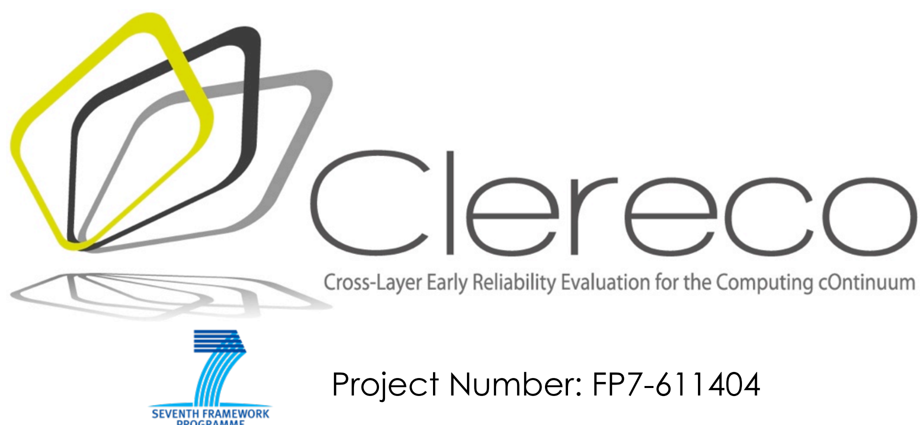

SEVENTH FRAMEWORK PROGRAMME

Project Number: FP7-611404

D5.4.1 - Early Validation Report (Preliminary)

Authors[1]

M. Pipponzi (YOGITECH), F. Sforza (YOGITECH), S. Di Carlo (POLITO), A. Savino (POLITO)

Version 1.0 – 27/04/2015

Lead contractor: Yogitech
Contact person: Mauro Pipponzi Yogitech Spa Via Lenin 132/P 56017 S.Martino Ulmiano (PI) - Italy E-mail: mauro.pipponzi@yogitech.com
Involved Partners[2]**:** YOGITECH, POLITO, UoA, CNRS, UPC
Work package: WP5
Affected tasks: T5.5

Nature of deliverable[3]	R	P	D	O
Dissemination level[4]	PU	PP	RE	CO

[1] *Authors listed here only identify persons that contributed to the writing of the document.*

[2] *List of partners that contributed to the activities described in this deliverable.*

[3] *R: Report, P: Prototype, D: Demonstrator, O: Other*

COPYRIGHT

© COPYRIGHT CLERECO Consortium consisting of:

- Politecnico di Torino (Italy) – Short name: POLITO
- National and Kapodistrian University of Athens (Greece) - Short name: UoA
- Centre National de la Recherche Scientifique - Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (France) - Short name: CNRS
- Intel Corporation Iberia S.A. (Spain) - Short name: INTEL
- Thales SA (France) - Short name: THALES
- Yogitech s.p.a. (Italy) - Short name: YOGITECH
- ABB (Norway) - Short name: ABB
- Universitat Politècnica de Catalunya: UPC

CONFIDENTIALITY NOTE

THIS DOCUMENT MAY NOT BE COPIED, REPRODUCED, OR MODIFIED IN WHOLE OR IN PART FOR ANY PURPOSE WITHOUT WRITTEN PERMISSION FROM THE CLERECO CONSORTIUM. IN ADDITION TO SUCH WRITTEN PERMISSION TO COPY, REPRODUCE, OR MODIFY THIS DOCUMENT IN WHOLE OR PART, AN ACKNOWLEDGMENT OF THE AUTHORS OF THE DOCUMENT AND ALL APPLICABLE PORTIONS OF THE COPYRIGHT NOTICE MUST BE CLEARLY REFERENCED

ALL RIGHTS RESERVED.

[4] ***PU**: public, **PP:** Restricted to other programme participants (including the commission services), **RE** Restricted to a group specified by the consortium (including the Commission services), **CO** Confidential, only for members of the consortium (Including the Commission services)*

INDEX

COPYRIGHT 2

Scope of the document 4

1. Introduction 5

2. Description of the problem 5

2.1. Complexity of the injection campaign 6

3. The Adopted Solution 6

4. Phases of the Validation Campaign 7

5. Set Up of the Validation Platform 8

5.1. Organization of the Platform 8

5.2. Modeling of the faults 8

5.3. Modeling of the HW 8

5.4. Modeling of the SW 9

6. Injection Campaign 9

6.1. Fault Simulation 9

6.2. About the complexity 11

6.3. Improvement of the fault simulation performances 12

6.3.1. Saving the vectors produced by the Golden Simulation 12

6.3.2. Excluding the faults in the non-functionally relevant portions of the design 12

6.3.3. Saving the snapshot of the simulation 12

7. Operational Profiler 13

7.1. Operational Profiler Concept 14

7.2. Operational Profiler Flow 15

7.3. Inactivity Windows 16

7.4. Computation of f and g 17

7.5. Current Status 19

8. Getting more information out of the simulation 20

8.1. Computation of the probabilities from the fault detection data 21

9. Bibliography 22

Scope of the document

This document is the main outcome of task T5.5 **"Preliminary Validation of Developed Models"**, elaborated in the Description of Work (DoW) of the CLERECO project under Work Package 5 (WP5).

Figure 1 depicts graphically the goal of this deliverable, its main results, the inputs it uses and the outputs it provides (including which WPs will use its outputs).

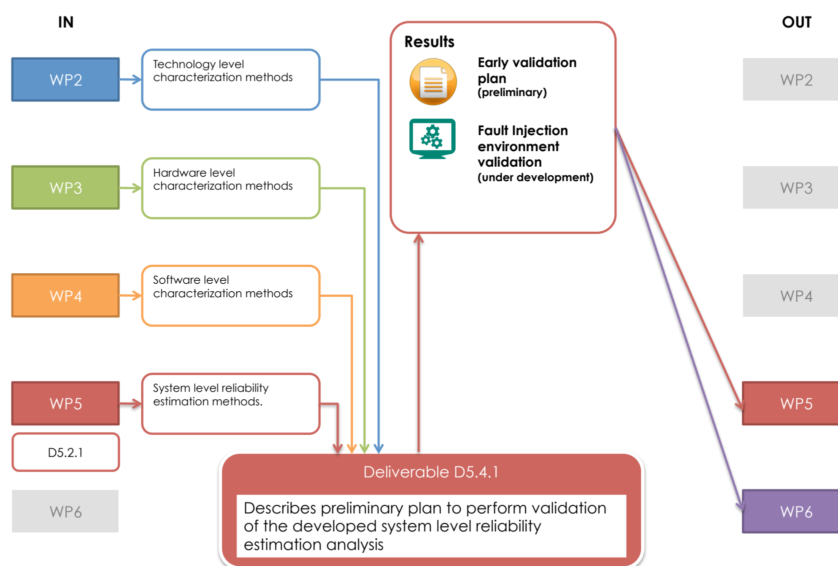

Figure 1: inputs and outputs of this deliverable

This document describes the preliminary validation environment envisioned for the validation of the models and algorithms developed in CLERECO for system level reliability evaluation. It takes its inputs from the results of several work packages and it is deeply connected with the activities and results of all WPs.

