H2020-COMPET-2014

MaMMoTH-Up

Massively extended Modular Monitoring for Upper Stages

Type of Action: Research and Innovation Action (RIA)

Topic: COMPET-02-2014 Independent access to space

Grant Agreement no: 637616

MaMMoTH-Up
MASSIVELY EXTENDED MODULAR MONITORING FOR UPPER STAGES

Report on proposed dependability evaluation techniques (v. 2.2) (Deliverable D4.1)

Start date of the project: May 1, 2015 Duration: Three years

Organisation name of lead contractor for this deliverable: DLR

Project funded by the European Commission within the Horizon 2020 programme for research, technological development and demonstration (2015-2018)		
Dissemination Level		
PU	Public, fully open	☒
CO	Confidential, restricted under conditions set out in Model Grant Agreement	☐
CL	Classified	☐

Notices

For information, please contact Matteo Sonza Reorda, e-mail: matteo.sonzareorda@polito.it. This document is intended to fulfil the contractual obligations of the MaMMoTH-Up project concerning deliverable D4.1 described in Grant agreement no: 637616.

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
1.1	1.4.2016	Initial version.	M. Sonza Reorda	all
2.0	27.4.2016	Draft complete version	M. Sonza Reorda	all
2.1	28.4.2016	Revised complete version (references updated)	G. Fey, M. Sonza Reorda	all
2.2	29.4.2016	Final version	M. Sonza Reorda	all

Author, Beneficiary

- Serhiy AVRAMENKO, PDT
- Stefano ESPOSITO, PDT
- Matteo SONZA REORDA, PDT
- Massimo VIOLANTE, PDT

Executive Summary

The following document presents the techniques proposed within the MaMMoTH-Up project for dependability evaluation.

.

List of Abbreviations

Acronym	Meaning	Explanation
AF	Acceleration factor	Factor used in dependability analysis to estimate a component's failure rate
ASIC	Application-Specific Integrated Circuit	
ASM	Assembly	Integration on stage
ASMTEST	Assembly Test	ESC checkout, inspection, and stage acceptance
BAF	Batiment d'Assemblage Final	Launcher final assembly building in CSG/Kourou
BDM	Background Debug Mode	
BIL	Batiment d'Integration Lanceur	Launcher integration building in CSG / Kourou
CSG	Centre Spatial Guyanais	Space Center in Kourou with Ariane 5 Launch Pad
CSGSTO	Storage in CSG	
DD	Dangerous Detected	A fault that may have catastrophic consequences and is detected
DSM	Deep sub-micron	
DU	Dangerous Undetected	A fault that may have catastrophic consequences and is not detected
EDA	Electronic Design Automation	
EPC	Etage Principal Cryotechnique	
ESC	Cryogenic upper stage	Etage Supérieur Cryotechnique
ESCSTO	ESC Storage in Europe	
EUSTO	Europe Storage	Component's storage in Europe before assembly
FIT	Failure in Time	Failures in a billion hours
FISB	Fault Injection Support Board	
FI-VHDL	Fault Injection using VHDL	
FI-VP	Fault Injection using Virtual Platforms	
FPGA	Field Programmable Gate Array	
HDL	Hardware Description Language	
HF	High Frequency	
I/O	Input/Output	
LASM	Launcher integration in BIL	
LASMTEST	Launcher Integration Test	

Acronym	Meaning	Explanation
PASM	Payload Assembly	Final preparation and payload assembly
PASMTEST	Payload Assembly Test	Inspection, test, and preparation before payload assembly
PCB	Printed Circuit Board	
PCI	Peripheral component interconnect	
PREPZL	Preparation in ZL	Preparation for final chronology in ZL, propellant tank loading, Helium loading, stand-by phase, propellant topping and monitoring before launch, final count-down.
RAM	Random Access Memory	
RF	Radio Frequency	
RTL	Register-Transfer Level	
SD	Safe Detected	A fault that has no catastrophic consequences and is detected
SU	Safe Undetected	A fault that has no catastrophic consequences and is latent in the system
TRANSP1	Transport 1	Transport to stage integration location
TRANSP2	Transport 2	Transport to CSG
TRANSP4	Transport 4	Transport from BAF to ZL
USB	Universal Serial Bus	
ZL	Zone Launcher	Launcher Pad

Table of Contents

1 Introduction

When designing, manufacturing and delivering an electronic system for any mission- or safety-critical application it is crucial to evaluate its dependability, so that we can check whether the obtained figures match the expected ones. In the negative case, dependability analysis can also be adopted to identify the most critical parts of the design, so that we can improve them and thus more easily achieve the target dependability figures.
Dependability analysis is based on the available information about the target system, as well as on its mission (including the environment it is expected to work in) and its life cycle.
This kind of analysis is particularly critical when COTS-based components are adopted, since the amount of information about their reliability is often more limited than for space-qualified components. Hence, the meaningfulness of the analysis strongly depends on how detailed the information about them (either provided by the manufacturer or extracted somehow) are.
Dependability evaluation can typically be split in two parts:

- In the first we evaluate the probability that a fault affects any component in the system
- In the second we evaluate the probability that the effects of the faults evolve into a failure (taking also into account the possible fault detection and recovery mechanisms), weighting failures depending on their severity.

The goal of this document is first to describe the procedure proposed to evaluate the dependability of the MaMMoTH-Up system (Section 2). Secondly, since the assumptions about the effects of faults arising in the system should be validated experimentally, we propose to resort to some fault injection solution. For this reason, we overview in Section 3 the most important techniques for performing fault injection.

Section 4 draws some final conclusions.

2 Dependability analysis

This section outlines the reliability analysis of the MaMMoTH-Up system. This analysis aims first at computing the Failure in time figure following the FIDES guide [1]. The section is organized as follows. Sub-section 1 describes the procedure detailed in [1] to determine the Fault Rate of a system. Sub-section 2 describes how this procedure has been applied to the OBC-S board, which is a major component of the target system. Sub-section 3 describes how the computed failure rate can be refined considering the failure modes of the components. Finally, Sub-section 4 draws some conclusions.

2.1 *Fault rate determination*

The fault rate has been determined using the FIDES guide and the FIDES software that supports the process described in the guide. In the following of this section, the steps that were followed according to the FIDES guide are detailed.

2.1.1 Process quality determination

The FIDES guide defines some metrics to quantify the quality of the production process. Such detailed evaluation can be avoided, assigning average values to all the metrics, at the cost of some loss in the precision and quality of the resulting reliability figure. A more accurate method to quantify these metrics would be an audit performed on the design and manufacturing company by a third party to quantify the respect of several reliability management policies as adopted in various phases of a product design, production and support. If an audit is not feasible, the default averages recommended by the FIDES guide can be used (as we did for the work reported here).

2.1.2 Product Life Profile

The life profile of the product subject to analysis must be carefully described, as recommended by the FIDES guide. Different phases of the life of a product must be defined to capture and quantify the different stresses to which the product is subjected during its lifetime.
Each phase quantifies a series of physical and chemical stresses:

1. **Thermal Stress:** the thermal stress is modeled as a sequence of temperature cycles. A cycle is the variation of the temperature from a reference temperature and the restoration of the reference value. Each cycle is represented by a duration in hours, a maximum temperature and a maximum difference in temperature. Each phase can contain several temperature cycles, although the duration of the phase is not forced to be a multiple of the duration of the cycles.
2. **Relative Humidity:** the relative humidity is the ratio of the water vapor pressure to the water vapor saturation pression, which depends on the temperature. The value is expressed as a percentage and is on average 70%.

