase are to generate:

- an *Executable Model* of the system including mutants, if any[2] ;
- an *Experiment Control List* containing the commands necessary to control the simulator for each experiment in the campaign.

For a given VHDL description of the target system, the user specifies during the Setup Phase (i) the fault set to be used, (ii) the readouts to be saved (e.g., signal traces) in the campaign, (iii) how to select faults from the fault set for each experiment, (iv) when to inject the selected fault(s), and (v) when to terminate each experiment.

In the first part of the Setup Phase, the tool analyzes the VHDL target model and provides a list of fault targets to the user. The user specifies the fault set to be used in the campaign by selecting targets from the list. Each target belongs to a target group which corresponds to a certain class of VHDL objects, e.g., signal, variable or component. Each target group has at least one attribute such as a type (`BOOLEAN`, `INTEGER`, etc.) for a signal or an implementation (VHDL entity declarations and architectures) for a component. For each fault target, a user-definable fault type can be selected. This fault type is called an *Abstract Fault Model*, or AFM. The act of selecting a fault target and a fault type (AFM) specifies a fault.

AFMs can be applied to any of the target groups described above. It is also possible to restrict the application of an AFM to targets with a specific attribute, e.g., signal of type `BIT`. Thus, for each target there exists a set of selectable AFMs from which the user can chose.

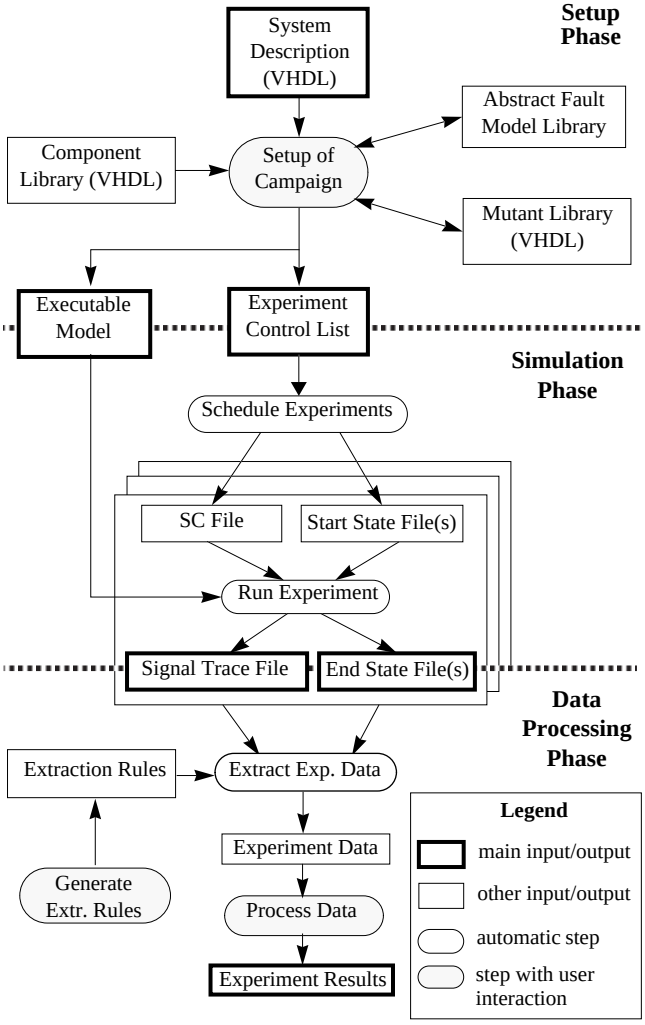


Figure 4: Overview of the fault injection tool

[2] Saboteurs are not directly supported by the system, but can easily be handled. The use of a saboteur is regarded as the generation of a mutant. Thus, saboteurs can be handled the same way as mutants.

When mutants are used, after the fault set is specified by the user, a final VHDL configuration for the target system is created and then the Executable Model can be generated. Once the final configuration is decided, all signals, variables etc., are identified and presented to the user. From this list the user selects the readouts to be saved for each experiment. Finally, the user must specify for each experiment how to select faults from the fault set, when to inject them and when to terminate the experiment. This information is needed by the tool to generate the Experiment Control List.