The document is organized in the following sections:

- **Description of the problem**. This section introduces the main challenges to face in the implementation of the validation plan
- **Set Up of the Validation Platform**. This section describes the fault injection environment used for validation.
- **Injection campaign**. This section overviews the main steps of the fault injection campaign
- **Operational profiler**. This section describes the operational profiler, one of the mechanisms developed in order to reduce the complexity of the validation task
- **Getting more information out of the simulation**. In this final section, we discuss how the data obtained from the validation campaign can also be used to validate intermediate results obtained with high-level characterization tools.

1. Introduction

Validating system level reliability estimations obtained resorting to the high-level models developed in CLERECO requires to compare obtained results with well established industrial practices to perform reliability evaluation.

This document details the features of the validation environment set up for the CLERECO project. It describes the adopted solutions, the fault injection environment and the operations carried out to validate the developed models, algorithms and metrics.

2. Description of the problem

The different models and algorithms composing the CLERECO system level reliability evaluation method need a low level validation step, that enables to guarantee that the metrics computed with an high-level analysis are confirmed when the actual hardware and software structure are at play.

Figure 2 shortly recalls the Bayesian Network model exploited in CLERECO to perform system level reliability analysis described in deliverable **D5.2.1 – System Reliability Estimation Models (preliminary)**. The CLERECO system level analysis models the system in terms of a Bayesian network whose nodes represent system technologies, hardware components and software blocks. Interactions among nodes represent capability of the system to propagate errors from one component to another component. The reliability evaluation method, estimates the rate of failure at the application level by propagating failure rates across nodes describing the system (Figure 2) using a set of conditional probabilities P_{fail} that model the probability that a failure at a certain level of the architecture hierarchy is propagated to the higher level. The final figure is the failure rate at the application level. The conditional probabilities P_{fail} characterizing the models are the results of a set of high-level fault injection campaigns carried out – with different tools - at the micro architectural and at the software levels.

It is the task of the validation to confirm the estimated failure rate at the application level by applying state-of-the-art injection of faults at a lower level (i.e., RTL level).

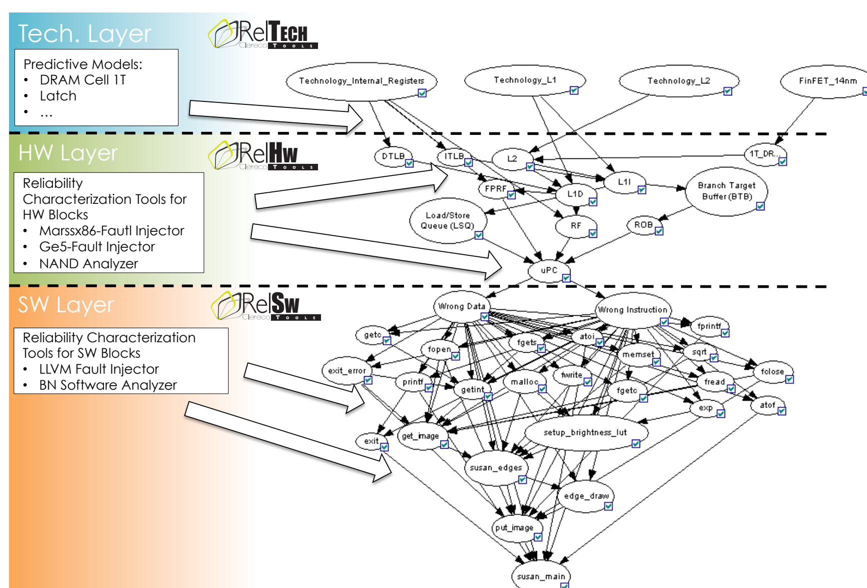

Figure 2: Reliability evaluation model.

2.1. Complexity of the injection campaign

The problem has a nontrivial level of complexity because of the performances of a low level simulation (RTL and below). A complex system can require injecting a number of faults in the order of the hundreds of millions, and the performances of the injection are highly dependent on the performances of the simulation, that can vary quite dramatically from one design to another. In our experience, an injection campaign can vary from tens of thousands of faults injected per hour to a few faults injected per hour (see also Section 6.2).

The parameters determining these numbers are many:

- The workload used for the fault simulation (which in turn reflects the application).
- The size of the design.
- The performances of the simulator.
- The number of simulators that can be run in parallel.
- The structure of the simulation environment.
- The possibility to adopt techniques that make it possible to speed-up the injection process (e.g., avoiding to run the full simulation for each fault).

In general, it is a safe assumption to claim that an injection campaign can require months of simulation even on specifically designed system.

It is therefore necessary, in order to be able to bring the problem within an acceptable number of simulation hours, to adopt an approach that, without sacrificing the accuracy of the analysis permits to reduce the simulation time as much as possible.

3. The Adopted Solution

During the project two main courses of actions have been analyzed and discussed with the partners to establish a validation protocol for the project results:

- The first option involves a divide-and-conquers approach: partitioning the systems into blocks, performing fault-injection based analysis of the different blocks and then combining them developing an ad-hoc methodology for the composition of the metrics. This approach is already in the roadmap of YOGITECH's mainstream development.
- The second approach is based on RTL level fault-injection on the full system using different techniques to reduce the fault injection time by dropping simulations for which the effect of the fault can be somehow predicted.

The discussion within the consortium has considered both approaches as interesting. However, the first approach requires developing methods to compose partial low-level fault injection results, mimicking at a lower level what the CLERECO reliability estimation model performs at system level. While this is still an interesting approach, it is not well consolidated at industrial level and may bias the robustness of the validation phase. Differently, the second approach represents a consolidated method. It must also be stressed the fact that the validation would be more significant with an approach that, even if at a different level of abstraction, does not repeat steps already used by the solution to be validated.

As a conclusion, we decided to adopt, at least for demonstrators in the embedded domain that will be based on ARM architecture and for the preliminary use cases, an RTL injection validation campaign.

4. Phases of the Validation Campaign

The validation campaign is carried out through a fault injection on the complete hardware in order to validate the failure rate at the application level as estimated by the high-level reliability estimation method.

Figure 3 shows the validation workflow.

