Distributed Real-time Architecture for Mixed Criticality Systems

Fault injection framework D5.2.3

Project Acronym	DREAMS	**Grant Agreement Number**		FP7-ICT-2013.3.4-610640	
Document Version	1.0	**Date**		**Deliverable No.**	5.2.3
Contact Person	Mohammed Abuteir	**Organisation**		USIEGEN	
Phone	+49 271 740 2546	**E-Mail**		mohammed.abuteir@uni-siegen.de	

Contributors

Name	Partner
Mohammed Abuteir	USIEGEN
Zaher Owda	USIEGEN
Cristina Zubia	IKL
Félix Casado	IKL
Marcello COPPOLA	ST
Lukas Kohutka	TTT
Arjan Geven	TTT
Jorn Migge	RTaW
Manuel Muñoz	FENTISS

Table of Contents

Table of Figures

Executive Summary

This deliverable presents the fault injection and validation framework of DREAMS. The fault injection framework serves for verifying the DREAMS architecture using hardware fault injection and simulations. The fault injection comprises two dimensions, namely cluster level fault injection and chip level fault injection.

- ***Hardware fault injection at cluster level***: The hardware fault injector at the cluster level aims at testing the proper operation of a safety communication layer. The fault injector has been developed using a ZedBoard, which includes programmable logic (programmed using VHDL language) and an ARM processor (programmed using C language). Moreover, an expansion FMC board with two Ethernet ports to access the medium has been added. The whole system is controlled by a PC (with a C# application) that commands the fault injector through the ZedBoard's ARM via its Ethernet port.
- ***Fault injection in simulation at cluster level***: The fault injection at cluster level in the simulation was developed to simulate the system behaviour under the effects of design faults and operational faults. Thereby, the simulation framework supports the quantitative evaluation of the reliability of a DREAMS system with subsystems of different criticality.

  Moreover, the validation framework has been developed based on the low-level simulation of the communication between different DREAMS nodes to perform logic simulation. These low-level simulation models aim at finding potential errors in the implementation of protocols and standards in hardware.
- ***Hardware fault injection and simulation at chip level***: The hardware fault injection at the chip level was implemented to check failures according to the SoC under test specification, which gives the ability to increase the failure robustness for Functional Safety. Additionally, a novel definition of a framework that introduces the fault injection mechanisms in the current emulation flow is given in this deliverable.

The fault injection and validation framework is configured using the model-based configuration tools from WP4. Tool support is also provided for the visualization and analysis of the trace analysis results.

1 Introduction

This deliverable is dedicated to the implementation of the fault injection framework. The fault injection framework has been implemented at different levels, i.e. hardware fault injection at the cluster level and chip level as well as fault injection in simulations. To validate the reliability of the DREAMS architectures, different use-cases have been used.

In order to seamlessly integrate the virtual platform into the DREAMS tool chain, a model-driven configuration file generator has been implemented. Its goal is the automatic creation of simulator configuration files from a given system description in order to avoid error-prone manual editing. Furthermore, to ease the analysis of simulation results, high-level properties such as end-to-end delays have been synthesized out of simulation traces.

Furthermore, a formal validation framework has been implemented for mapping of simulation results to hardware to provide an end-to-end simulation environment that detects errors that cannot be found by simulating only the components.

1.1 Relationship to other DREAMS Deliverables

The hardware building blocks that have been implemented within WP2 and WP3 were the main input for the fault injection to validate and test different hardware building blocks such as STNoC and EtherCAT components. The virtual platform that was implemented in T5.2 served as the primary input for the fault injection in simulations. The fault injection framework will be used in different use-cases from WP6 and WP7.

The meta-model (T1.4 & T1.6) was the main input for the model-driven configuration of the simulation building blocks. This was developed in cooperation with WP4.

The validation framework is used to evaluate the functionality of the hardware components that are implemented within WP3 such as the gateway and the switch.

1.2 Positioning of the Deliverable in the Project

The goal of task T5.2 “Simulation, verification and fault-injection framework” is to provide a framework for simulating and verifying the behavior of a mixed-criticality system based on the DREAMS architecture. To achieve this goal, task T5.2 is divided into three deliverables: D5.2.1, D5.2.2 and D5.2.3.

D5.2.1 Specification of simulation framework

D5.2.2 Prototype implementation of simulation framework for DREAMS architecture

D5.2.3 Fault injection framework

This document is the final deliverable of task T5.2. The dissemination level of this deliverable is public (PU) i.e. once approved by the European Commission (EC), it will be freely available for download through the DREAMS project website (http://dreams-project.eu).

1.3 Structure of the deliverable

The remainder of this deliverable is structured in three parts. Part A introduces the implementation of the hardware fault injection. The implementation of the real-time fault injection framework at cluster level is

given in section 2, while the implementation details for the on chip fault injection framework are explained in section 3.

Part B covers the evaluation and validation for the DREAMS simulation environment. The co-simulation of the DREAMS chip with the execution environment is given in section 4. Section 5 describes the implementation of the fault injection in the simulation framework. The detailed specification of the virtual platform configuration file generator is provided in Section 6. The description focuses on the mapping between the DREAMS meta-model entities and the configuration file entities. Section 7 describes the functionalities implemented for the visualization of simulation traces and delay statistics derived from the traces for verification purposes.