The main user interactions in the Setup Phase are further described in Section 4. A more comprehensive description can be found in [17].

3.2 The Simulation Phase

In the Simulation Phase, the simulation model created in the Setup Phase together with the Experiment Control List are used to schedule the execution of the injection experiments.

The simulations are executed on a set of simulators running on a network of computers. An experiment scheduler reads the Experiment Control List and schedules simulations on the available simulators. For each simulation, the scheduler reads an entry with simulation control commands from the Experiment Control List and creates a simulation control file (*SC File*) for controlling the simulator, and a start state file. The *Start State File* specifies from which simulation state a simulation should start. The final simulation state (*End State File*) of each simulation can be saved; this makes it possible to use the final simulation state of a previous simulation as a start state for a new simulation. Moreover, the readouts (*Signal Trace File*) from each simulation are saved for further processing in the Data Processing Phase.

3.3 The Data Processing Phase

The Data Processing Phase involves two steps: (i) extraction of *Experiment Data* from the raw data produced by the simulations, and (ii) processing of the Experiment Data into *Experiment Results*.

The purpose of the extraction process, which is guided by Extraction Rules generated by the user, is to extract and convert to a convenient format the data needed to generate the experimental results (e.g., error detection coverage and latency measures).

The actual processing of the Experiment Data into Experiment Results is application-dependent and must be tailored for each fault injection campaign. For instance, in one campaign the user may want to compute coverage figures for a fault tolerance mechanism, while in another campaign he may want to obtain enough data to study the error propagation in the system.

4 Main user interactions in the Setup Phase

Section 4.1 and Section 4.2 describe the generation of AFMs and mutants, respectively. Then, Section 4.3 describes the selection of the fault set. Finally, Section 4.4 presents the selection of experiment parameters.

4.1 Generation of AFMs

MEFISTO includes an AFM database containing predefined AFMs. The user can expand the database by generating and adding new AFMs to it. By default, certain target groups and attributes are recognized by the tool (such as signals of the type `BIT`), and associated simple AFMs (e.g., stuck signal of type `BIT` to 0) for injecting faults are stored in the database.

In addition to the predefined AFMs, the user can add both simple and complex AFMs, such as those supporting the saboteur and mutant-based techniques, to the database. Those techniques require the generation of a mutant. The mutant is stored in a database (the mutant library) and a complex AFM is associated with the mutant, containing a reference to the mutant and also a “description” of how to activate it.

4.2 Generation of mutants

A *mutation* of a component description can be carried out in several ways by (i) manually mutating it, (ii) inserting saboteurs into it and (iii) manually selecting mutation rules for the automatic mutation. A mutation rule states where the mutation must occur, and what modification to make. The only rules available in MEFISTO are those that map one VHDL grammar rule to another one. A mutation rule can make calls to other mutation rules, in a recursive manner. Indeed, the mutation of an entity containing instances (i.e., components) of other entities may be obtained by the mutation of any of these components.

Examples of fault models possible to describe as mutation rules are found in [18] that divides behavioral-level fault models into eight fault classes: Stuck-Then, Stuck-Else, Assignment Control, Dead Process, Dead Clause, Micro-operation, Local Stuck-data and Global Stuck-data.

4.3 Selection of the fault set

The first step in the process of selecting a fault set consists in the analysis of the VHDL description to get information on potential targets for fault injection: *target group* (e.g., signal, variable, component etc.), *attribute* (e.g., boolean, integer, etc.), etc. As mentioned earlier, because of its group belonging and attribute, each target implicitly defines one or more selectable AFMs.

After analyzing the VHDL description, the faults to be injected are determined interactively and added to a fault set. This is done by selecting targets and AFMs.

The fault injection tool provides three main categories of target selection mechanisms:

- explicit target selection,
- selection of targets that satisfy a given property,
- random target selection.