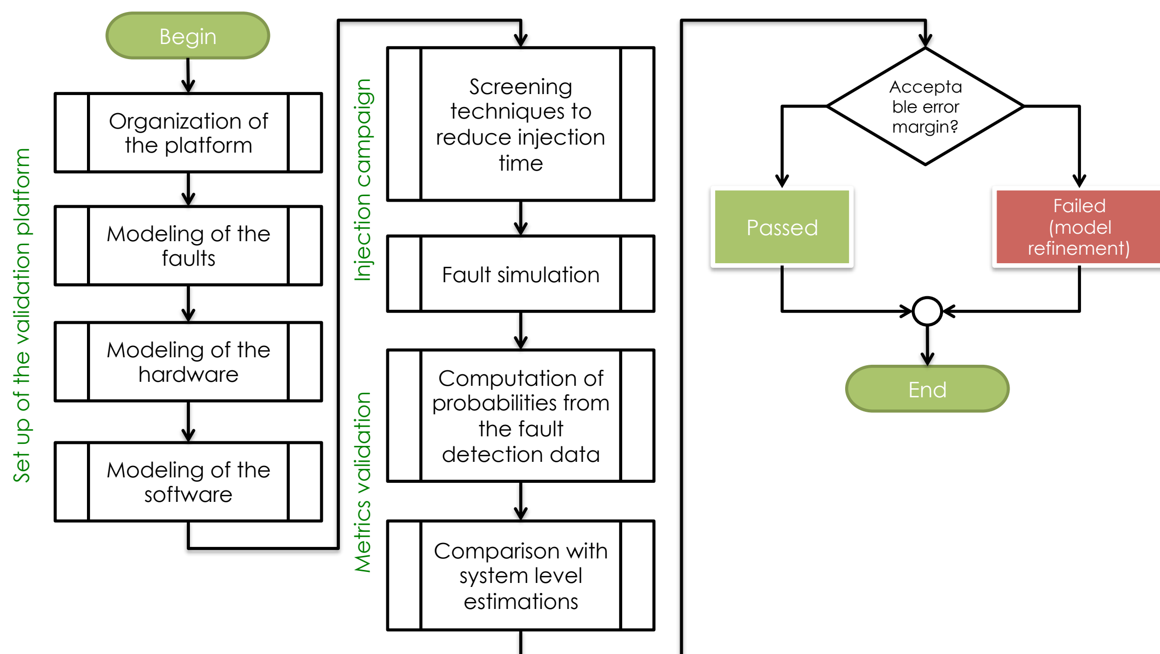

Figure 3: Validation workflow

The validation campaign can be partitioned into three main phases:

1. **Set up of the validation platform:** this phase requires studying the target system to validate in order to identify the models of the different components, and to define the target fault models.
2. **Injection Campaign:** in this phase the actual validation is performed by means of a low-level fault injection campaign. Due to the complexity of this task screening techniques to reduce the injection time must be implemented.
3. **Validation of the metrics:** in this final phase, application level failure rate are computed starting from the fault detection data obtained with the fault injection campaign. These numbers are compared with equivalent numbers obtained through CLERECO early reliability estimation models in order to assess the accuracy of the high-level models. In case accuracy is not enough, further refinements of the models will be implemented and a new validation phase will be required.

5. Set Up of the Validation Platform

5.1. Organization of the Platform

It has been decided by the consortium that the preliminary validation plan is based on RTL injection on ARM based microprocessor, at least for the embedded demonstrators and for the preliminary use cases. In particular, an ARM15 based platform has been chosen.

The choice of the platform and of the strategy, imply a number of requirements, such as:

- RTL database of the full HW layer.
- Transient fault model (bit flip of the output of memory elements / memory cells).
- Workloads (tests) at different level of abstraction modeling the low level functions of the software up to the target application.
- A test environment built in such a way to allow using the target simulator's features for saving and restoring simulation snapshots.

5.2. Modeling of the faults

The CLERECO system level reliability estimation methodologies enables to consider several types of fault model. In this preliminary validation plan we focus on analysis of the effect of transient fault that represent a significant class of faults for reliable systems. In particular we model faults as Single Event Upset (SEU), where the value of a bit is flipped (0->1 or 1->0) at a certain point of the simulation (see also Figure 5).

The fault is characterized by:

- **Location**: where the fault takes place in the mode of the HW.
- **Time**: when the fault takes place during the simulation.

Typically, transient faults are injected at the output of memory elements, therefore, in a RTL model they are injected on:

- Clocked signals.
- Output of memories.
- If the model of the memory makes the bit accessible, individual bits within a memory block.

5.3. Modeling of the HW

The HW is being described as a hierarchical RTL model of the platform as sketched in Figure 4. The hierarchical organization of the RTL model – even if not mandatory – should match as much as possible the high level decomposition of the hardware functions. This would allow additional benefits, such as speed up the fault injection campaigns (for further information, see Section 2).

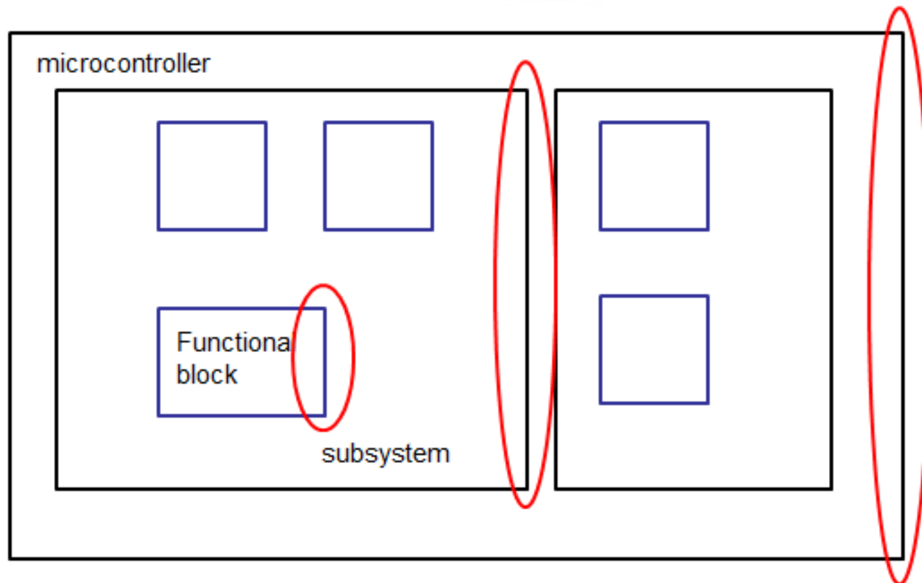

Figure 4: Hardware layers

The list of the subset of outputs of the functional blocks significant with respect to the detection of the fault (so called *observation points*) is part of the model of the functional blocks.

5.4. Modeling of the SW

The software functions, from the most basic to the full application, are modeled as workloads (test benches) for the hardware. These workloads can be either modeled with a HDL or can be modeled as vectors to be applied at the input of the HW.

The vectors to be applied to the hardware are captured by *strobing* the application running on a physical platform, saving them and then formatting them as suitable inputs for the simulation software.

As for HW, also in the case of the SW, part of the model of each function is the set of signals (registers, contents of memories, etc.) that are significant with respect to the detection of the injected faults.

6. Injection Campaign

6.1. Fault Simulation

The injection campaign injects transient faults (identified by the where/when pairs) into the HW, in order to determine whether these faults change or not the behavior of the device.