3. **Mechanical Stress (vibrations):** is expressed as the energy of the vibrations to which the system is subject. It can be computed in the frequency spectrum as the square root of the area described by the amplitude of the vibration signal.
4. **Chemical stress:** chemical stresses are only qualitative and take into consideration the corrosive action of the salinity in the air, which is higher in coastal regions and lower in regions far from the coast, the contamination of environmental pollutant and pollutant deriving from the application in which the product is used, and the protection level of the product, that can be hermetic or non-hermetic. A hermetic protection can protect from salinity and pollutants, but humidity can accumulate in a hermetic container.
5. **Application: [1]** states that a questionnaire should be used to quantify several aspects of the products applications. Due to lack of specific information, the default values suggested in [1] are used in this analysis. For reference, the items proposed in [1] are:
    a. **User type:** represents professionalism, respect of procedures, influence of operational stresses.
    b. **User qualification:** represents the control level of the user or the operator in an operational context.
    c. **System mobility:** represents problems related to the possibilities of system movement.
    d. **Product manipulation:** represents the risks of false manipulations, shocks, falls, etc.
    e. **Power supply type:** refers to the electrical disturbance level expected on power supplies and signals: power on, power supply switching, connection/disconnection.
    f. **Exposure to human activity:** represents exposure to problems related to human activity: shock, change in the final use, etc.
    g. **Exposure to machine disturbances:** represents problems related to functioning of machines, motors, actuators such as shocks, overheating, electrical disturbances, aggressive pollutants.
    h. **Exposure to the weather:** represents exposure to rain, hail, frost, sandstorm, lightning, dust, etc.

Each of these parameters is used to define a specific acceleration factor (AF), which modifies the failure rate of the product. An additional coefficient is defined, that is a quantification of the effort in the production process to enhance the product reliability. As specified in the FIDES guide, it can be quantified by evaluating how several recommendations specified in the guide have been followed during the production process.

2.1.3 Components

Each component has to be taken into account for the failure rate estimation. The FIDES guide defines several models to compute the failure rate of each component type. Once the type of component has been specified, several parameters must be inserted into the model to allow failure rate calculation. The parameters for each component type are:

- **Connectors:** Number of contacts, number of cycles (connection/disconnection) per year, maximum rise in temperature;
- **Integrated circuits (including FPGAs):** number of pins, maximum rise in temperature;
- **Resistors:** ratio between dissipated power and rated power. For SMD resistors, also the number of networks sharing the same case;
- **Capacitors:** ratio between applied voltage and rated voltage**;**
- **Discrete semiconductor devices:** number of discrete semiconductor devices sharing the same case, maximum rise in temperature
- **Inductors:** maximum rise in temperature.

All parameters that can vary during product’s life can also be specified with different values for each life period, although this is an optional action.

2.2 Failure Rate evaluation using FIDES guide

In this section the process used to evaluate the Failure In Time (FIT) figure of the system is described, according to the steps described in the previous section.
The FIDES guide requires to describe the life profile of the system in order to perform the analysis. The reader should refer to the FIDES guide for an exhaustive description of the models used to compute the FIT figure for each component and the parameters used in these models. The following list explains the acronyms used in the rest of the document as name of the phases. The phases are coherent with the life profile described in [4].

- EUSTO: Storage in Europe
- TRANSP1: Transport to Stage Integration Location
- ASM: Integration on BMA
- ASMTEST: ESC checkout + inspection + stage acceptance
- ESCSTO: ESC Storage in Europe
- TRANSP2 : Transport to CSG
- CSGSTO: Storage at CSG

- LASM: Launcher Integration in BIL
- LASMTEST: Inspection, tests, and preparation after integration on launcher
- TRANSP3: Transport from BIL to BAF
- PASMTEST: Inspection, tests, and preparation before payload assembly
- PASM: Final preparation and payload assembly
- TRANSP4: Transport from BAF to ZL
- PREPZL: Preparation from final chronology in ZL; Propellant tank loading; Helium loading; Stand-by, propellant topping, and monitoring before launch, Final count-down.
- LAUNCH: Cryogenic arms withdrawal, EPC engine ignition sequence, Flight.

Each phase is considered as ON or OFF depending on whether the system is connected to power or not during the phase. The last column is the duration in hours of the phase. All parameters required by the FIDES guide, as specified in section 2.1.2, will be extracted from the requirements documents.
The FIDES guide also recommends a procedure to define a value for $\pi_{application}$ and $\pi_{ruggedising}$, which are two parameters used in the FIT computation. If specific information is not available, the default value recommended by the guide can be used, which are $\pi_{application}$=1.9 for all phases and $\pi_{ruggedising}$=1.7.

Once these preliminary steps are performed, an analysis of the components of the system is needed. All the components included in the Bill-Of-Materials (BOA) will be considered for this evaluation, including the PCB and the connectors. Each component will be analyzed separately, using information available from the schematics and from specific analysis performed on a real system to compute electrical parameters. To each component a type must be associated, selected from the following list:

- Resistor
- HF/RF Resistor
- Ceramic Capacitor
- Tantalum Capacitor
- HF/RF Capacitor
- Discrete Semiconductor
- Integrated circuit
- Connector
- PCB.

Each type of component has several subtypes. Each component must be assigned to the correct subtype using information available from the component's datasheet. A final parameter to be assigned to each component is the *function*, which can be assigned from information available on the schematics.
Some parameters must be assigned to each component, as specified in [1] and reported in section 2.1.3. For passive components, the parameters can be computed using circuit analysis.
For integrated circuits, some parameters can be extracted from the component's datasheet, i.e., number of pins and type of package, while the raise in temperature during operation can be computed using the thermal model suggested in the FIDES guide or using measurements performed on the actual system. The model requires an evaluation of the dissipated power and provides values for thermal resistance. The values in the guide can be used when the thermal resistance is not available from the component's datasheet. The evaluation of the dissipated power can be performed in three ways:

- When possible, the dissipated power of a component can be evaluated extracting available information from the component's datasheet. This is primarily the case of discrete semiconductors components.
- When available on the component datasheet, the typical used current can be used to evaluate the power, using the equation $P = V \times I$. The voltage can be extracted from the schematics or from measurements performed on the actual system.
- As a last resort, a maximum power rating from the component datasheet can be used. This was the cause for the FPGA and the connectors.

A slightly different model is provided for connectors in [1]. The model is $\Delta T = \alpha I_c^{1.58}$ where α is a parameter dependent on the number of pins in the connector and is provided in a table in [1], and I_c is the current on each pin of the connector. Values of α not available in the table can be obtained by linear interpolation. The value for I_c has been set as the maximum current rating in the connector datasheet. The connection/disconnection cycles for the connectors was set to 4 for debug connectors and 2 to all other connectors.
Once all the parameters are provided, the FIDES tool can extract a FIT rate for the system. Once this rate is available, the un-reliability can be computed using the formula reported below.

$$F(t) = 1 - e^{-\lambda t}$$

where λ is expressed in hours and is obtained multiplying the FIT by 10^{-9}.

2.3 Failure mode impact evaluation

In order to perform the required analysis, the failure rate figure calculated through the FIDES guide as described in the previous section is not sufficient. It has to be considered that each failure can cause different kinds of misbehavior. For instance, a failure in a resistor can transform the resistor in an open circuit or in a short circuit, or it can modify the value of the resistance. Each of these scenarios can have different effects on the system and as such must be considered separately. Moreover, each misbehavior is not equally probable. The probability distribution of each failure mode can be found in standard publications and in the scientific literature. Once the failure

mode distribution has been evaluated, each failure mode contribution to the system failure rate can be considered.
Each fault of each component can be classified as

- **Safe:** the fault does not compromise any safety goal
- **Dangerous:** the fault compromises one or more safety goals.

Each failure can also be classified as:

- **Detected:** the fault can be detected by the system
- **Undetected:** the system cannot detect the fault.

Therefore, there are four classes of failure modes to be considered in the analysis:

- Safe and detected (SD)
- Safe and undetected (SU)
- Dangerous and detected (DD)
- Dangerous and undetected (DU).

2.4 Conclusions

The Failure Rate has been evaluated and a first unreliability figure was computed. The FMECA can be continued by performing the analysis described in section 2.3. Once concluded, it will produce the following metrics:

- *Single point fault* metric, which is the contribution of DUs and DDs.
- *Latent fault* metric, which is the contribution of SUs.

3 Fault Injection techniques

The goal of this Section is to provide an overview of the main Fault Injection techniques proposed so far in the literature or adopted in practice.
Sub-section 1 introduces the general topic and provides further motivations for fault injection. Sub-section 2 reports the general architecture of a Fault Injection environment. Sub-sections 3 to 5 outlines the key characteristics of simulation-based, emulation-based and software-based fault injection systems, respectively. Sub-section 6 draws some conclusions.