Part C provides the description of the validation framework for DREAMS with a new methodology for evaluating the functionality of the hardware components.

Part A: Hardware Fault Injection

2 Real-Time Fault-Injection Framework

2.1 Introduction

When end-to-end communication takes place, several faults might occur during communication such as transmission errors, repetitions, deletion, delay, etc. An SCL (Safety Communication Layer) has been developed (in WP3) with the intent of considering the communication channel as a black channel and leaving all detection mechanisms in charge of this layer that will isolate the end application from the communication channel faults. The SCL will be stressed in the evaluation plan by means of the real time fault injection framework. The design and development of this Fault-Injector is the topic of this section.

2.2 Fault Injector Requirements

The main objective of this activity is to develop an FPGA-based system able to inject faults in a communication between two devices by placing it between them. Ideally, the system has to be able to perform an idle state (i.e. the injector forwards the frames received without introducing any perturbation) and start injecting faults when it is commanded to do so via software. At this point, it has to be raised that just one fault can be injected per iteration (an iteration is defined as a complete sequence of frames exchanged by the two devices). Thus, users have to introduce, via software, which type of fault is injected in a specific iteration. In addition, the Fault Injector has to be bidirectional, that is, being able to inject faults in both directions of the communication.

It will be used to test the Safety Communication Layer's (SCL) behavior, which implements the safety measures (techniques) defined by IEC 61784-3-3 to avoid the communication errors listed below and further detailed in deliverable D3.3.1 section 7.1.1.1:

- **Corruption:** Messages may be corrupted due to errors within communication channel participant, due to errors on the transmission medium, or due to message interference.
- **Unintended repetition:** Due to an error, fault or interference, old not updated messages are repeated at an incorrect point in time.
- **Incorrect sequence:** Due to an error, fault or interference, the predefined sequence (for example natural numbers, time references) associated with messages from a particular source is incorrect.
- **Loss:** Due to an error, fault or interference, a message is not received or not acknowledged.
- **Unacceptable delay:** Messages may be delayed beyond their permitted arrival time window, for example due to errors in the transmission medium, congested transmission lines, interference, or due to communication channel participants sending messages in such a manner that services are delayed or denied (for example FIFOs in switches, bridges, routers).
- **Insertion:** Due to a fault or interference, a message is inserted that relates to an unexpected or unknown source entity.
- **Masquerading:** Due to a fault or interference, a message is inserted that relates to an apparently valid source entity, so a non safety relevant message may be received by a safety relevant participant, which then treats it as safety relevant.
- **Addressing:** Due to a fault or interference, a safety relevant message is sent to the wrong safety relevant participant, which then treats the reception as correct.

2.3 Fault Injector Implementation

2.3.1 Hardware platform and overall solution

The system has been developed in a ZedBoard[1], which uses a Network Board (connected via FMC) to access the medium. In Figure 1, a sketch of the system is shown.

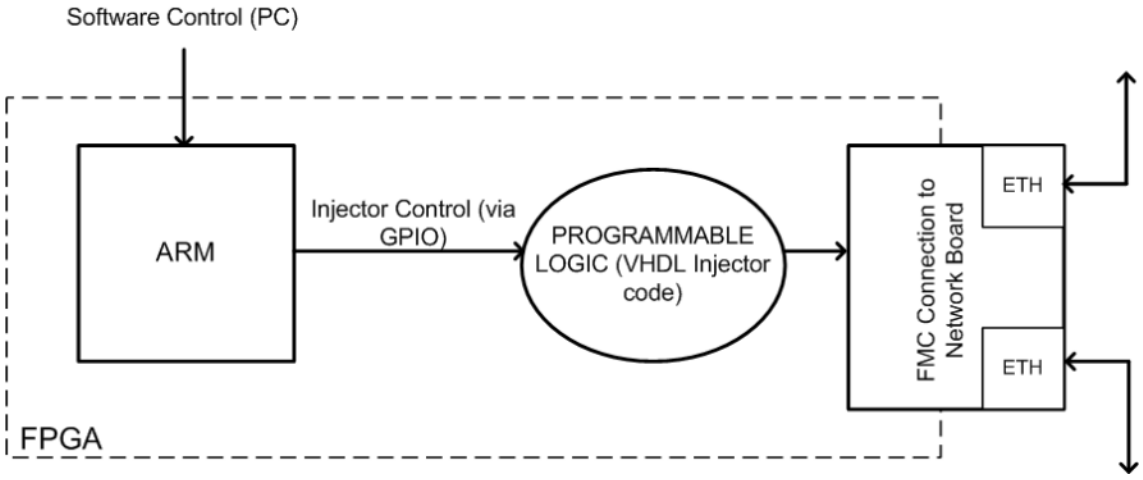

Figure 1 Sketch of the Fault Injector Deployment

The VHDL language has been used for the hardware programming (using Vivado software[2]), and the C language for the software programming and the integration with the hardware part (using SDK software[3]). Additionally, a Graphical User Interface (GUI) has been developed in C# (using Visual Studio Express[4]), in order to provide users with a graphical interface to control the Fault Injector.