The faults are modeled as Single event upset (SEU), i.e., bit-flips on the output of memory elements (flip-flop, latches, memories). Figure 5 shows how bit-flips can be emulated at the simulation level (the example relies on HDL language formalisms).

```
flip (s)
input s;

begin
  if (s==1'b0)
    flip = 1'b1;
  elsif (s==1'b1)
    flip = 1'b0;
  else
    flip ? 1'bX;
  endif
end
endfunction
```

Figure 5: bit-flip model

The detection takes place at the output of the hierarchical blocks and functions, i.e., a fault is detected if it changes the behavior of the device at the selected sets of observation points (see Figure 6). In our case, the observation points are those registers, memory locations and outputs that are significant with respect to the data produced by the application.

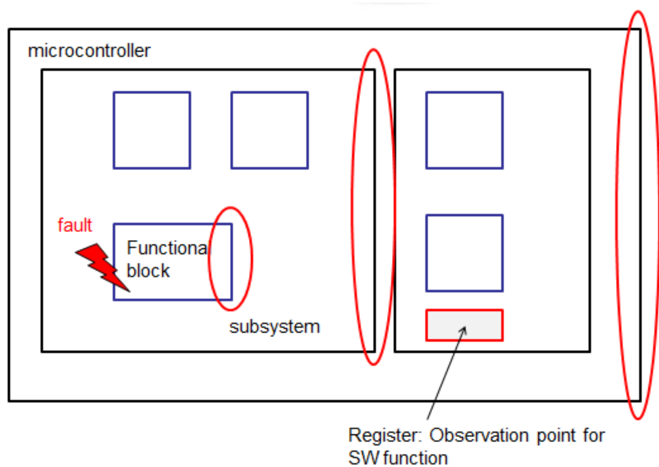

Figure 6: Observation Points

The actual injection platform (see Figure 7) is organized as a sequence of functional simulations, one for each fault injected, where a faulty machine is instantiated together as a golden machine and the two simulation results are compared with respect to the observation points.

Each fault is simulated and classified after the result of the simulation, whether detected or undetected.

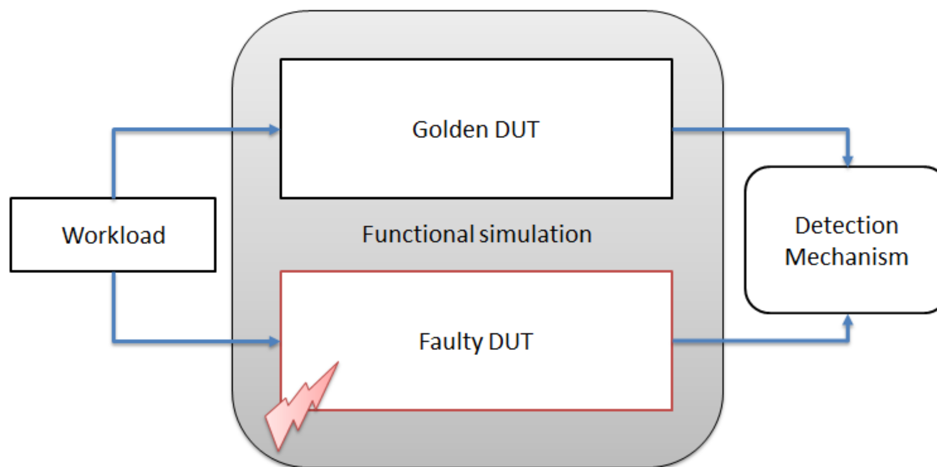

Figure 7: transient injection setup

6.2. About the complexity

As mentioned already, a fault injection campaign is a challenging task, because the number of faults to be injected in a complex system can rise rapidly to a number in the billions.

There are a number of factors affecting the length of a fault injection:

- The number of the faults injected.
- The length of the simulations.
- The frequency of the clock.
- The possibility to parallelize the simulations on many CPUs.
- The way the test bench is built.

Typically, in our experience, completing an injection campaign on an ARM based microcontroller requires 2-3 months of simulations, with up to 100 parallel simulations running in a computing farm.

We can report here a few quantitative figures, from actual projects YOGITECH has been working on.

Table 1: Quantitative examples of time required and parallelization level for different designs sizes.

	Size (Mgates)	Overall time to complete the campaign	Parallel CPUs
Example 1	5	2 months	10
Example 2	15	3 months	15
Example 3	30	3 months	100

6.3. Improvement of the fault simulation performances

The previous considerations make it very to be able to run an exhaustive injection campaign in an acceptable time problematic, not only from CLERECO project perspectives.

There are however a number of areas where it is possible to intervene to decrease dramatically the number of injected faults, and, therefore, the number and the length of the simulation. A set of approaches has been already considered to deal with it.

6.3.1. Saving the vectors produced by the Golden Simulation

The conceptual model of running the golden and faulty simulation for each fault can be improved by:

- Running the golden simulation once,
- Saving the observation signals as vector and,
- Using these vectors for the comparison with the simulation of the faulty machine.

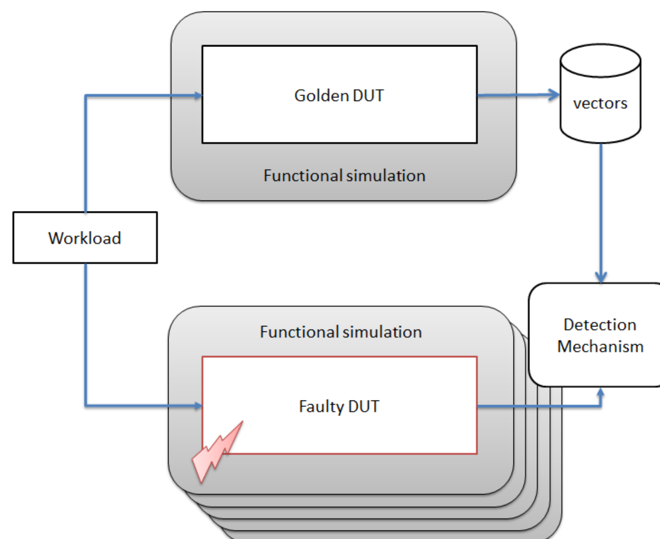

Figure 8: running the golden machine just once

The setup is shown in Figure 8 and allows decreasing the load on the computing machine by practically halving the number of simulations.

6.3.2. Excluding the faults in the non-functionally relevant portions of the design

A second area of improvement requires an architectural knowledge of the platform being simulated, and involves removing from the list of the faults to be injected all faults that are injected on portion of the hardware that is not functionally significant (e.g., test logic) or is not affected by the workload being simulated.

6.3.3. Saving the snapshot of the simulation

As mentioned in Section 5.1 one of the requirement for the validation platform is to build a test environment that allows using the target simulator's features for saving and restoring simulation snapshot.

This feature is pivotal because it allows huge improvements in the simulation time, since it makes it possible to run just a portion of the simulation in the interval near the time of injection, rather than the full injection for each fault.