3.1 Introduction

Designers of electronic systems often rely on simulation and emulations tools for their work: the common design practice entails the building of models of the system being designed that are simulated to check whether the design solutions meet the user requirements. Nowadays, a top-down design flow is typically used in many cases: starting from a very abstract model of the system, which captures the behavior (i.e., the algorithm) the system has to implement, details are added by refining iteratively the model, eventually obtaining a cycle-accurate description of the structure of the system. The system model can be emulated using an appropriate hardware platform to verify whether the design actually meets all the user requirements, and can be used for manufacturing of the system. When a reasonable confidence about the correctness of the model is reached, the manufacturing can start. Thanks to this approach, shorter time-to-market and lower costs can be achieved, as design bugs and inconsistency with user requirements can be captured when production is not yet started.
The Electronic Design Automation (EDA) industry offers designers a wide range of products that support this top-down design flow. Hardware description languages (HDL) are available (like for example VHDL, Verilog [5], and SystemC [6]) that can be used for describing systems, and a number of HDL simulators are available for studying the dynamic behavior of the obtained models. Both the description languages and the simulation tools support different abstraction levels and domains of representations, so that designers can start reasoning on the system behavior during the early design stages, and following the top-down flow can end up with a structural description of the system which is represented as a netlist connecting logic gates.
Beside simulation tools, a number of EDA solutions are available for supporting designers in the refinement of the model. Tools are available for analyzing of the partitioning between hardware and software implementations of the system functions, and synthesis tools are available to implement in an automatic fashion the transitions from higher, less detailed, abstraction levels/representation domains to lower, mode detailed, abstraction levels/representation domains.
The EDA tools are mostly focused on the analysis and the synthesis of the functionalities the system must implement to fulfill the user requirements. If we limit our analysis to simulation tools, we can see that they provide efficient ways for evaluating how the model react to input stimuli representing the typical workload the system has to process, and they provide advanced debug features to simplify the identification of design bugs or part of the model that does not fulfill adequately the user requirements. Moreover, they can provide timing and power information to simplify the tuning of parameters like speed and power consumptions.
Nowadays, simulation tools are essential ingredients of any successful design flow, and they are widely adopted in industries because they provide the type of support designers need: the capability of producing useful information, and to deal with ever

growing designs by exploiting clever simulation algorithms, and possibly taking advantage of dedicated hardware emulators.

To continue to provide useful support to designers of electronic systems, we expect that in the coming years simulation tools will start providing features related to dependability evaluation.

The concept of dependability, which can be seen as the property of an electronic system of being able to deliver service that can justifiably be trusted [7], is well known to developers of electronic systems employed in mission- or safety-critical applications, where failures may lead to loss of money or human lives. An essential part of the design flow of such kind of systems is indeed the dependability evaluation process, which must provide evidence of the capability of the system to fulfill the dependability requirements. Since a few years ago, dependability and dependability evaluation were mostly bound to very specific application domains, like space, military, medical, and transportation. With the advent of deep sub-micron (DSM) manufacturing technologies, things are changing and dependability is becoming an important concept also for developers of commodity applications. Moreover, faulty scenarios that were bound to very specific application domains (e.g., space, nuclear science, and medical) are becoming more and more important as DSM technology becomes widely used.

Among the different techniques for performing dependability evaluation, the one which is most suitable for being introduced in the current design flows is fault injection [10]. The concept of fault injection consists in inoculating a fault in a system, and observing how the fault propagates within it, eventually reaching the system outputs. This concept perfectly fits with the purpose of simulation tools that offer the capability of studying the dynamic behavior of a system model. Fault injection can be provided as an additional feature of simulation tools, which will be able to allow designers studying the dynamic behavior of system models when affected by faults, along with the more traditional and more established features oriented to design debug, performance and power evaluations.

3.2 Overview of a fault injection system

The basic components of a fault injection system are depicted in Figure 1.

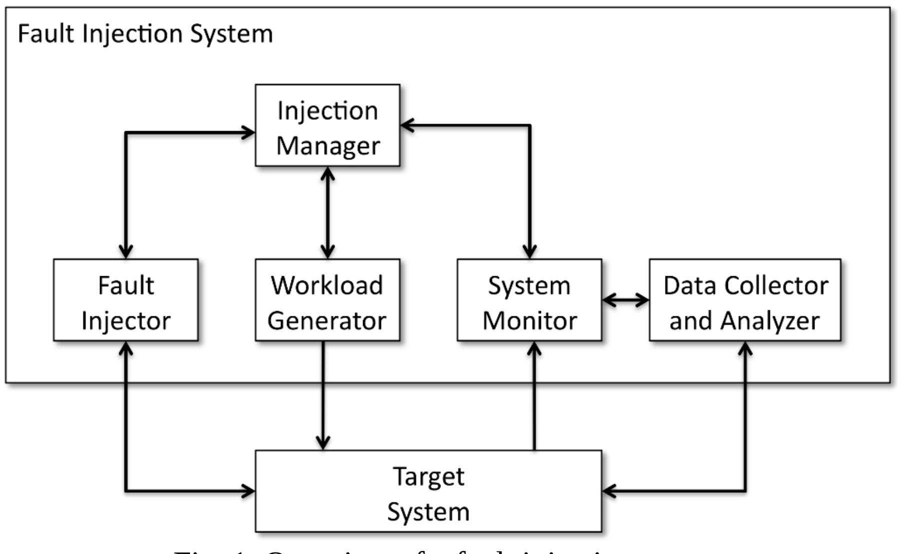

Fig. 1. Overview of a fault injection system

The target system is the object of the investigation, where faults will be inoculated and whose behavior will be monitored to study the impact of the injected faults. As this discussion is general, we do not refer to a specific type of target system; for the sake of this section the target system can be either a behavioral model of an electronic system, a structural model, a hardware-emulated model, or even a prototype implementing the functionalities of the model.

With the term fault we refer to the malfunctions the target system may be subject to during its lifetime, which may let the target system behavior deviates with respect to the intended one. The malfunctions that affect a system may be:

- *Permanent*, in case the malfunction always affects the system
- *Transient*, in case the malfunction affects the system when a certain set of conditions is met. Otherwise, the system functions normally.

During injection, fault models are used to capture such malfunctions, which are described using the same modeling approach used for the target system. Among them, the most widely used is the *single stuck-at*, which models a permanent malfunction that may affect a circuit by tying permanently to one or to zero one net of the structural model of the target system.

In more recent years a new fault model, the *bit-flip* fault model, is becoming to be used in the industry to represent the malfunctions provoked by the charge deposition induced by a radioactive particle in a semiconductor device [11]. The bit-blip fault model consists in the random mutation of the content of one memory element of a design, which changes its content from 0 to 1 (or vice-versa). Bit-flips are random in both time and space: they can affect a system during its entire lifetime, and they can strike any of the memory elements the system embeds. It is worthwhile to remark here that the bit-flip fault model refers to the modification of the stored information that is corrupted, and not of the storage cell that preserves its correct functionality.