In order to fulfill the requirements, three faults have been identified and implemented, which are:

- Byte Modification (BM): The Fault Injector changes a byte of the original frame. By changing a byte, communication errors such as data corruption and corruption of the sender/receiver addresses are emulated.
- Frame Delay (FD): The Fault Injector retains a frame and releases it after a certain delay. By delaying frames, it is possible to address the communication error of data packages outside temporal constraints.
- Frame Loss (FL): The Fault Injector does not forward one of the frames. With this fault, the communication error of a wrong sequence of packages is addressed, as every frame has been generated with its own sequence number.

[1] http://zedboard.org/

[2] http://www.xilinx.com/products/design-tools/vivado.html

[3] http://www.xilinx.com/tools/sdk.htm

[4] https://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx

With these three faults the Fault Injector is capable of reproducing most of the errors listed in section 2.2 as it is shown in the table below.

		Fault techniques					
		Implemented			Not implemented		
Error	**Can it be reproduced?**	**Byte Modification**	**Frame Delay**	**Frame Loss**	**Unintended repetition**	**Masquerading**	**Addressing**
Corruption	Yes	x					
Unintended repetition	No				x		
Incorrect sequence	Yes		x				
Loss	Yes			x			
Unacceptable delay	Yes		x				
Insertion	No	-			x	x	
Masquerading	No					x	
Addressing	No						x

- Corruption: This fault can be recreated using the fault “Byte Modification” in which the user is allowed to change a packet’s byte.
- Unintended repetition: Not implemented.
- Incorrect sequence: It can be reproduced using the “Frame Delay” fault, retaining the message and after a certain time (enough for the subsequent frame to be sent), releasing it.
- Loss: It can be emulated by the “Frame Loss” fault.
- Unacceptable delay: It can be recreated by using the “Frame Delay” fault.
- Insertion: It can be reproduced by changing a byte of the source direction using the “Byte Modification” fault, but the CRC will be wrong and the destination will discard the packet.
- Masquerading: By using the “Byte Modification” fault, a byte of the source direction can be modified. However, with this modification, the CRC computed at SCL level will be automatically wrong and the SCL layer at the destination will discard this packet.
- Addressing: As in the previous case, the destination direction can be changed using “Byte Modification”. But, again, the SCL at destination will discard the packet due to a CRC error.

The currently implemented Fault Injector would need some modifications in order to fulfil all the requirements. The pending techniques, which will not be addressed in the current version of the Fault Injector, could be developed in the following way:

- Unintended repetition: when the ‘Unintended repetition’ fault is chosen, as in the case of the ‘Frame Delay’ fault, the received packed will be stored in memory and will be transmitted after a given amount of time. Unlike the ‘Frame Delay’ fault, the received packet has to be forwarded. The retransmission time has to be provided by the user.

- Masquerading: consider the scenario shown in Figure 1. When the 'Masquerading' fault is chosen, instead of forwarding the packet received from 'Device 1' to 'Device 2', the Fault Injector will generate a 'non-safety' packet with the 0x03 valid source direction.

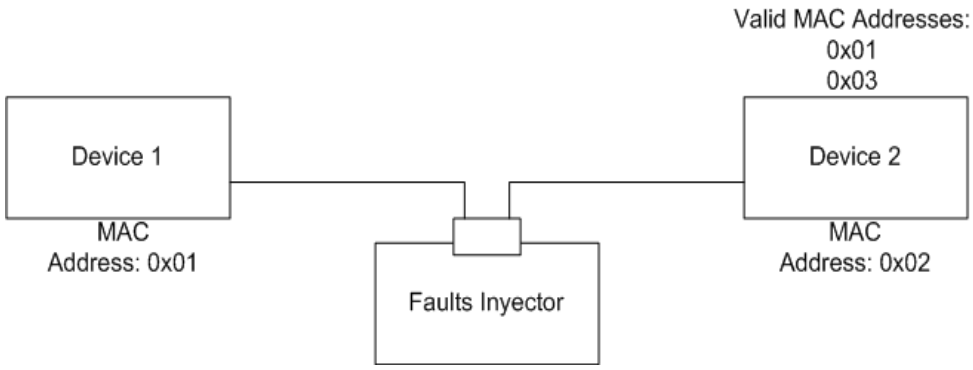

Figure 2 Sketch for masquerading fault

- Addressing: when the 'Addressing' fault is selected, the Fault Injector will change the original destination address of the received packet by another of an existing device and will also recalculate the CRC. For that purpose a switch will be required in the network as the Fault Injector has just two Ethernet ports.
- Insertion: The Insertion fault will be a combination of 'Unintended repetition' and the 'Masquerading' faults where the source address of the packet stored in memory is changed and the CRC is recalculated.

The implementation of these new techniques in future developments will depend on the needs encountered in the implementation of the SCL and the Fault Injector in the wind power demonstrator, which are not clear yet.

Regarding the control of the Fault Injector, the following signals have been defined:

- Inyectar_falta: this signal is a flag that indicates when a fault has to be injected.
- Tipo_falta: used to select the type of the fault: BM, FD or FL.
- Offset: used when BM is selected. Indicates the offset of the byte to be changed.
- Mask_Mod_byte: used when BM is selected. The byte modified by BM is the result of the logical xor operation between the original byte and the Mask_Mod_byte.
- Delay: used when FD is selected. Specify the amount of time that the frame is retained.

Essentially, the implementation of the Fault Injector has consisted in three main parts, which are: idle operation, fault injection and software control.