The properties for the second category concern (i) the entity-instances relationship in the model (e.g., select mutated architectures for components that are instances of a specific entity), (ii) the composition structure of the model (e.g., all signals in a specific architecture, or all variables in a specific process), (iii) the VHDL semantics (e.g., all variables, all signals, all processes, all variables or all signals that carry values of a specific type) and (iv) a given topological property (e.g., all signals connected to a given component).

The selection criteria may be composed; for instance, it is possible to select "all signals that are of a given type and are connected to a given component".

4.4 Selection of experiment parameters

The experiment parameters include preconditions, fault activation conditions and termination conditions.

The *preconditions* specify the starting state of each experiment. It can be (i) the zero state, i.e., the state corresponding to a reset of the target system, or (ii) the final state of a previous experiment. The preconditions also define the functional activation of the simulation model for each experiment. In particular, the functional activation of a microprocessor is also called the workload.

The *activation* of the injected faults (i.e., AFMs associated with targets) can be triggered according to any mix of the following conditions provided by the simulation environment:

- a given period of simulation time has elapsed,
- a signal has changed (to a given value),
- a given line of code has been executed, or
- one of the above conditions has occurred for the *nth* time.

All values in these definitions may be chosen at random.

The *termination conditions* state when an individual experiment should complete. The completion of the experiment is characterized by a predicate *P* on the system's state. Depending on its complexity, the assessment of the *P* predicate can be performed on-line or off-line with respect to a running simulation within an experiment.

When computed on-line, the *P* predicate is expressed by means of simulation tool expressions or VHDL code. The simulation environment provides the same means to terminate a simulation as for detecting the time for activating a fault as described above. Expressing the predicate in VHDL is a matter of embedding specific expressions, processes or even components in the original model. Their activation may create events that satisfy one of the conditions that can be monitored by the simulation

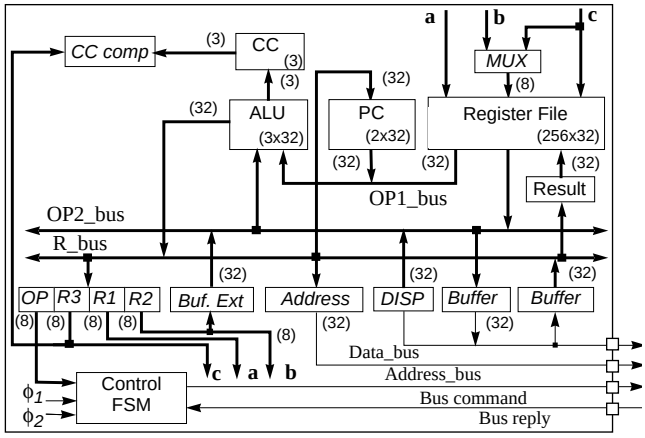


Figure 5: Structural architecture of the DP32 processor

environment to stop the simulation. When computed off-line, the same means as those cited above are used to assess a first predicate *P1*. However, once *P1* has been satisfied, the current simulation state is saved, the simulation is stopped and a second, more thorough, analysis is performed to test a second predicate *P2*. If *P2* is not satisfied, the simulation continues from the saved state.

5 A case study: the DP32 processor

This section presents some preliminary results of fault injection experiments carried out with MEFISTO on two models of a simple 32-bit processor. The main goals of the experiment consist in the analysis of the impact of the choice of the injection method and the model description level on the error outcome. More specifically, the differences observed when using (i) signal or variable manipulation methods[3] , and (ii) a structural model or a behavioral model of the target system are studied. The next paragraphs present the design and the realization of the campaign. Finally, the experimental results are given and analyzed.

5.1 Design of the campaign

The target system is a model of a very simple 32 bits processor, the DP32 described in [19], for which two architectures, one behavioral and one structural, are available. The structural model, on the register-transfer level, is depicted in Figure 5; it is mainly composed of a finite state control machine (Control FSM), an ALU, a program counter (PC), a register file and several buffers and latches. The behavioral model, on the procedural description level, mainly consists of a VHDL process containing a large "CASE" statement that initiates the appropriate bus cycles with respect to the operation code of the fetched instruction.