The main components of a typical fault injection system are:

1. *Injection Manager*: it supervises the injection campaign. Given the list of faults that has to be inoculated in the system, the *fault list*, it runs an injection experiment for each of them. Each experiments encompasses the following step:
    a. The first fault *f* to be injected is selected from the fault list.
    b. The target system, the workload generator, the data collector and analyzer are set to the reset state.
    c. The fault injector is programmed to inoculate the fault *f*, and the system monitor is programmed accordingly. For example, let suppose the fault *f* has to be injected in the target system at time t_f, being t_0 the time when the first input stimuli is applied. The fault injector is instructed to stop the system at time t_f, to inoculate the fault (accordingly to the type of the target system), and to resume the target system after fault injection to let the fault to propagate. Moreover, the system monitor is programmed to trigger data collection from time t_f until the end of the workload.
    d. The workload generator is activated. The input stimuli are applied to the target system; in accordance to step (c) fault *f* is injected in the target system, and data are collected from t_f onward.
    e. Upon the completion of the workload, the fault effect is classified, and the whole procedure is repeated from step (a).
2. *Fault Injector*: it inoculates a fault in the target system as it activated by the by the workload generator. As detailed in the following sections, a number of techniques

can be used to achieve fault inoculation, depending on the type of the target system.
3. *Workload Generator*: it generates the input stimuli to activate the target system during the fault injection experiment. The input stimuli can be synthetic workload generated ad-hoc, or real inputs taken from the application where the system will be deployed.
4. *System Monitor*: it observes the target system and when necessary it triggers the collection of data from the target system.
5. *Data Collector and Analyzer*: when triggered by the monitor, it collects from the target system data that are useful to classify the impact of the inoculated fault. For example it can collect the outputs produced by the target system, as well as status information. The collected data are then processed by the data analyzer, which produces the classification of the fault effect. This task is normally performed by comparing the data collected on the faulty system with those produced by the fault-free system when activated by the same workload used during the injection experiment.

A number of different implementations are possible for the fault injection system, which can be grouped in different categories as a function of the type of the target system. We can have:

1. *Simulation-based fault injection*. This category collects all the methods that have been developed to inoculate faults in a model of the target system whose dynamic behavior is evaluated through the use of simulations tools. The category can be further divided in sub-categories, by considering the abstraction level at which the target system is modeled:
    a. *System-level simulation*, where the system is described in terms of complex components like processors executing software, memories, I/O peripherals, possibly connected through a network infrastructure. This abstraction level is suitable for modeling complex systems as for example a cluster of computers that build a server farm.
    b. *Register-transfer-level simulation*, where the system is described in terms of components like registers, arithmetic and logic units, caches, etc. This abstraction level is suitable when the target system is a component in a large infrastructure, as for example one of the network card of a computer in server farm.
    c. *Gate-level simulation*, where the system is described in terms of logic gates, and simple memory elements. This abstraction level is suitable when the target system is a component in a larger infrastructure, as for example a protocol controller chip inserted in a network interface card.
2. *Emulation-based fault injection*: where the system is first described as in the register-transfer-level case, and it is then emulated using dedicated hardware, like for example, Field Programmable Gate Arrays (FPGAs)-equipped boards. This sub-category can be seen as an evolution of register-transfer-level simulation, where hardware emulation is exploited to boost the simulation performance. Indeed, when very complex workloads, and very large fault lists have to be considered, the time spent for each fault injection experiment can be prohibitive, and means to reduce it are needed.
3. *Software-based fault injection*: where the system is a physical model, i.e., a prototype, composed of processors, memories and I/O peripherals, possibly connected through a network infrastructure. Fault injection is implemented by means of specially crafted software that is added to the software the target system

executes to implement the desired functionality. This sub-category is intended for performing dependability evaluation when the prototype of the target system is available, and can be applied only to processor-based systems.

In the following sub-sections we will discuss in further details the different categories of fault injection systems.

3.3 Simulation-based fault injection

With the term simulation-based fault injection we refers to all those methods that have been developed to inoculate faults in a model of the target system whose dynamic behavior is evaluated through the use of simulations tools. Referring to Fig. 1, we have that:

- The target system is an executable model written in a description language (e.g., VHDL, Verilog, or even C/C++). Depending on the phase of the design flow when fault injection is used, the model can be at the system-, or RT-, or gate-level. Lower levels (transistor-, or device-level) are possible, but they will not be considered in this chapter. A test case where fault injection was performed at even higher level of description (e.g., Modelsim models) is reported in [25].
- The workload generator is normally a testbench [12] for the target system model, and it is applied to the target system by exploiting a simulation tool. Commercial off the shelf tools can be used for this purpose, which are the same used during the normal design practice.
- Injection manager, fault injector, system monitor and data collector and analyzer are ad-hoc software. They can be stand-alone software modules that run on top of commercial off the shelf simulation tools, or they can be integrated within the simulation tool in case it provides dependability-oriented features. In particular, fault injection can be implemented as follows:
    - The simulation tool is enriched with algorithms that allow not only the evaluation of the faulty-free target system, as normally happen in VHDL or Verilog simulators, but also their faulty counterparts. This solution is very popular as far as certain fault models are considered: for example commercial tools exist that support the evaluation of permanent faults like the stuck-at or the delay one [13]. Conversely, there is a limited support of fault models that represents radiation-induced faults, and therefore designers have to rely on prototypical tools either built in-house or provided by universities.
    - The model of the target system is enriched with special data types, or with special components, which are in charge of supporting fault injection. This approach is quite popular since it offers a simple solution to implement fault injection that requires limited implementation efforts, and several tools are available adopting it [14][16][17][18].

Both the simulation tool and the target system are left unchanged, while fault injection is performed by means of simulation commands. Nowadays, it is quite common to find, within the instruction set of simulators, commands for forcing desired values within the model [19]. By exploiting this feature it is possible to support a wide range of fault models.

3.3.1 An example of fault injection using system-level simulation

To provide an example, this section describes the FI-VP (Fault Injection using Virtual Platforms) tool developed at Politecnico di Torino for performing the injection of bit-flips in target systems described at the system abstraction level. The purpose of the

tool is to perform dependability analysis at that step of the design flow where the system architecture has been established, potentially exploiting the application software the system will run, but a prototype of the system is not available yet. The system is modeled as a structure of interconnected components that can be described according to different styles. They can be instruction set simulators for processor cores, behavioral/structural models of standard components like memories, network cards, processor bridges, etc., and user-defined behavioral/structural models of custom hardware. System simulation is demanded to the Virtutech's SimIcs tool [19].
Fig. 2 depicts how the generic structure of a fault injection system has been customized while implementing the FI-VP tool. A custom program coded in C language implements the fault injector, and the system monitor, while workload generator and target system are modeled and executed using Virtutech's SimIcs. The data collector and analyzer is a user-provided C function that is called by the FI-VP core at the end of each fault injection experiment, to implement fault effect classification.
For each injection experiment the fault injector generates a command script file that tells SimIcs at which time the fault has to be injected, and how to perform fault injection (the commands SimIcs offers run/stop the model, and to alter the content of any simulation object are used for this purpose). The outputs of the target system are collected (through SimIcs commands) and are stored in a trace log file, which is processed at the end of the simulation by the data collector and analyzer.
We adopted such architecture as any target system has its own peculiarity, and it is up to the user to define which outputs and which status information have to be collected, and used for fault effect classification. When preparing the fault injection campaign, the user is thus asked to provide a C function, which is linked to the FI-VP core, to process the trace log generated by each fault injection.
For implementing data analysis, before starting the fault injection campaign, the fault-free system is executed once, collecting the reference trace log to be used during fault effect classification.

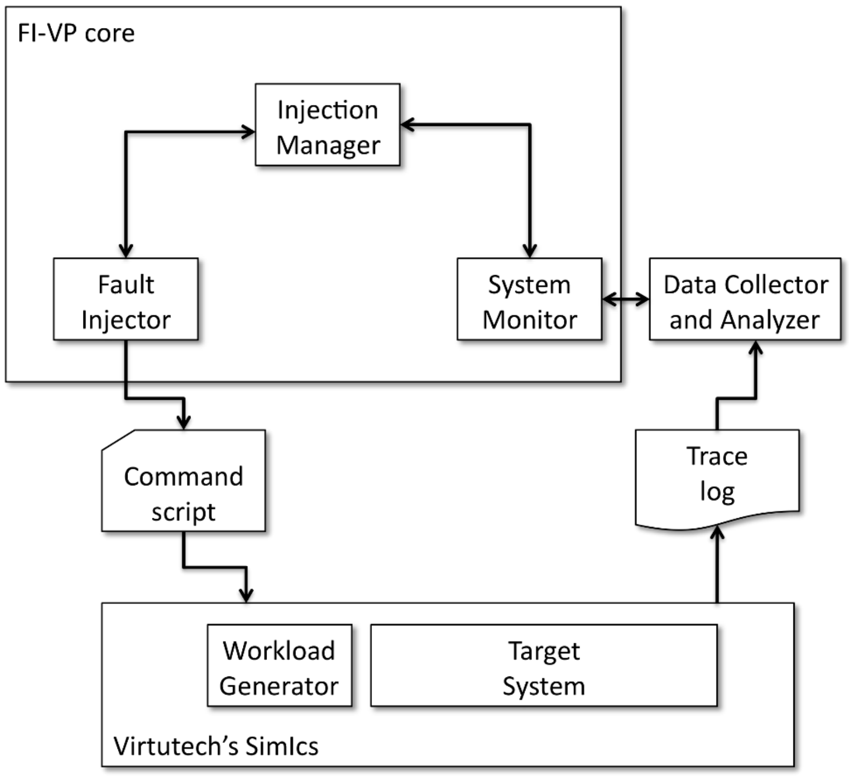

Fig. 2. Architecture of the FI-VP tool

The main benefit of the architecture lies in is generality. Any model can undergo to fault injection, reusing most of the FI-VP architecture as is, with the only exception of the data collector and analyzer module that has be provided by the user.