2.3.2 Idle operation

As it has been already stated, when no fault is required to be injected, the Fault Injector behavior must be transparent for the communication, that is, the frames received must be automatically forwarded. Thus, the first step in the design has been the access of the medium, which allows the system to receive and transmit frames. Therefore, Medium Access Control (MAC) was required and to that purpose, the

"Tri-Mode Ethernet MAC v8.0" (TMAC)[5] Intellectual Property (IP) core has been instantiated. By properly configuring the TMAC, both reception (to the FPGA) and transmission (from the FPGA) are available; thus, four (two for each direction) simple machine states, one for reception and the other for transmission, have been implemented in order to forward the received frames. Those machine states consist of, basically, the required commands to manage the access to a shared memory. In order to avoid any loss as a result of an arrival of frames faster than expected, a circular memory (rather than a fixed one) has been used. This circular memory has been configured to be able to store two frames, which allows the system to store a second frame while the first one is being transmitted.

In Figure 3, it is shown a capture of ChipScope where it can be observed that the data received by the TMAC ('rx_axis_mac_tdata[7:0]') is stored in the circular memory ('dina[7:0]') and, once the frame has been stored ('packet_stored') it triggers the transmission state machine ('flag_to_init_tx'), so the frame is forwarded from the memory ('doutb[7:0]') to the TMAC ('tx_axis_mac_tdata_eth2[7:0]'). For this specific scenario, the frames are made up of 61 bytes; thus, with such conditions, the delay introduced by the Fault Injector (i.e. from the time it receives the first byte till it releases the first byte) is approximately 7 µs. It has to be noted that the delay is proportional to the frame length. Thus, each additional byte adds a delay of 80 ns.

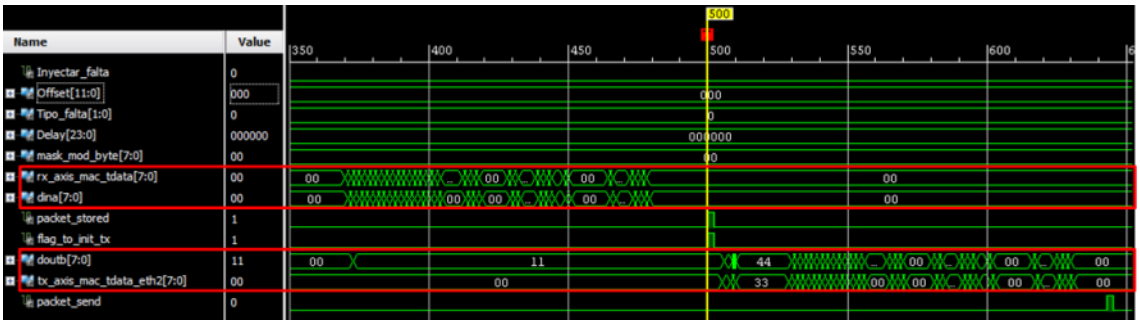

Figure 3 ChipScope capture of Idle Operation

2.3.3 Faults injection

Once the basic part (the idle operation) was built up, the next step was to add the required actions to perform the injection of faults. Indeed, each kind of fault requires specific actions that will shape the Fault Injector.

- Byte Modification (BM): When BM is the selected fault, the frame reception works just as in the case of idle operation; so, the major changes have been included in the transmission. Thus, when the byte to be transmitted is in the position indicated by the Offset control signal, instead of transmitting the original byte, the result of the logical xor operation between the original byte and the Mask_Mod_byte is sent. In the capture of ChipScope in Figure 4 it can be observed that when 'Inyectar_falta = 1', 'Tipo_falta = 1' and 'counter_tx[11:0] = Offset[11:0]', what is forwarded to the TMAC is not what is stored in memory (x"21") but the result of what is in memory xor 'mask_mod_byte[7:0]' => x"21" xor x"DD" = x"FC".

[5] http://www.xilinx.com/support/documentation/ip_documentation/tri_mode_ethernet_mac/v8_2/pg051-tri-mode-eth-mac.pdf

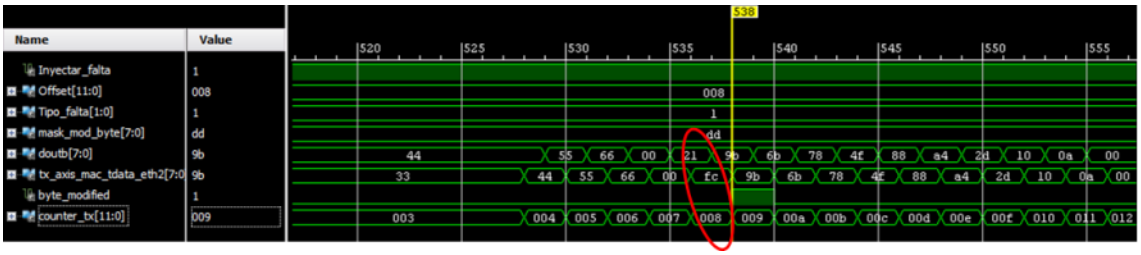

Figure 4 ChipScope capture of Byte Modification

- Frame Delay (FD): In case of selecting the FD fault, the received frame is not stored in the circular memory but in a parallel memory and, in addition, a timer is triggered. Once the timer reaches the value indicated by the Delay control signal, a flag is raised in order to transmit the delayed frame. Figure 5 shows two captures of ChipScope where it can be observed that when 'Inyectar_falta = 1' and 'Tipo_falta = 2' the data received by the TMAC is not stored in the regular memory but in another one specifically used for delayed frames ('dina_delay[7:0]'), and that a timer is triggered. When this timer equals the 'Delay[23:0]' signal, the transmission of the delayed frame is triggered ('tx_delayed_packet'1); thus, the frame forwarded to the TMAC comes from the memory used to store delayed frames ('doutb_delay[7:0]').