When running the golden simulation, a number of snapshots will be saved at predetermined intervals. Once a fault will be injected at a certain time, instead of running the simulation from the beginning, the snapshot closest in time to the time of injection of the fault will be chosen (see Figure 9).

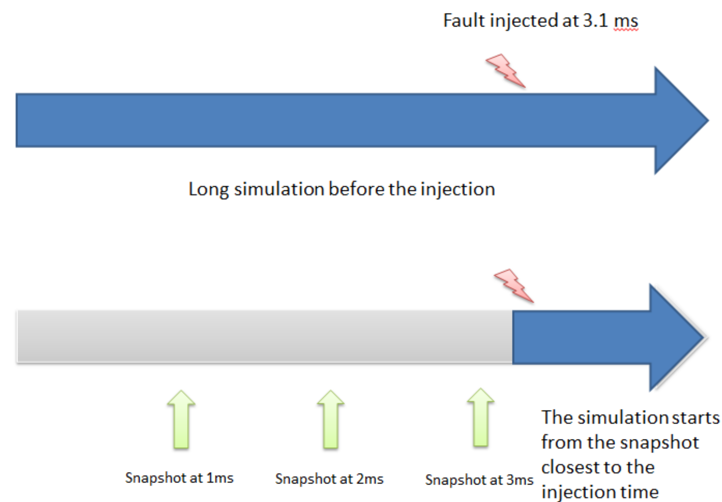

Figure 9: Use of snapshots

The most significant source of optimization, however, comes from the development and use of the Operational Profiler, as described in the next section.

7. Operational Profiler

The operational profiling is a solution to dramatically reduce the number of faults to be injected during a simulation and offers a big improvement in meeting the performance challenges of the low-level fault injection.

This technique is being refined, engineered and improved, with respect to the original concept [1] in order to specifically target the challenges of the CLERECO validation environment.

The basic goal is to determine upfront in which instants a fault at a certain location may produce any visible consequence at output pins or not. This information can be used to identify the **Windows of Opportunity** (WoO) in which it makes sense to inject a fault knowing that that the fault has a probability to propagate to the observation points (see Figure 10).

The simulation is pre-analyzed in order to identify the "inactivity windows", i.e., all those intervals where it is guaranteed that the fault is not propagated (e.g., a fault injected on a register is not propagated to its fan-out due to the selection of a different input of a multiplexer).

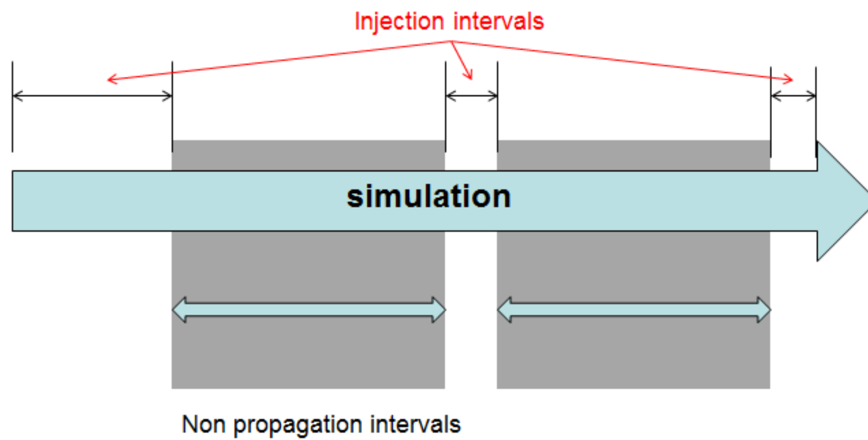

Figure 10: Identifying the WoO

The technique combines both static (analysis of the design database) and dynamic (simulation) analysis to identify the "inactivity windows". This analysis allows detecting the masking condition of the device without having to actually run a full simulation and allows a fairly good level of simplification.

This mechanism can be made hierarchical, starting from sub-blocks and propagating the faults to the HW top, enlarging the non-propagation intervals as the analysis moves up the stack.

7.1. Operational Profiler Concept

The idea behind the technique works on a concept called "parasitic simulation": given a particular test bench or set of stimuli driving an RTL design, the related fault tolerance is assessed using data generated by the RTL simulation – the vectors u and x – in order to check the observability conditions and hence screening unobservable faults and potentially observable faults.

The vectors u and v can be explained considering a logic module as a Mealy machine, expressed as a pair of logic equations as reported in Figure 11.

$$\mathrm{x}(\mathrm{k}+1)=\mathrm{f}\big(\mathrm{x}(\mathrm{k});\mathrm{u}(\mathrm{k})\big)$$

$$\mathbf{y}(k)=\mathbf{g}\big(\mathbf{x}(k);\mathbf{u}(k)\big)$$

Figure 11: logic equation - Mealy machine

Where:

- $\mathbf{x}(\mathrm{k})$ is the space-state array, i.e., the array of flip flops included in the module.
- $\mathbf{u}(\mathrm{k})$ is the array of input Boolean variables.
- $\mathbf{y}(\mathrm{k})$ is the array of output Boolean variables.

$\mathbf{f}$ and $\mathbf{g}$ are combinational functions.

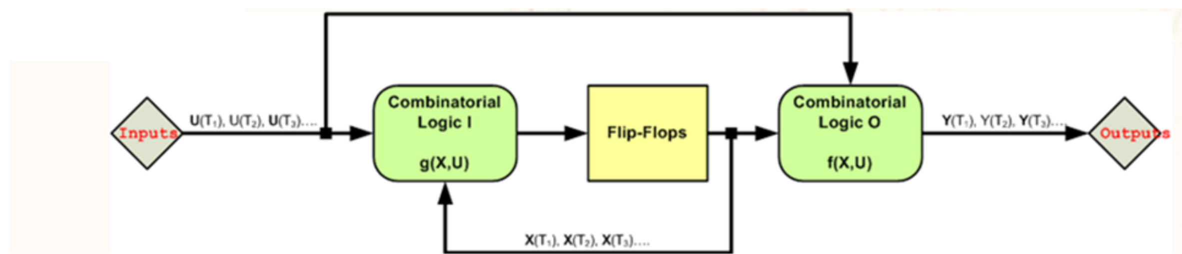

Figure 12: mealy machine

If we know **f**, **g**, **u**(k) and **x**(k) at a certain instant k of the simulation, a sufficient condition or the *non-observability* of a fault affecting a certain state at the k-th instant is:

$$f(0,x_2(k)\ldots x_N(k);u_1(k),u_2(k),\ldots,u_M(k))=f(1,x_2(k),\ldots x_N(k);u_1(k),u_2(k),\ldots,u_M(k))$$

Expression 1

while a necessary condition of the observability of the same fault at the same instant is:

$$g(0,x_2(k)\ldots x_N(k);u_1(k),u_2(k),\ldots,u_M(k))\neq g(1,x_2(k),\ldots x_N(k);u_1(k),u_2(k),\ldots,u_M(k))$$

Expression 2

These logic conditions are those used to separate observable and unobservable faults occurring at the instant k. It is enough to just solve combinational expressions, under the condition the **f** and **g** are known.