The FI-VP was used to perform fault injection in a target system composed of one PowerPC440GP, and 128 Mbytes of RAM. The target system runs a Linux 2.6 operating system, and an application that performs 10,000 times the product of two 10x10 matrices. When injecting 10,000 faults in the processor program counter, we recorded the following fault effects.

- 32.94% of injected faults do not affect the system behavior. The workload produces the expected results.
- 26.18% of the injected faults corrupt the matrix multiplication application, which produces results different from the expected one.
- 10.20% of the injected faults lead to a CPU trap, indicating that the error triggered an error detection mechanism the PowerPC embeds.
- 30.59% of the injected faults corrupt lead to operating system crash.

3.3.2 An example of fault injection using register-transfer-level simulation

To provide an example, this section describes the FI-VHDL (Fault Injection using VHDL) tool developed at Politecnico di Torino for performing injection of bit-flips in target systems described at the register-transfer abstraction level. The purpose of the tool is to support designers in developing dependable components to be implements as ASICs of FPGAs. The system is modeled as a structure of interconnected modules described using the VHDL language either at the behavioral, data flow, or structural representation domain. Target system simulation is demanded to the Mentor Graphics Modelsim tool [20]. The architecture of FI-VHDL is the same of FI-VP, while SimIcs is replaced with Modelsim.

In case of register-transfer-level simulations, injection campaigns may require huge amount of time (many hours or days) for their execution, depending on the complexity of the model of the target system, the efficiency of the VHDL simulator adopted, of the workstation used for running the experiments, as well as the number of faults that have to be injected. In order to overcome this limitation, in [18] a technique was proposed, aiming at minimizing the time spent for running fault injection. The technique encompasses three steps:

1. *Golden-run execution*: the target system is simulated without injecting any fault and a trace log file is produced, gathering information on the target system behavior and on the state of the simulator.
2. *Static fault analysis*: given an initial list of faults that must be injected, by exploiting the information gathered during golden-run execution we identify those faults whose effects on the target system can be determined a-priori, and we remove them from the fault list. Since the injection of each fault encompasses the simulation of the target system, by reducing the number of faults that we need to inject we are able to reduce the time needed by the whole experiment.
3. *Dynamic fault analysis*: during the injection of each fault, the state of the target system is periodically compared with the golden run at the correspondent time instant. The simulation is stopped as early as the effect of the fault on the target system becomes known, e.g., the fault triggered some detection mechanisms, the fault disappeared from the target system, or it manifested itself as a failure. Although the operations needed for comparing the state of the target system with that of the golden run come at a not-negligible cost, the benefits they produce on the time for running the whole experiment are significant. In general, a fault is likely to manifest itself (or to disappear) after few instants since its injection. As a result, by monitoring the evolution of the fault for few simulation cycles after its injection, we may be able to stop the simulation execution in advance with respect of the completion of the workload. We can thus save a significant amount of time. Similarly, in case the fault is still latent until few simulation cycles after its injection, it is likely to remain latent, or manifest itself, until the completion of the workload. In this case, the state of the target system and those of the gulden run are no longer compared, thus saving execution time, until the end of the injection experiment.

3.3.3 Final remarks on simulation-based fault injection

Simulation-based fault injection is a powerful technique that we expect to appear in the next years as additional feature of simulation tools, as we expect that dependability evaluation will become a primary design issue. In particular, we expect that simulation-based fault injection will be used to debug and validate the fault detection and correction embedded in future designs target DSM technology.

As simulation is a slow process, and the complexity of designs is ever growing, we expect that simulation-based fault injection will be mainly used for injecting a small set of carefully selected faults for outlining possible design bugs. The adoption of a top-down design flow, based on exploiting system-level simulation for analyzing complex infrastructures, and demanding to lower level simulations the analysis of detailed models of infrastructure components, can alleviate the simulation speed problem.

However, when very complex workloads and huge amount of faults have to be injected we will reach the limit of the capability of simulation-based fault injection. Simulation time will be unfeasible, even in case of very abstract system models, and alternative solutions to speed-up injection campaigns will be needed. In the next sub-section we will present the concept of emulation-based fault injection, along with an example, which shows its effectiveness as a solution to the simulation speed problem.

3.4 *Emulation-based fault injection*

As the complexity of target systems is ever growing, and so is the execution time needed for running simulation-based fault injection campaigns, several researchers proposed to boost performance by running the target system in hardware, instead of using simulation tools. For this purpose Field Programmable Gate Arrays (FPGAs) are used to implement a prototype of the target system, so that workload evaluation is done using actual hardware [21][22][23]. The conceptual architecture of an emulation-based fault injection system is outlined in Fig. 3.

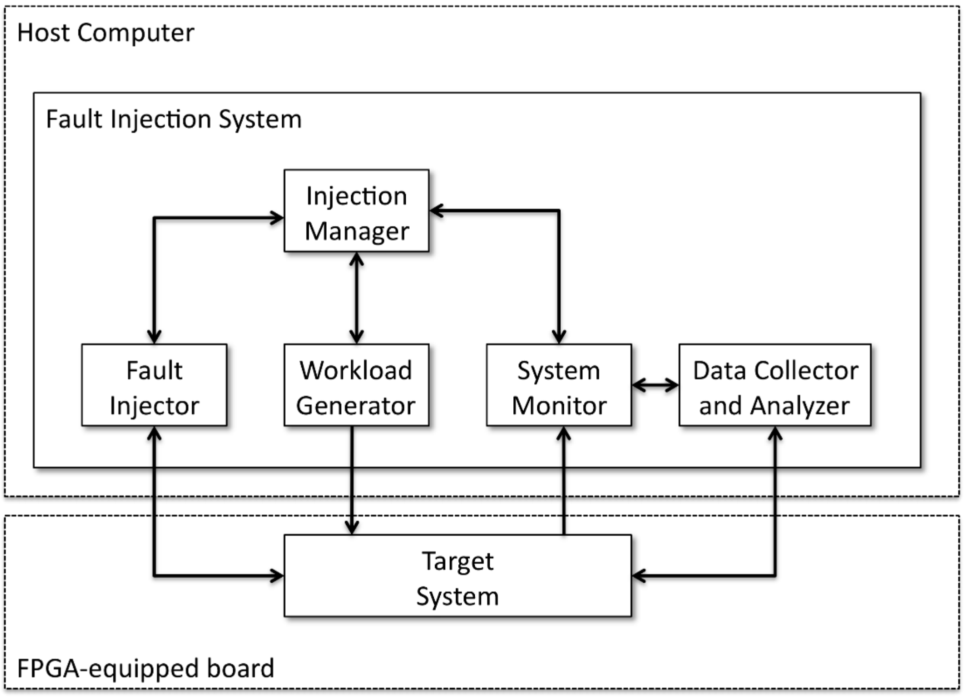

Fig. 3. Conceptual architecture of an emulation-based fault injection system

A host computer runs the software modules implementing the fault injection system, while an FPGA-equipped board emulates the target system. The idea is potentially very effective as the speed ratio between the simulation of a model of the target system and hardware emulation of the same model can be several orders of magnitude.