All external operations are synchronized on two non-

[3]Saboteurs and mutants are not used in the study.

overlapping clocks Φ_1 and Φ_2 (a cycle is 40 *ns*); Φ_1 is the main clock which synchronize all bus accesses.

Figure 6 rates the "simulation efficiency", the "complexity" and the "average simulation duration" for the two models. This table shows that for the processor used in the experiments, less events are needed to complete the execution of the same workload on the behavioral model than on the structural model, indicating that the structural model is more complex than the behavioral model. Furthermore, the simulator can simulate the events generated by the behavioral model faster, i.e., more efficiently, than those generated by the structural one. This clearly illustrates the trade-off between speed and accuracy.

Model	# Simulated events per sec	# Generated events per nsec	Average exec. time (3600 clock cycles)
Structural	4636	2.06	80 s
Behavioral	7927	1.07	25 s

Figure 6: Efficiency and complexity of the models

The fault models used in the experiments share the following characteristics:

- the time of injection — expressed in cycles — is uniformly distributed in the range [0, the execution time of the workload] and all injections are synchronized with the raising edge of Φ_1,
- the fault values are uniformly distributed in the range of permitted values for the target (e.g., [0,1] for targets of type BIT, [0,255] for targets of type BYTE, etc.).

At the structural level, single random temporary stuck-at faults have been injected systematically on all atomic signals (compound signals, such as buses, were splitted into their basic bit components), with a fault duration of one Φ_1 clock cycle. At the behavioral level, single random bit-flips were injected into variables.

Two fault injection campaigns with two different workloads — a Bubblesort (22 bytes) and a Heapsort (63 bytes) sorting programs applied to a set of 16 values — were run on each model.

5.2 Realization of the Campaign

In the following paragraphs, the implementation of the experiments is described in terms of the two main phases that have been identified in Section 3, i.e., setup and data processing. The completion of the simulation phase took about one week on two Sun IPC workstations.

5.2.1 The Setup Phase: In this case, it was not necessary to generate new AFMs as predefined AFMs corresponding to signal and variables manipulation techniques could be used.

The analysis of both VHDL models resulted in a set of target signals and variables. The selection mechanism described in Section 4.3 was used to reduce the initial set of (signal) targets resulting from the analysis of the structural model to a set of signals strictly internal to the processor[4]. A manual selection was used to exclude variables that were not used during the execution of the activations from the (variable) targets of the behavioral model. All output ports of the DP32 processor (*i.e., address bus*, *data bus*, *read*, *write*, *fetch*) were chosen as readouts for the fault injection experiments.

5.2.2 The Data Processing Phase: The final results were extracted from the experiment data by means of a two-step procedure consisting of (i) the determination of the time and location of the error manifestation on the output ports of the processor and (ii) the classification of these observations with respect to a set of error classes integrating information on latency, location and instruction cycle.

Each injected fault belongs to one of the fault classes listed in Figure 7. Each fault class corresponds to the function of the target signal/variable.

Structural fault class	Target	Behavioral fault class	Target
Buses	Internal buses	PC	Program counter
Xfer	Buffer control lines	CR	Control register (flags)
Latch	Latch control lines	IR	Instruction register
Select	Mux control lines	AR	Address register
Func	ALU control lines	DR	Data register
Misc	OTHER	UR	User register

Figure 7: Fault classes

The first step of the data processing was achieved by comparing the readouts of each fault injection simulation with the readouts of a reference simulation. In the second step, error classification predicates were used to classify the readouts of each experiment. The errors classes derive from an *a priori* knowledge of the processor structure and behavior. Figure 8 defines the predicates that characterize the considered error classes.