- **If Expression 1 is satisfied and Expression 2 is not satisfied, the fault does not propagates at instant k.**
- If Expression 2 is satisfied, the fault may or may not be observable depending on how it propagates outside the module.
- If Expression 1 is satisfied and Expression 2 is not satisfied, the fault does not propagates at instant k.

The faults satisfying the second conditions are called **potentially observable**.

The **"parasitic simulation"** will determine the vectors **u** and **v** at each instant of a RTL simulation, which will be used, together with **f** and **g**, to evaluate the logic expressions.

7.2. Operational Profiler Flow

- The Operational Profile analyzes the RTL code and calculates the functions **f** and **g** for each module in the model and creates a list of relevant signals that are part of **u** and **v**
- An RTL simulation is run – using a standard functional simulator. For each module in the RTL model, **u** and **x** are stored in vector files (named with a .vcd file extension).
  - A single simulation is needed.
  - Relevant data are stored at clock edges
- Using **x**, **u**, the observability conditions are checked using **f** and **g**.
- Unobservable faults are set apart.

- Potentially observable faults are propagated, using the $\mathbf{f}$ and $\mathbf{g}$ functions, until they reach any output port at the top level (or any observation point), until the discarded during the process.

The process turns out to be very fast:

- The observability conditions for each fault (i.e., a pair $\mathbf{x}, \mathbf{k}$) are checked “locally” within the module the injection point belongs to, by solving $\mathbf{f}$ and $\mathbf{g}$, and using the data stored after the RTL simulation.
- When a fault is potentially observable, it is checked for actual observability by propagating it only to neighbor modules, again using the simulation data, and solving $\mathbf{f}$ and $\mathbf{g}$.
- The check only takes place at clock edges.
- Simulation may be limited to a subset of all the clock instants.

7.3. Inactivity Windows

In a complex system, there are many components interconnected one another, but it is unlikely that all these components are active at the same time. In general, only a subset of these components is switching at a certain instant. Moreover, we may encounter periods when the state vector $\mathbf{x}(\mathrm{k})$ and the input vector $\mathbf{u}(\mathrm{k})$ do not change. Consequently, during these intervals no changes may occur at the outputs $\mathbf{y}(\mathrm{k})$ or the next state vector $\mathbf{x}(\mathrm{k}+1)$.

Finally, considering complex tasks executed by the system over a long period of time, involving the cooperation of many components, it is likely that for most components, each of them will be working for a small fraction of the whole system activity (see Figure 13).

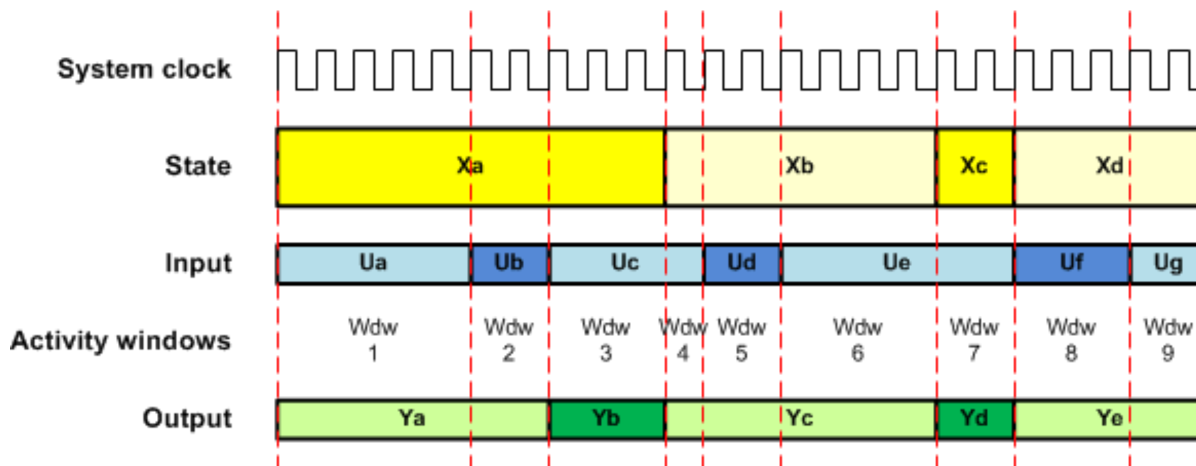

Figure 13: Inactivity Windows

Based on these considerations, we can introduce a great simplification: observability conditions need to be checked only when some changes occur, at $\mathbf{u}$ or at $\mathbf{x}$.

The boundaries of inactivity windows are a limited subset of the whole set of clock cycles. If the process limits its computations only to those windows, we have a significant simplification with respect to the standard RTL simulation.

It is also worth noticing that a .vcd file, given its structure intended to store differential information, automatically provides a simple way of building the inactivity windows of a certain module.

7.4. Computation of f and g

We introduce the concept of "leaf component" and "grey component".

A "leaf component" (Figure 14) is a module in the design hierarchy that includes:

- Input signals.
- Output signals.
- State elements (flip-flops).
- Combinational logic.
- No instances of sub modules.

A grey component (Figure 15) also includes instances of sub modules. In this case:

- The inputs of the sub modules are considered as output of the including module.
- Dually the output of the sub modules are considered as inputs of the including module.