According to the conceptual architecture of Fig. 3, input stimuli, the collected data, and the operations needed to implement fault injection have to travel from the host computer to the FPGA-equipped board (and vice-versa). As a result, the communication link between these two components becomes the bottleneck, which may limit the actual performance improvement.

As far as the operations needed for fault injection are considered, two options have been proposed so far:

- Insert into the target system features to support the inoculation of the fault model of concern. For example, the approach presented in [22] replaces every memory element of the device with a special cell that offers fault injection capabilities. Thanks to this approach every FPGA model can be used for target system emulation, but modifications to it have to be inserted. As a result, this approach is applicable only when the model of the target system is accessible and modifiable, while it cannot be exploited when the model includes encoded intellectual property (IP) cores. Moreover, issues can be raised on the intrusiveness of the method: as the model of the target system that undergoes to fault injection is different from that of the system that will be deployed in the application, efforts have to be spent in validating the representativeness of the attained observations.

- Adopt the features of the FPGA device used to emulate the target system. Approaches like [23] and [24] exploit the reconfiguration capabilities of Xilinx FPGA for inoculating different types of faults. The main benefit of this approach lies in the capability of performing fault injection without altering the structure of the target system; therefore, it is possible to inject faults also in models that embed encoded IP cores, and the representativeness of the achieved result is less questionable, as no modifications to the model of the target system are required.

As far as the communication link bottleneck is considered, two solutions have been analyzed:

- Adopt high-speed communication bus, like USB 2.0, or PCI to maximize the transfer rate between the host computer and the FPGA-equipped board. This solution is preferable when commercial-off-the-shelf boards are used for implementing the FPGA-equipped board. The architecture remains the same of Fig. 3, and the FPGA is used for emulating the target system only.
- Move workload generator, system monitor and data collector and analyzer to the FPGA-equipped board, by exploiting a custom-designed board. Thanks to this approach, which is typically more expensive than the previous one as it may requires the design of custom equipment, it is possible to optimize the communication link used to deliver input stimuli to, and to collect output data from the target system. To improve further the performance, the fault injector is normally moved to the FPGA-equipped board, thus leaving to the host computer only the supervising task implemented by the injection manager.

3.4.1 An example of emulation-based fault injection

As an example of emulation-based fault injection we present FT-UNSHADES [23]. The tool was developed at the School of Engineering of University of Seville with the support of European Space Agency to perform the analysis of the effects of single bit-flips in designs intended for space applications.

The tool supports the emulation of target systems modeled as IP cores, hence limited knowledge of the internal structure of the target system is needed for running injection campaigns; moreover, no modifications are needed to the target system to support fault inoculation, and thus FT-UNSHADES can be credited to provide accurate analysis on the faulty behavior of the actual system when deployed in the field.

FT-UNSHADES implements the conceptual architecture of emulation-based fault injection as depicted in Fig. 4.

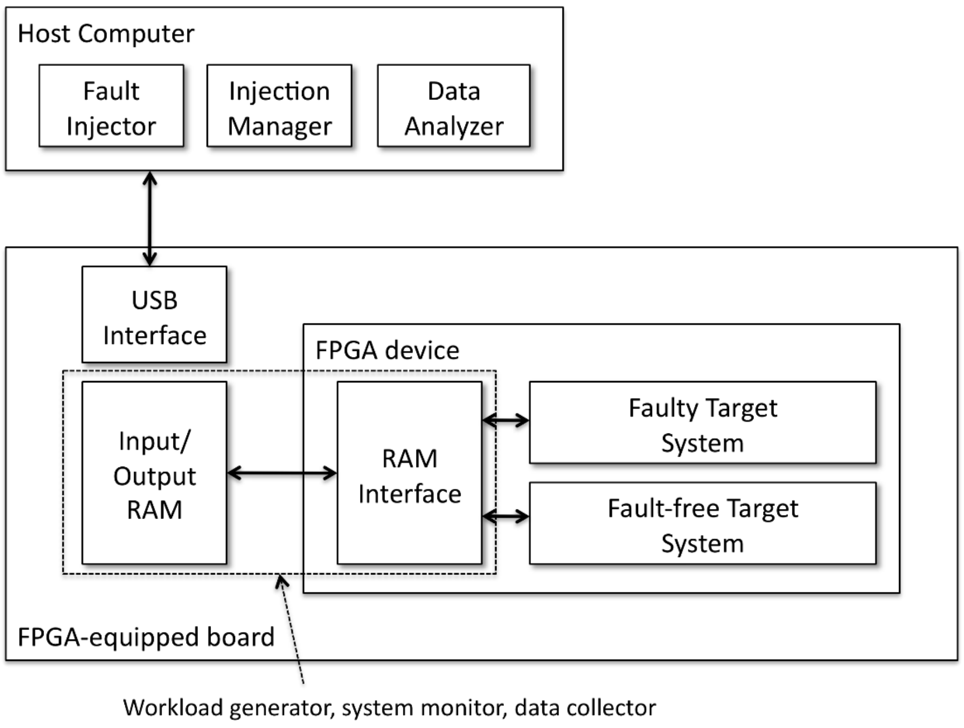

Fig. 4. Architecture of FT-UNSHADES

An ad-hoc FPGA-equipped board is exploited where an FPGA-device (Xilinx Virtex 8000 device) implements three functionalities:

1. A RAM interface circuit to access the Input/Output RAM chips located outside the FPGA. These chips store the workload to be used during injections and the responses of the injection experiment.
2. An instance of the target system that will be subject to bit-flip fault inoculation, the so-called faulty target system.
3. An instance of the target system that will not be affect by faults, the so-called fault-free target system.

The board also features Input/Output chips for workload and output data storage, and an USB chip to communicate with the host computer.

By moving workload generator close to the device used for target system emulation, the designers of FT-UNSAHDES minimized the volume of data that has to be exchanged with the host computer during testing. As a result, the injection speed no longer depends on the throughput of the interface between host computer and FPGA-equipped board, but only on the much faster on-board bus between the Input/Output RAM and the FPGA device.

Moreover, the adoption of two instances of the target systems simplifies the system monitor and data collection functions. The same input stimuli are processed by the faulty and fault-free target systems at the same time; therefore, system monitor can be implemented by comparing the output signals produced by the two instances looking for mismatches. Data collection is triggered only in case a mismatch is detected, and upon such an event the injection experiment is stopped. As a result, the amount of output data to be sent to the host computer for each experiment is minimized: it is either a status indicating no mismatches, or indicating that a mismatch occurred.

The host computer implements the fault injector, injection manager and data analyzer functions. Among them, the most interesting is the fault injector which perform the inoculation of single bit-flips in the faulty target system.

Fault inoculation is obtained by exploiting the partial reconfiguration feature of the Xilinx FPGA device used for emulating the target system. Using such a feature, the FT-UNSHADES software operates according to the following algorithms:

1. Activate workload generator until the injection time T_i is reached.
2. Read from the FPGA device the value *v* of the flip-flop *ff* to be affected by bit flip.
3. If $v = 0$, perform a partial reconfiguration of the FPGA to assert the set control signal of *ff* and proceed to 5.
4. If $v=1$, perform a partial reconfiguration of the FPGA to assert the reset control signal of *ff*.
5. Re-activate the workload generator until the workload is completed, or a mismatch indication is received.

Thanks to this approach, and exploiting the one-hot behavior of the flip-flip reset/set control signals, one partial configuration operation is needed for each fault in the fault list. The amount of data that have to be exchanged between the host computer and the FPGA-equipped board thus account for two configuration frames (few hundreds of 32-bit words). As a result, thanks to the speed of the USB interface, and the limited amount of data that have to be exchanged, injection speed is not bounded by the host computer/FPGA-equipped board interface.

Thanks to its custom design, FT-UNSHADES can reach notable fault injection speed. As an example, the injection of faults in a design with 796 flip-flops activated by a workload encompassing 200,000 clock cycles takes 0.13 seconds on the average for each fault.