Error class	Predicates
Direct Execution	PDE = an error on any output port during the execution of instruction(i)
Direct Flow	PDF = an address bus error in the fetch phase of instruction(i+1)
Indirect Flow	PIF = an address bus error in the fetch phase of instruction(≥i+2)
Indirect Data	PID = an address bus error in the read or write phase of instruction(≥i+1), or a data bus error in the write phase of instruction(≥i+1)

Notes:
*(1)"indirect" refers to the fact that the error **propagated** from one instruction to the other **through a storage device** such as a user register or a condition code bit, and **stayed latent** until the storage element was used;*
(2) instruction (i) refers to the instruction executed when the fault is injected;
(3) All predicate values equal 1 when they are asserted and 0 otherwise.

Figure 8: Error classes

[4]The input and output ports have been excluded from this selection.

Class	Total		Buses		Misc		Xfer		Latch		Select		Func	
	%	#	%	#	%	#	%	#	%	#	%	#	%	#
Injected	100	3433	47.5	1632	41.6	1428	5.0	172	4.5	155	0.7	23	0.7	23
Effective	16.6	568	22.8	372	8.2	117	23.3	40	18.7	29	8.7	2	34.8	8
Direct Flow	**28.7**	163	31.5	117	15.4	18	37.5	15	31.0	9	0	0	50.0	4
Direct Execution	**49.5**	281	53.2	198	35	41	45.0	18	65.5	19	100	2	37.5	3
Indirect Flow	9.1	**52**	6.2	**23**	21.4	**25**	7.5	3	3.5	1	0	0	0.0	0
Indirect Data	12.7	**72**	9.1	**34**	28.2	**33**	10.0	4	0.0	0	0	0	12.5	1

a: Structural model for Heapsort

Class	Total		Buses		Misc		Xfer		Latch		Select		Func	
	%	#	%	#	%	#	%	#	%	#	%	#	%	#
Injected	100	3820	50.3	1920	40.8	1560	4.2	160	3.7	140	0.5	20	0.5	20
Effective	14.4	550	20.1	386	6.3	98	20.6	33	19.3	27	10.0	2	20.0	4
Direct Flow	**29.8**	164	32.9	127	22.5	22	15.1	5	37.0	10	0	0	0	0
Direct Execution	**51.5**	283	54.4	210	33.7	33	54.5	18	59.3	16	100	2	100	4
Indirect Flow	8.0	**44**	7.5	**29**	12.2	**12**	6.1	2	3.7	1	0	0	0	0
Indirect Data	10.7	**59**	5.2	**20**	31.6	**31**	24.2	8	0.0	0	0	0	0	0

b: Structural model for Bubblesort

Class	Total		PC		CR		IR		AR		DR		UR	
	%	#	%	#	%	#	%	#	%	#	%	#	%	#
Injected	100	3664	8.5	313	0.8	28	11.0	403	18.2	665	0.3	11	61.2	2244
Effective	21.9	803	45.7	143	3.6	1	7.2	29	2.9	19	0	0	27.2	611
Direct Flow	**11.7**	94	56.6	81	100	1	24.1	7	0	0			0.8	5
Direct Execution	**18.9**	152	43.4	62	0	0	51.7	15	100	19			9.2	56
Indirect Flow	33.2	**267**	0.0	0	0	0	3.4	1	0	0			43.5	**266**
Indirect Data	36.1	**290**	0.0	0	0	0	20.7	6	0	0			46.9	**284**

c: Behavioral model for Heapsort

Class	Total		PC		CR		IR		AR		DR		UR	
	%	#	%	#	%	#	%	#	%	#	%	#	%	#
Injected	100	3626	11.6	421	0.8	28	12.8	466	21.0	763	0.6	20	53.2	1928
Effective	20.6	746	47.7	201	0.0	0	5.2	24	3.9	30	10	2	25.4	489
Direct Flow	**19.2**	143	65.7	132			12.5	3	0	0	0	0	1.6	8
Direct Execution	**24.3**	181	34.3	69			50.0	12	100	30	0	0	14.3	70
Indirect Flow	33.9	**253**	0.0	0			4.2	1	0	0	0	0	51.5	**252**
Indirect Data	22.6	**169**	0.0	0			33.3	8	0	0	100	2	32.5	**159**