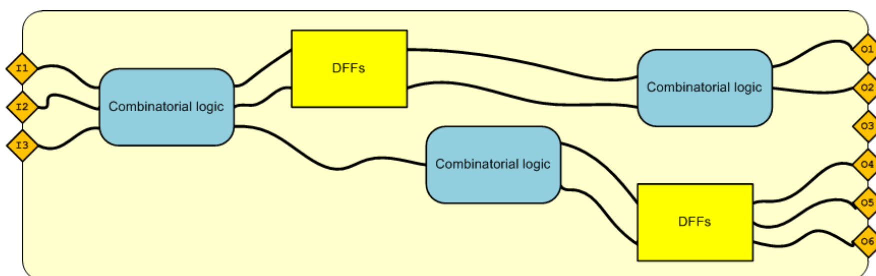

Figure 14: leaf component

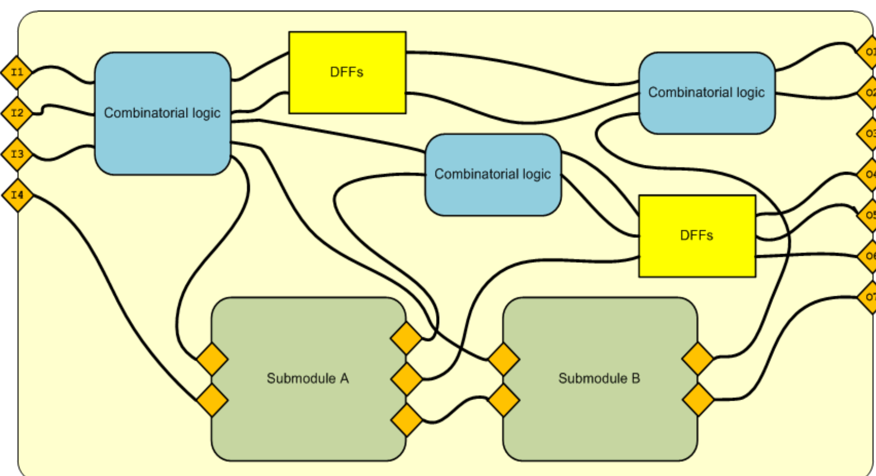

Figure 15: grey component

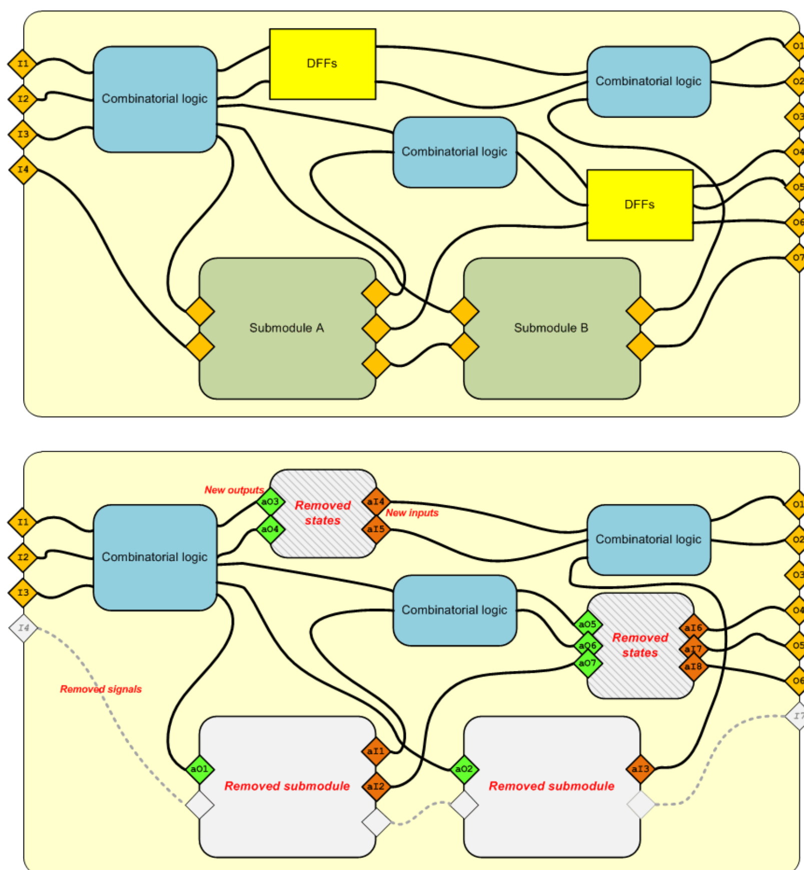

Figure 16: Substitution

Each module can be reduced to a piece of standalone combinational logic by removing internal sub modules and internal states (clocked process). In order to guarantee logic equivalence, the removed parts need to be replaced by the related signals at their boundaries (Figure 16 and Figure 17). During this process, some new I/O ports are added while others may be removed. I/O ports directly mapped to other I/O ports, without any combinational functions in the middle, may be removed.

The remaining logic is combinational logic and it is used to build the functions **f** and **g**:

- The union of the fan-in of all the registers is used to build **f** (state function).
- The union of the fan-in of all the output ports is used to build **g** (output function).

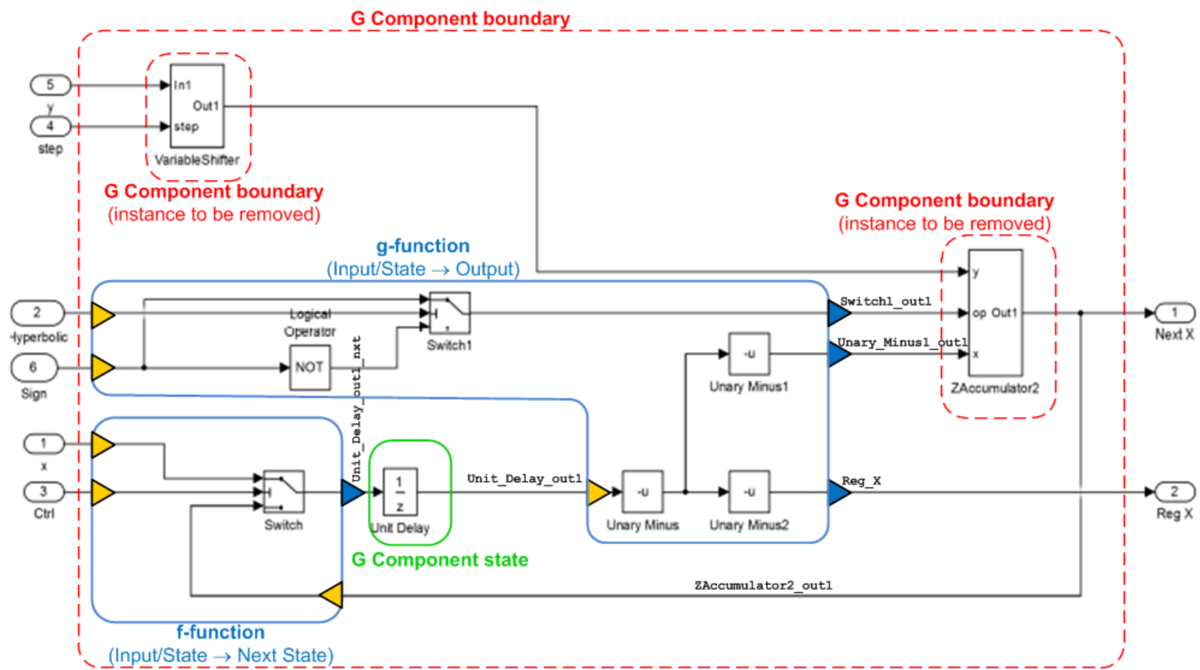

Figure 17: Actual Example

f and **g** are built with a manipulation of the original RTL code (Figure 18) by:

- Removing the sub modules,
- Adding the related I/O ports,
- Reducing all the clocked processes to the related combinational logic, and,
- Adding the related I/O ports to module definition.

Some attention must be paid to the transformation where asynchronous signals are present, because the logic is transformed into the equivalent of a synchronous reset.