3.4.2 Final remarks on emulation-based fault injection

Emulation-based fault injection is an effective solution to the problem of assessing the dependability of complex designs when very large fault lists and extensive sets of input stimuli have to be considered. To achieve such a goal, the following conditions have to be met:

1. A suitable emulation-based fault injection system must be available capable of hosting the system under test, and able to provide efficient communications mechanisms to achieve high throughputs. To optimize performance, ad-hoc hardware is likely to be needed.
2. For emulation purposes the model of the system under test should be suitable for being implemented using FPGA devices. This requirement imposes some limitations on style the designers have to use to code the model, and is likely to be enforced only when many of the design decisions have been taken. As a result, emulation-based fault injection is likely to be used only in the late phase of the design flow, when the detailed model of the system is available.

3.5 Software-based fault injection

Virtually any electronic systems today embed processor cores running application software, and the systems used in critical applications follow the same trend. During the early phases of the design flow, models of the processor cores can be used for running simulation- or emulation-based fault injection. Later in the design flow, when a system prototype is available, different approaches are in general required. Indeed, when the prototype is built using ad-hoc manufactured ASICs, and discrete processor chips, different techniques than those based on the features of simulator tools and FPGA devices are needed. In the following we will focus on processor chips only,

letting the reader to refer to other sources, like for example [24], for an overview of techniques that can be used for inoculated faults in ASICs.
The concept of software-based fault injection consists in enriching the application software the processor runs with a routine, the injection routine, whose purpose is to implement fault inoculation. Injection routine is never used during the normal operations of the system, as it does nothing useful for the system user. The injection routine is activated only when the fault under investigation has to be inoculated, and its execution is stopped as soon as the injection is performed.
In the past years several techniques have been proposed to put in practice software-implemented fault injection. All of them use similar injection routine that when activated can modify any of the user-accessible resources. As fault inoculation takes place by means of software, it can affect only those parts of the system that can be reached using the processor instruction set and that are hence visible to the programmer. As an example, in case we are interested in inoculating bit flips in a processor, software-implemented fault injection allows reaching all the registers of the instruction set architecture (e.g., general and special purpose registers, control, and status registers), as well as all the memory addresses the processor is able to reach. Conversely, those registers that are embedded in the processor but that are invisible to the programmer (e.g., the boundary registers in the processor pipeline) cannot be attacked by injection, as instructions are not available to alter their contents.
The techniques developed so far differ in the method used for triggering the injection routine. The following methods have been proposed:

1. In case the processor runs an operating system, the services the operating system provides are used for trigger the injection routine. As an example, FERRARI [25] exploits the `ptrace()` system call of the Unix/Linux operating system to break the execution of the application whose behavior has to be studied in presence of faults. Other approaches implement injection routine triggering by means of ad-hoc developed device drivers [26].
2. In case the processor does not run an operating system, or in case modifications to the application software are not possible, the activation of the injection routine can be demanded to an interrupt request. According to this technique, the injection routine is set to be the handler of an interrupt not used by the system, so that injection takes place when the corresponding interrupt is triggered. The Xception tool [27] exploits processor self-generated interrupt (i.e., software traps) to perform injection when one of the following events is detected: instruction fetch from a specific address, operand load/store at a specific address, a specific time is reached, a combination of the above events. In [28] the authors suggest to activate the injection routine though the processor self-generate debug trap; moreover, they suggest exploiting the Background Debug Mode (BDM) Motorola processors have to avoid performance reduction. Conversely, the FISB [29] uses a programmable board that monitors the processor bus and triggers the interrupt request to which the injection routine is attached. The board can be instructed to trigger injection when one of the following events is detected: a certain number of instructions fetch is reached, or a specific address is accessed.

3.5.1 An example of software-implemented fault injection

In this section we present the software-implemented fault injection system introduced in [29], which takes benefit of the custom-developed Fault Injection Support Board (FISB) to perform the injection of bit-flip in a processor-based system.

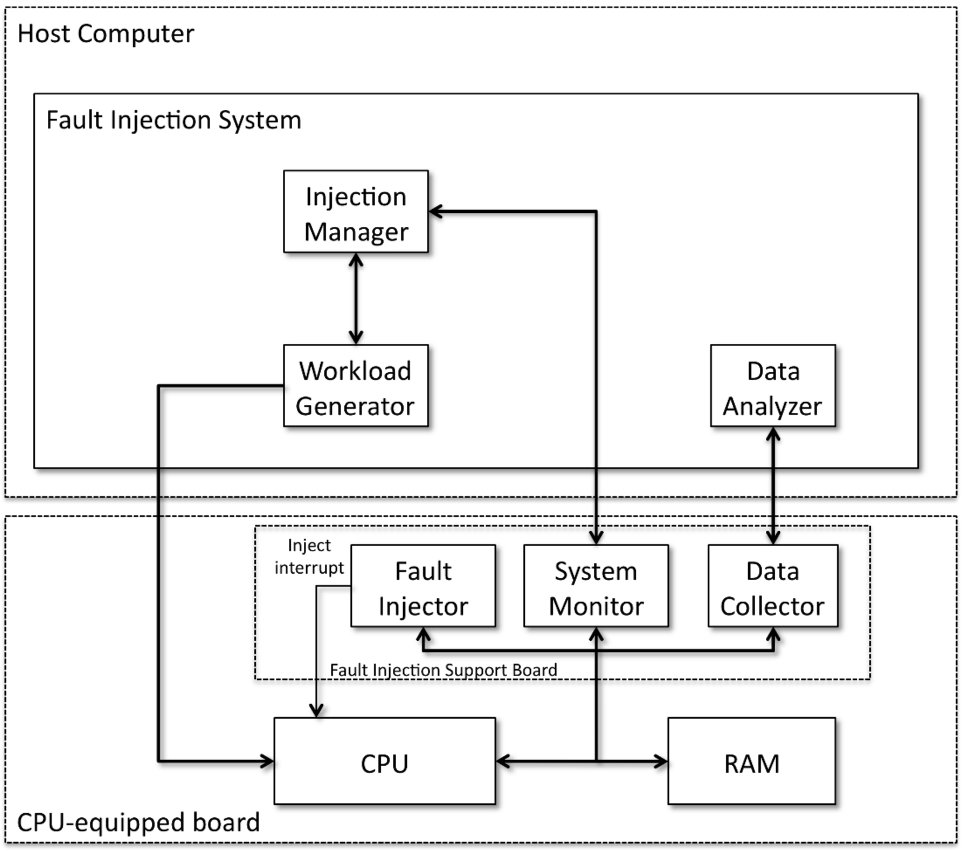

Fig. 5. Architecture of FISB

The purpose of the system is to allow fault injection in a processor-based system minimizing the time overhead. The CPU-equipped board which implements the system under investigation is enriched with an ad-hoc board, which is called Fault Injection Support Board, whose purpose is to implement System Monitor, Fault Injector and Data Collector functionalities, while the other components of the fault injection system are implemented in software and run by a host computer.

FISB is composed of an FPGA and 256 kwords each 7 byte wide. It is connected to the processor bus to monitor the processor behavior during application software execution, and it is seen by the processor as a memory mapped device. During its operation, FISB collects the following information:

1. The number of instructions the processor fetched from the memory until the application started.
2. The address and the value read/written from/to the memory by the application.

The Fault Injection manager can program FISB to trigger the inject interrupt as soon as a desired number of instructions has been executed, so that the associated injection routine is activated and fault injection can take place.

A typical fault injection run encompasses the following operations:

1. The fault injection manager programs FISB with the desired injection time T_i, and the desired fault injection mask *M*, which tells which register/memory address has to be corrupted during injection, and which bit of the register/memory address has to be altered by a bit-flip.
2. The fault injection manager activates the workload generator, so that the application software run by the CPU-equipped board is executed.
3. At injection time T_i, after the execution of the desired number of instruction, FISB issues the inject interrupt request, so that the injection routine is executed.

4. The injection routines accesses FISB to read the injection mask *M*, and inoculates the bit-flip accordingly.
5. The execution of the application software continues, and the system monitor module that FISB implements observes the behavior of the system under investigation, while the data collector gathers useful data to define the fault effect.
6. At application software completion, the collected data are analyzed and the fault effect classified.