d: Behavioral model for Bubblesort

Figure 9: Error outcomes for the models and workloads

5.3 Results

5.3.1 Error behavior outcome: Figure 9 shows the error outcomes of both models for the two activations used in the experiments. In the structural model, about 80% of all errors manifest as direct Flow or Execution errors. Furthermore, the workload dependency for these errors appears to be low. For the behavioral model, only 30-43% of all errors manifest directly, and a workload dependency is observed for these errors. For the indirect errors, both models exhibit a workload dependency. The major fault classes contributing to the indirect errors are for the structural model, the fault classes Buses and Misc, while the UR fault class is the major contributor to indirect errors in the behavioral model.

Class	Error Latency [instructions]	Structural Model		Behavioral Model	
		Heap	Bubble	Heap	Bubble
Indirect Flow	min	2	2	2	2
	max	13	20	63	129
	median	2	2	8	39
	mean	3.42	2.93	10.96	41.21
	std	2.79	3.37	9.58	31.26
	# of errors	52	42	267	253
Indirect Data	min	1	1	1	1
	max	18	5	105	6
	median	3	3	5	2
	mean	3.17	2.85	8.02	2.27
	std	2.70	0.55	11.25	1.49
	# of errors	72	59	290	169

Figure 10: Indirect errors manifestation latencies

5.3.2 Error manifestation latency: We focus our analysis on the indirect error classes, i.e., errors which propagate through the user register file, as the latency for direct errors is by definition smaller than 2. Figure 10 shows the latency due to indirect errors of both models and activations used in the experiments. Latency is defined here as the time from the injection of a fault to the manifestation of an error on any output port, and is measured by the number of instructions executed between the two events.

For the structural model, the mean latency is rather short (approx. 3 instructions), while for the behavioral model the latency is much longer. In the following paragraphs, we provide an explanation for this observation.

The latency means shown in Figure 10 are estimates of the mathematical expectation $E[l] = \sum_i i \times p(e(o^i))$, where $p(e(o^i))$ represents the probability that an error occurs on an output port when instruction number i is being executed, given that a fault was injected during execution of instruction $i=0$. The differences in the values of the latency obtained for the two models originate from differences in the values of $p(e(o^i))$.

Now, let r represent a register in the user register file, and let r^i and r_i denote a user register used as a destination and source register, respectively, during instruction i. Furthermore, let $e(r)$ denote that r is erroneous. Then, $p(e(o^i))$ can be characterized by:

$$p(e(o^i)) = p\left[e(o^i) \middle| e(r_i)\right] \times p[e(r_i)] \qquad (1)$$

The first term is completely determined by the workload and characterizes the propagation of the error from a user register to an output port. The second term combines two

phenomena: (i) the propagation of the errors among the registers, and (ii) the potential overwrite of errors. This term depends on both the workload and the fault injection method.

For the structural model we assume that the effect of the injected fault during execution of instruction 0 is one of four possible events. The terms $p_1 \ldots p_4$ represent the probabilities of occurrence of each of these events:

- p_1 corresponds to the occurrence of an error in registers r^0 and r $(r^0 \neq r)$ due to incorrect register selection;
- p_2 corresponds to the occurrence of an error in r^0 due to a fault on the data lines;
- p_3 corresponds to the occurrence of an error in r^0 due to a fault on the control line (no latch);
- p_4 corresponds to the occurrence of an error in r $(r \neq r^0)$ due to a fault on the control line (unexpected latch).

As the fault can affect at most two user registers during instruction 0, it can be easily verified that:

$$p[e(r^0)] + \sum_{r^0 \neq r} p[e(r)] - \sum_{r^0 \neq r} p[e(r^0) \wedge e(r)] = 1$$

The three terms can be respectively developed as:
$p[e(r^0)] = p_1 + p_2 + p_3$, $\sum_{r^0 \neq r} p[e(r)] = p_1 + p_4$ and
$\sum_{r^0 \neq r} p[e(r^0) \wedge e(r)] = p_1$.