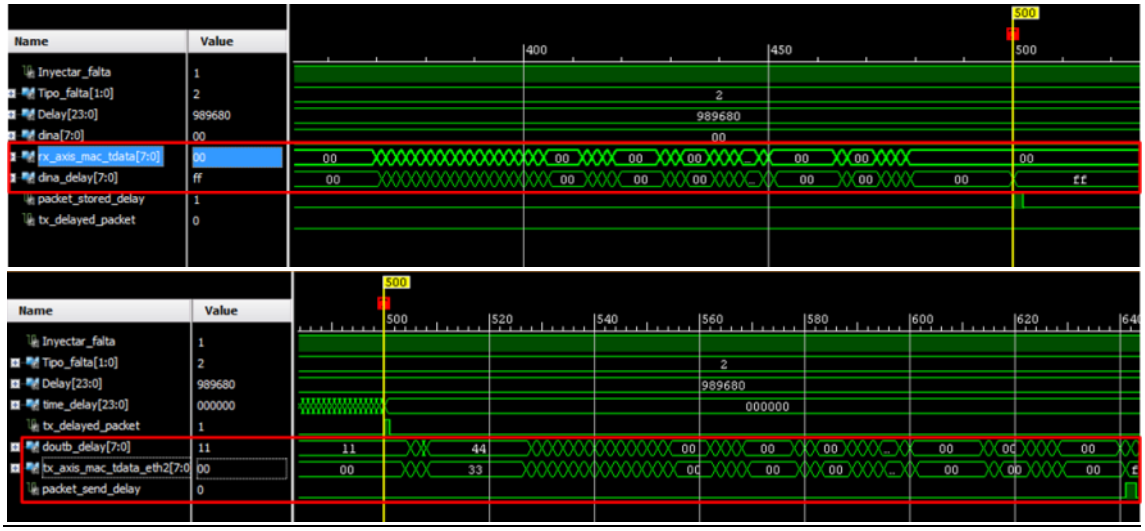

Figure 5 ChipScope capture of Frame Delay

- Frame Loss (FL): when the FL fault is selected, the reception works just like in the case of idle operation, but the flag to indicate that the frame is ready to be forwarded is never raised. Hence, the stored frame will be overwritten by new frames and will not be transmitted. Figure 6 shows a capture of ChipScope where it can be observed that when 'Inyectar_falta = 1' and 'Tipo_falta = 3', the data received by the TMAC is stored in memory, but the flag to start the transmission ('flag_to_init_tx') is never raised, hence, the frame is never transmitted.

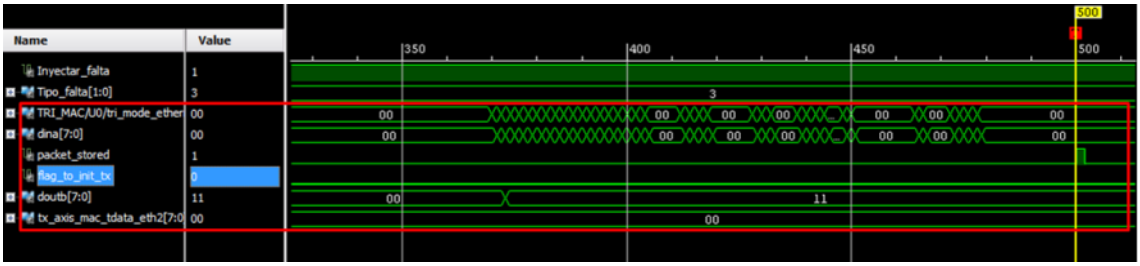

Figure 6 ChipScope capture of Frame Loss

2.3.4 Software Control

In order to allow users to steer the Fault Injector to their convenience, the control signals are controlled via software. For this purpose, ten "AXI GPIO" IP cores (one for each control signal and for both directions) have been instantiated. Thus, a simple application can be created using SDK in order to initialize and modify the control signals of the Fault Injector. Therefore, the software control part consists of two main parts: Graphical User Interface (GUI) and control data acquisition.

2.3.4.1 Graphical User Interface (GUI)

The main purpose of the GUI is to allow users to change the parameters that control the Fault Injector. It has been developed using C#, and its main functionalities are detailed below with a capture (Figure 7) of the implementation as a guide.