At the end of the process **f** and **g** are simple and purely combinational chunks of RTL code that can be solved individually by a simple logic expression solver.

```
always @(posedge clk or posedge reset)
    begin : Unit_Delay_process
      if (reset == 1'b1) begin
        Unit_Delay_out1 <=
24'sb000000000000000000000000;
      end
      else begin
        if (enb) begin
          Unit_Delay_out1 <= Switch_out1;
        end
      end
    end
```

```
always @(*)
    begin : Unit_Delay_process
      if (reset == 1'b1) begin
        Unit_Delay_out1_nxt <=
24'sb000000000000000000000000;
      end
      else begin
        if (enb) begin
          Unit_Delay_out1_nxt <= Switch_out1;
        end
      end
    end
```

Figure 18: transformation of the clocked processes.

7.5. Current Status

An operational profile prototype has been built and it is currently working according to the description provided and it is being used for preliminary runs aimed at setting up the simulation environment.

Current issues to be addressed involve the amount of data to be stored from the RTL simulation of complex designs, that can be potentially huge, and the congruence of results with respect to the use with different functional simulators.

8. Getting more information out of the simulation

The purpose of the campaign is to validate the rate of failure estimated by the reliability estimation at the high level.

The procedure described so far, accomplished exactly that. However, with relatively little effort, it would be also possible to validate intermediate information concerning more detailed numbers – such as the conditional probabilities P - used at the lower levels of the HW-SW stack to compute the rate of failure.

This can be accomplished introducing the idea of hierarchical detection (Figure 19), where we set up multiple sets of observation points, at different levels of the hierarchy. In this way, with the same simulation, we can get information about the effect of the fault within the local piece of hardware it belongs to, at the subsystem level, as well as at the output of the hardware and with respect to the software functions.

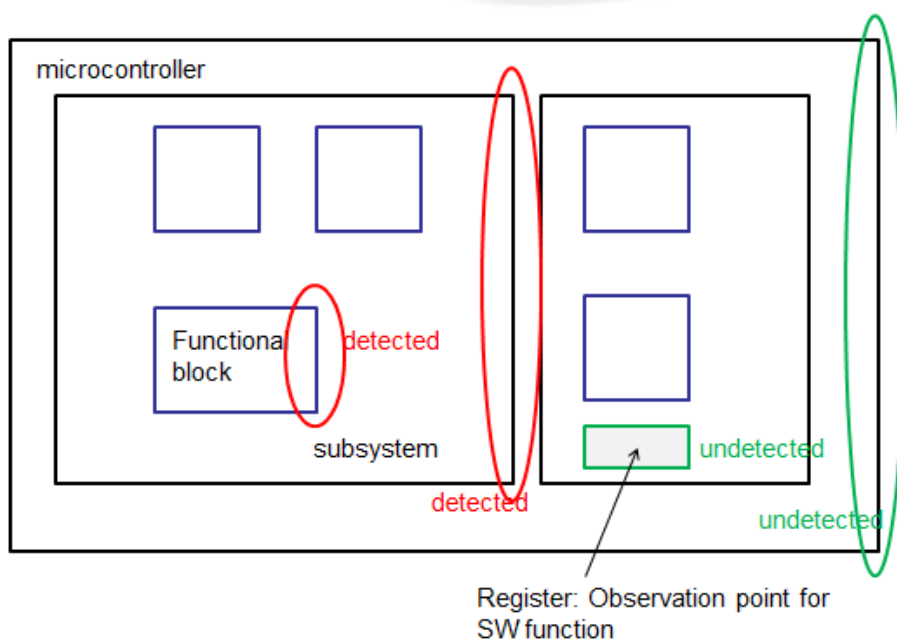

Figure 19: hierarchical detection

In our case, we are interested to evaluate how the faults propagate trough several layers of HW and SW, the detection will be considered separately with respect to the different observation points in the hierarchy of the point of injection.

In order to make this mechanism work, we need to introduce a variant to the classical mechanism of dropping detected faults. Where in the most common fault injection, a fault is dropped from the list of injected faults once detected, in our case detection at a lower case is not enough to drop a fault, because we need to give time to the fault to propagate to the higher levels of the stack. Only when a fault is detected at the higher level, it can be dropped, or, in alternative, after the expiration of a *timeout* in which no higher level detected it, the fault will be classifies as undetected for all the levels.

8.1. Computation of the probabilities from the fault detection data

The fault simulation classifies each fault with respect to each hierarchical block including the hardware element where the fault has been injected.

For instance a fault can be:

- Detected with respect to the observation points of the ALU of the microprocessor.
- Detected with respect to the observation points of the CPU.
- Undetected with respect to the microcontroller top.
- Undetected with respect to the SW layers.

While another one can be:

- Detected with respect to all the HW layers.
- Detected with respect to the low level OS Function.
- Undetected with respect to the Application main.

Once we have this classification available, we can evaluate which percentages of faults, among those injected, are detected at the lower functional levels, and which are propagated up the stack up to the application layer.

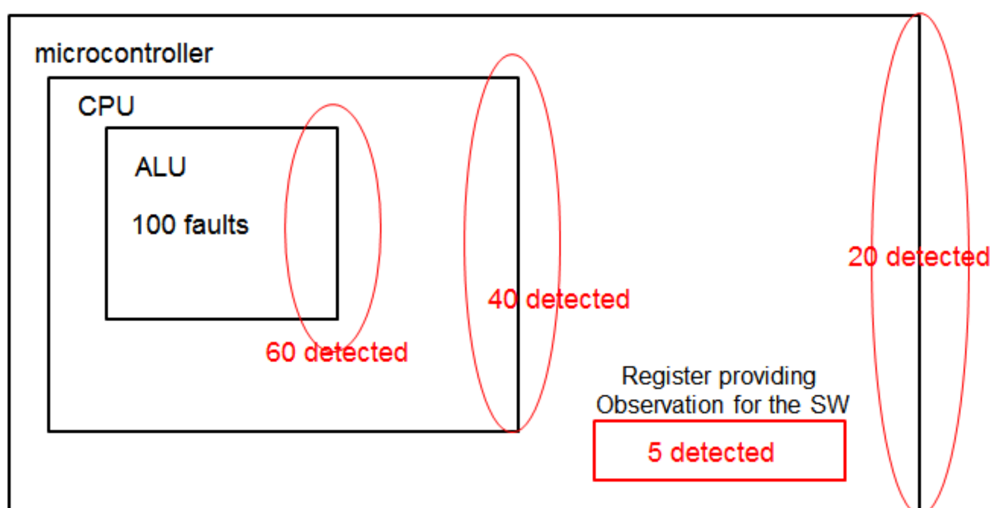

Figure 20: results of the injection at different levels of abstraction

In the example of Figure 20 we can see that, among the 100 faults injected in the ALU block, we have:

- 60 faults detected (60%)
- 40 faults propagate in the CPU block (40/60 = 66%)
- 20 faults propagate to the microcontroller level (20/40 = 50%)
- 5 faults propagate to the application (5/20 = 25%)

9. Bibliography

[1] Benso, A.; Bosio, A.; Di Carlo, S.; Mariani, R., "A Functional Verification based Fault Injection Environment," Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on , vol., no., pp.114,122, 26-28 Sept. 2007, doi: 10.1109/DFT.2007.31