As the fault locations are uniformly distributed among the set of signals and since the data and register selection lines are much more numerous than the control lines (32+24 vs. 1), it can be stated that both p_3 and p_4 are much smaller than p_1 and p_2. More specifically, if an error propagates into the user register file, it is very likely that it will propagate into the destination register of instruction 0 (i.e., r^0). In other words, at the end of the execution of instruction 0:

$$\forall r^0 \neq r,\ p[e(r^0)] >> p[e(r)] \qquad (2)$$

For the behavioral model, the fault locations are also uniformly distributed. In this case, almost all indirect errors are caused by bit-flips injected into the user registers. For these errors, the error occurrences are totally decoupled from the activity of the system, i.e., there is an equal probability to hit any of the user registers. Therefore,

$$\forall r^0 \neq r \quad p[e(r^0)] = p[e(r)] \qquad (3)$$

Expression (2) implies that faults injected by means of signal manipulation are *likely* to affect a register (r^0) which is, by application of the temporal locality principle [20], *likely* to be used soon. Thus, the mean time until the next access of the affected register will be short.

On the other hand, expression (3) implies that faults injected by variable manipulation affect arbitrary user registers, which leads to a longer access delay as temporal locality does not shorten it.

In our case, since the sorting workloads perform mainly indexed transfers between registers and memory, there is a high probability for the content of an erroneous register to appear either on the data or address buses. Consequently, the first term of expression (1) can be expected to be close to 1 and thus:

$$p(e(o^i)) \approx p[e(r_i)] \qquad (4)$$

As a result, the error latency for both workloads is mainly determined by the access delay, which has been demonstrated to be shorter for errors caused by signal manipulation faults than variable manipulation faults. This explains why the error latency is much shorter for the structural model compared to the behavioral one.

6 Summary and concluding remarks

This paper presents (i) a description of methods to inject faults in VHDL models, (ii) an overview of a simulation-based fault injection tool, called MEFISTO and (iii) a case study where signal and variable-related faults are injected into a structural and behavioral model of a 32-bit processor, respectively.

The case study has proven the capability of both the signal and variable manipulation methods to inject faults in VHDL models. Signal manipulation can be used on a VHDL model, in a similar way as pin-level fault injection is used on physical systems. Using the former technique, it is possible to inject a fault on any signal, not just on the IC pins. The variable manipulation technique also makes it possible to inject faults that are injectable on physical systems; it can be used to simulate the effect of bit-flips in registers, as caused by heavy-ion fault injection. Simulation-based fault injection provides perfect controllability over where and when a fault is injected, in contrast to the heavy-ion technique.

In the case study, the fault effects are classified into two error classes: direct errors, which manifest directly on the output ports, and indirect errors which stay dormant for some time in the user registers before manifesting. The signal-related faults injected into the structural model are shown to result in direct errors in 80% of the cases, while the variable-related faults injected into the behavioral model only result in direct errors in less than 45% of the cases. Furthermore, the error manifestation latency for indirect errors is much shorter for faults related to signals than variables.

The variation in error latency with respect to the fault type is analyzed in detail. It is argued that the faults related to signals will spread inside the processor, as well as to its output ports, more rapidly than those related to variables. It is interesting to note that this line of reasoning most likely can be extended to include a comparison between pin-level (signal) and heavy-ion (variable) induced errors.

Simulation speed and accuracy are important factors to account for when considering fault injection for error propagation studies. The execution times of the sorting programs are much smaller using an instruction-level simulator, as in the experiments (cf. Figure 6), than using a more accurate gate-level simulator, thus making it possible to simulate much longer instruction sequences. Furthermore, it can be noted that the behavioral model is more efficient (faster) than the structural one.

Both the structural and behavioral models are abstractions of the same processor at an abstraction level significantly above the gate and circuit level. It is possible to inject only a subset of all possible bit-flip faults using the behavioral model, as it only includes a subset of all "real" registers (variables) that would be present in a physical implementation of the DP32. Likewise, it is possible to inject temporary stuck-at faults on only a subset of all "real" signals using the structural model. This limits the use of simulation models on these abstraction levels for validation of fault tolerance mechanisms.