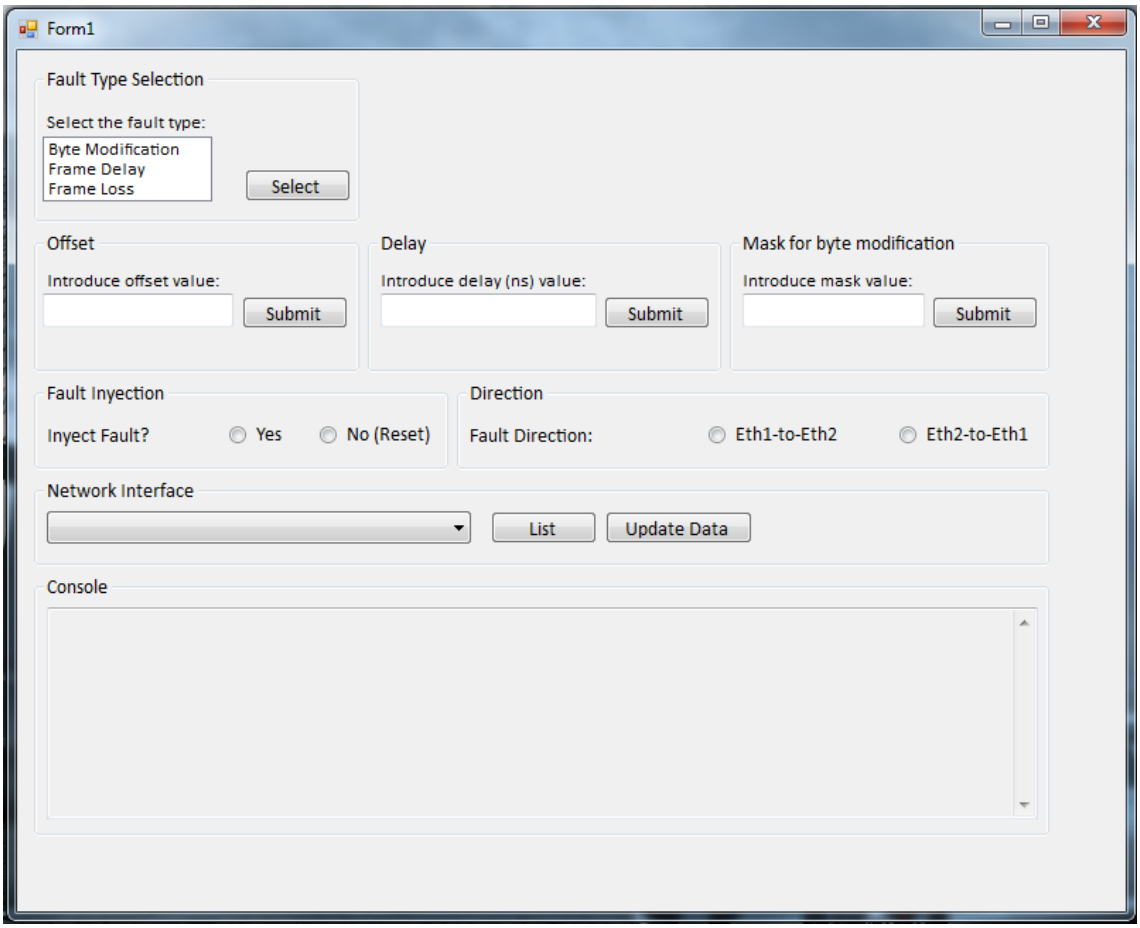

Figure 7 Graphical User Interface (GUI)

The parameters that have to be configured in the GUI are the following ones:

- *Fault Type Selection:* the three faults implemented (BM, FD and FL) are available for users to select one of them.
- *Offset:* Offset for the BM fault.
- *Delay:* the Delay (in nanoseconds) for the FD fault.
- *Mask for byte modification:* the Mask_Mod_byte for the BM fault.
- *Fault Injection:* at this point users select if the configured fault has to be injected or not. In order to reset the Fault Injector, after one fault has been injected users have to select No (Reset) and push the 'Update Data button'.
- *Direction:* selection of the communication direction where the fault is to be injected.
- *Network Interface:* by pushing the 'List' button, the available network interfaces are listed, so the most suitable one can be selected.
- *Console:* used by the system to show informative messages.
- *Update Data (button):* when the update data button is pressed the configured fault is sent, via socket, to the control data acquisition program.

2.3.4.2 Control data acquisition

The Control data acquisition is a C++ based program that has been developed using the SDK platform. It is in charge of receiving the configured fault from the GUI and updating the Fault Injector control signals accordingly. Essentially, this program initializes the communication interfaces (from the GUI and to the PL), and remains permanently waiting for any change in the fault configuration. On one hand, the communication between the GUI and the C++ based program is managed through the ZedBoard Ethernet Port. On the other hand, the control signals in the PL are updated through the use of the "AXI GPIO" bus.

To summarize, and gathering all the pieces, the Fault Injector operates as follows: the user configures the control parameters through the GUI (which is running in a PC); and therefore the GUI forwards the data to the control program running in the ARM, which updates the PL transferring the control parameters through the GPIOs.

2.4 Evaluation

2.4.1 Set-up

Different tools have been used in order to evaluate the behavior of the Faults Injector. On one hand, Ostinato[6] has been used to generate the traffic that passes through the Fault Injector. On the other hand, Wireshark[7] has been used at the end point in order to gather the traffic that goes through the Fault Injector. In Figure 8 the setup used for the validation is shown. As it can be observed, the setup is made of a PC generating packets with Ostinato, a second PC where users interact with the GUI and a third PC running the Wireshark tool to analyze the packets forwarded by the Fault Injector.