The main advantage of using FISB lies in the possibility of running the application software at nominal speed, without any performance degradation. Indeed, FISB works in parallel with the system under investigation, and it interacts with it only when it is time for the injection routine to be activated (via the inject interrupt). As a result, real-time systems can be analyzed as the fault injection system has minimal impact of the system timing.

3.5.2 Final remarks on software-implemented fault injection

Software-implemented fault injection is an effective technique to inoculate faults in processors running application software, and several tools are available for supporting such a technique. However, the following limitations have to be remarked:

1. When compared to simulation- and emulation-based techniques, software-implemented fault injection results less versatile in terms of fault models and fault injection location. Being based on a software code running on a physical device, certain faults cannot be inoculated (e.g., delay faults). Moreover, only user-accessible resources can be target of injection, while other hidden resources cannot be attacked. The latter can become an issue when very complex processors are exploited, where many resources remain hidden. Indeed, if we consider modern high speed processors that can find their way in critical application demanding very high computing resources, we can see that they include a lot of resources for implementing advanced features as speculative execution, deep pipelines, etc., which are not accessible through the instruction set. Therefore, software-implemented fault injection can hardly be used for assessing the impact of faults in such resources.
2. Software-implemented fault injection requires the modification of the system software to insert the injection routine, which has to be triggered when needed. The size of the injection routine is normally negligible, and thus it does not introduce a significant memory occupation overhead. Conversely, the technique used to trigger injection may impact heavily on the system performance, in particular when special operational modes of the processor are used (e.g., the debug mode), or very computational-intensive features of operating system are used (e.g., `ptrace`). In case an interrupt line is used, it must be available.

3.6 Conclusions

The evaluation of the effects of faults possibly affecting an electronic system is a problem that can find solution in fault injection, which may have many different implementations. None of them can be considered as the ultimate solution to the dependability evaluation problem, as each of them has its own benefits and its own limitations. However, all of them can be put to work together so that designers can exploit positively their benefits. In an ideal design flow, fault injection should be used extensively since the initial design phases, from the system conception down to its prototypical implementation:

1. Simulation-based fault injection should be used to assists designers in debugging the error detection and correction mechanisms the system under investigation embeds. In this way bugs will be identified as soon as possible, and important parameters as system performance in presence of faults can be evaluated at a time in the design flow where modifications have a low impact on the time-to-market, as they entails low development costs.
2. Emulation-based fault injection should be used when the system model has been consolidated, to perform exhaustive validation of the error detection and correction mechanisms the system embeds. Such a validation is mandatory to avoid producing incorrect systems that will require very expensive design re-spins. As the number of faults is likely to be several orders of magnitude higher than those considered for design debug, emulation-based fault injection is likely to be the only possibility to keep the injection time acceptable. In case the system is processor based, software-implemented fault injection should be used as well, to complete the validation of the system under investigation before committing it to production.

Activities are currently underway to identify the solution which best suits the constraints and targets existing within the MaMMoTH-Up project.

4 Final Conclusions

This document summarized some solutions that could be adopted in the MaMMoTh-Up project to assess its dependability. The successful identification of the best mix of techniques out of those reported in the document is key to allow the analysis to be completed with a reasonable effort and in acceptable time, producing fair evaluation of the real dependability.

5 References

[1] UTE FIDES guide 2009, Edition A, September 2010

[2] RCA FMD-97

[3] Failure Modes, Effects and Criticality Analysis (FMECA). D. European SpaceAgency. 1991. ECSS–Q–30–02A.

[4] HORIZON 2020, MaMMoTH-Up COTS Functional and Environmental Specification, MMMTH-ASL-FSP-0001, Issue 2 of 30/07/2015

[5] N. M. Botros, HDL Programming Fundamentals: VHDL and Verilog, Charles River Media, 2005, 978-1584508557

[6] T. Grotker, S. Liao, G. Martin, S. Swan, System Design with SystemC, Springer, 2002, 978-1402070723

[7] A. Avizienis, J. C. Laprie, B. Randell, C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing", IEEE Trans. On Dependable and Secure Computing, Vol. 1, No. 1, 2004, pp. 11-33

[8] J. L. Leray, "Effects of atmospheric neutrons on devices, at sea level and in avionics embedded systems", Microelectronics Reliability, Vol. 47, 2007, pp. 1827-1835

[9] M. White, Y. Chen, "Scaled CMOS Technology Reliability Users Guide", JPL Publication 08-14 3/08, 2008

[10] M. C. Hsueh, T. K. Tsai, R. K. Iyer, "Fault Injection Techniques and Tools", IEEE Computer, April 1997, pp. 75-82

[11] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics", IEEE Transactions on Nuclear Science, Vol. 50, No. 3, 2003, pp. 583-602

[12] J. Bergeron, Writing Testbenches: Functional Verification of HDL Models, Springer, 2003, 978-1402074011

[13] TetraMAX, www.synopsys.com

[14] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, "Fault Injection into VHDL Models: the MEFISTO Tool", FTCS-24, 1994, pp. 66-75

[15] T.A. Delong, B.W. Johnson, J.A. Profeta III, “A Fault Injection Technique for VHDL Behavioral-Level Models”, IEEE Design & Test of Computers, Winter 1996, pp. 24-33

[16] D. Gil, R. Martinez, J. V. Busquets, J. C. Baraza, P. J. Gil, “Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System”, Dependable Computing EDCC-3, September 1999, pp. 191-208

[17] J. Boué, P. Pétillon, Y. Crouzet, “MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance”, FTCS´98, 1998

[18] B. Parrotta, M. Rebaudengo, M. Sonza Reorda, M. Violante, “New Techniques for Accelerating Fault Injection in VHDL descriptions”, IEEE International On-Line Test Workshop, 2000, pp. 61-66

[19] www.virtutech.com

[20] www.mentor.com

[21] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, “Exploiting Circuit Emulation for Fast Hardness Evaluation”, IEEE Transactions on Nuclear Science, Vol. 48, No. 6, 2001, pp. 2210-2216

[22] L. Antoni, R. Leveugle, B. Fehér, “Using run-time reconfiguration for fault injection in hardware prototypes”, IEEE Int.l Symp. on Defect and Fault Tolerance in VLSI Systems, 2000, pp. 405-413

[23] H. Guzman-Miranda, J.N. Tombs, M. A. Aguirre, “FT-UNSHADES-uP: A platform for the analysis and optimal hardening of embedded systems in radiation environments”, IEEE Int.l Symposium on Industrial Electronics, 2008, pp. 2276-2281

[24] A. Benso, P. Prinetto, Fault injection techniques and tools for embedded system reliability evaluation, Kluwer Acadmic Publisher, 2003

[25] G. A. Kanawati, N. A. Kanawati, J. A. Abraham, “FERRARI: A Flexible Software-Based Fault and Error Injection System”, IEEE Transactions on Computers, vol. 44, no. 2, pp. 248-260, Feb. 1995

[26] G. Cabodi, M. Murciano, M. Violante, “Boosting Software Fault Injection for Dependability Analysis of Real-Time Embedded Applications”, ACM Transactions on Embedded Computing Systems, 2009

[27] J. Carreira, H. Madeira, J. G. Silva, “Xception: Software Fault Injection and Monitoring in Processor Functional Units”, Fifth IEEE Working Conf. Dependable Computing for Critical Applications, pp. 135-149, 1995

[28] M. Rebaudengo, M. Sonza Reorda, “Evaluating the Fault Tolerance Capabilities of Embedded Systems via BDM”, IEEE VLSI Test Symposium, pp. 452-457, 1999

[29] A. Benso, P. L. Civera, M. Rebaudengo, M. Sonza Reorda, “A Low-Cost Programmable Board for Speeding-Up Fault Injection in Microprocessor-Based Systems”, Annual Reliability and Maintainability Symposium, pp. 171-177, 1999

[30] H. Hakobyan, P. Rech, M. Sonza Reorda, M. Violante, “Early Reliability Evaluation of a Biomedical System”, 9th IEEE International Design & Test Symposium, Algiers, Algeria, December 2014, pp. 45-50