It should be noted that all faults which are injectable using SWIFI on a physical system, also are injectable using instruction-level simulation models, such as the structural and behavioral models used here. Clearly, more research is needed to investigate the accuracy of fault injection experiments at these levels (and using SWIFI), with respect to more detailed ones. Such an investigation could possibly identify classes of faults that can be injected at these higher levels without loss of accuracy; MEFISTO is well suited for this type of experiments.

An important point demonstrated by this study is the relative ease in performing fault injection campaigns using VHDL models. This is due to the ability of the used fault injection techniques to take advantage of available VHDL models, without requiring any model modifications.

Further work with MEFISTO will address the early test of fault tolerance mechanisms imbedded in fault tolerant systems. In particular, the favorable error tracing capabilities depicted in this paper will be used to monitor the fault activation/error propagation processes in order to:

- monitor the sensitization of the fault tolerance mechanisms, when the fault injection experiments are aimed at *removing potential fault tolerance deficiency* faults,
- study the mapping between the set of faults in the fault tolerant system and the error models used to test the fault tolerance mechanisms, in order *to evaluate their coverage*.

References

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

[2] R. K. Iyer and D. Tang, *Experimental Analysis of Computer System Dependability*, Univ. of Illinois at Urbana-Champaign, Tech. Report CRHC-93-15, September 1993.

[3] C. R. Yount, *The Automatic Generation of Instruction-Level Error Manifestation of Hardware Faults: A New Fault Injection Model*, Ph.D. Dissertation Thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1993.

[4] K. K. Goswami and R. K. Iyer, *DEPEND: A Simulation-Based Environment for System Level Dependability Analysis*, Univ. of Illinois at Urbana-Champaign, Tech. Report CHRC-92-11, June 1992.

[5] J. A. Clark and D. K. Pradhan, "Reliability Analysis of Unidirectional Voting TMR Systems Through Simulated Fault-Injection," in *Proc. Workshop on Fault Tolerant Parallel and Distributed Systems*, Amherst, MA, July 1992, pp.72-81.

[6] J. Ohlsson, M. Rimén and U. Gunneflo, "A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog," in *Proc. FTCS-22*, Boston, MA, July 1992, pp.316-325.

[7] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson and T. Lin, "FIAT — Fault Injection based Automated Testing Environment," in *Proc. FTCS-18*, Tokyo, Japan, June 1988, pp.102-107.

[8] R. Chillarege and N. S. Bowen, "Understanding Large System Failures — A Fault Injection Experiment," in *Proc. FTCS-19*, Chicago, MI, June 1989, pp.356-363.

[9] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," in *Proc. FTCS-22*, Boston, MA, July 1992, pp.336-344.

[10] J. M. Voas, "PIE: A Dynamic Failure-Based Technique", *IEEE Trans. on Software Eng.*, 18 (8), August 1992, pp.717-727.

[11] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications," *IEEE Trans. on Software Eng.*, 16 (2), February 1990, pp.166-182.

[12] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson and U. Gunneflo, "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms," *IEEE Micro*, 14 (1), February 1994, pp.8-23.

[13] 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, September 1993, pp.454-459.

[14] 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.), Prentice-Hall, 1992, pp.22-145.

[15] A. Miczo, "VHDL as a Modeling-for-Testability Tool," in *Proc. COMPCON'90*, IEEE, February 1990, pp.403-409.

[16] *IEEE Standard VHDL Language Reference Manual*, 1988.

[17] 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 ESPRIT Basic Research Project PDCS-2 Open Workshop*, LAAS-CNRS, Toulouse, France, September 1993, pp.461-483.

[18] 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.), Prentice-Hall, Englewood Cliffs, 1992, pp.240-303.

[19] P. J. Ashenden, *The VHDL Cookbook*, University of Adelaide, South Australia, Tech. Report, 1990.

[20] J. L. Hennessy, D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, 1990.