[6] http://ostinato.org/

[7] https://www.wireshark.org/

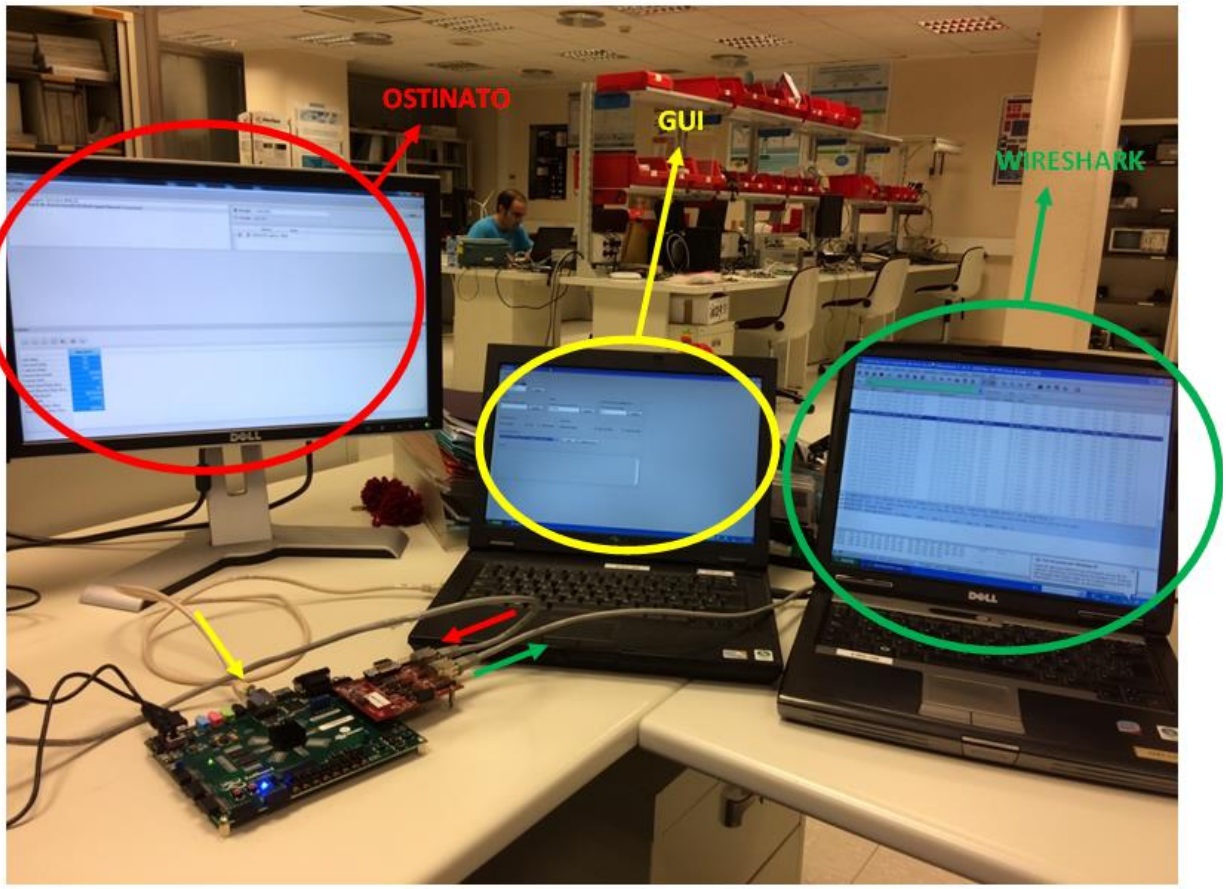

Figure 8 Validation setup

2.4.2 Test results

The three types of faults (BM, FD and FL) have been injected so the Fault Injector behavior is checked under all the conditions.

2.4.2.1 BM fault injection

In Figure 9 a BM fault is injected. The Ostinato tool generates 10000 ECAT packets that pass through the Fault Injector and, afterwards, they are gathered in a Wireshark capture. The Fault Injector has been configured with an Offset of 8 and a Mask_Mod_byte of 221. It can be observed that the 10000 packets reach the end point and that one of them has a different source direction (Figure 9) as a result of the BM fault.

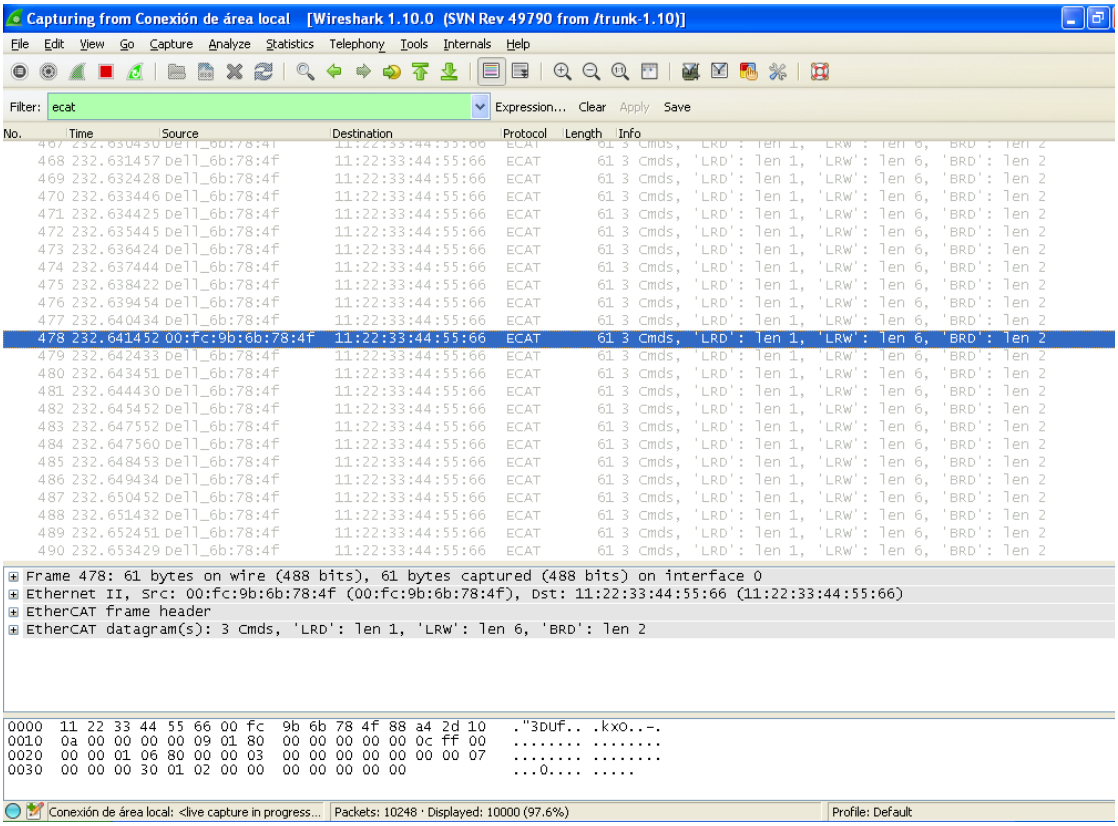

Figure 9 Byte Modification (BM) fault

2.4.2.2 FD fault injection

For this fault, the Ostinato tool is configured to deliver 40 ECAT packets with a rate of 100 ms and the Fault Injector is commanded to introduce a 9.5 ms delay. By taking a look at the time tag of the packets in Figure 10, it is clearly observed that the one marked in blue breaks the pattern by approximately 10ms, which validates the delay introduced by the Fault Injector.

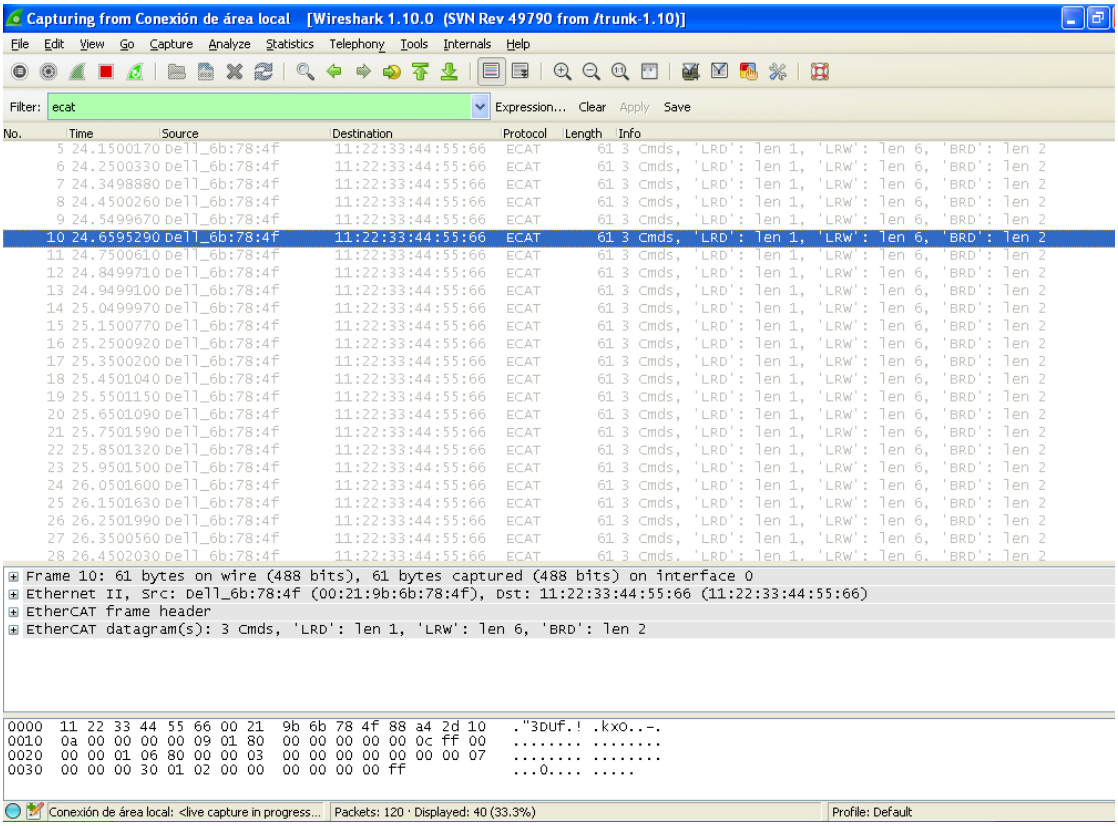

Figure 10 Frame Delay (FD) fault

2.4.2.3 FL fault injection

In this case, Ostinato is configured to deliver 10000 ECAT packets as in the case of the BM fault. In Figure 11, it is shown that just 9999 packets reach the end point, which validates the injected fault.

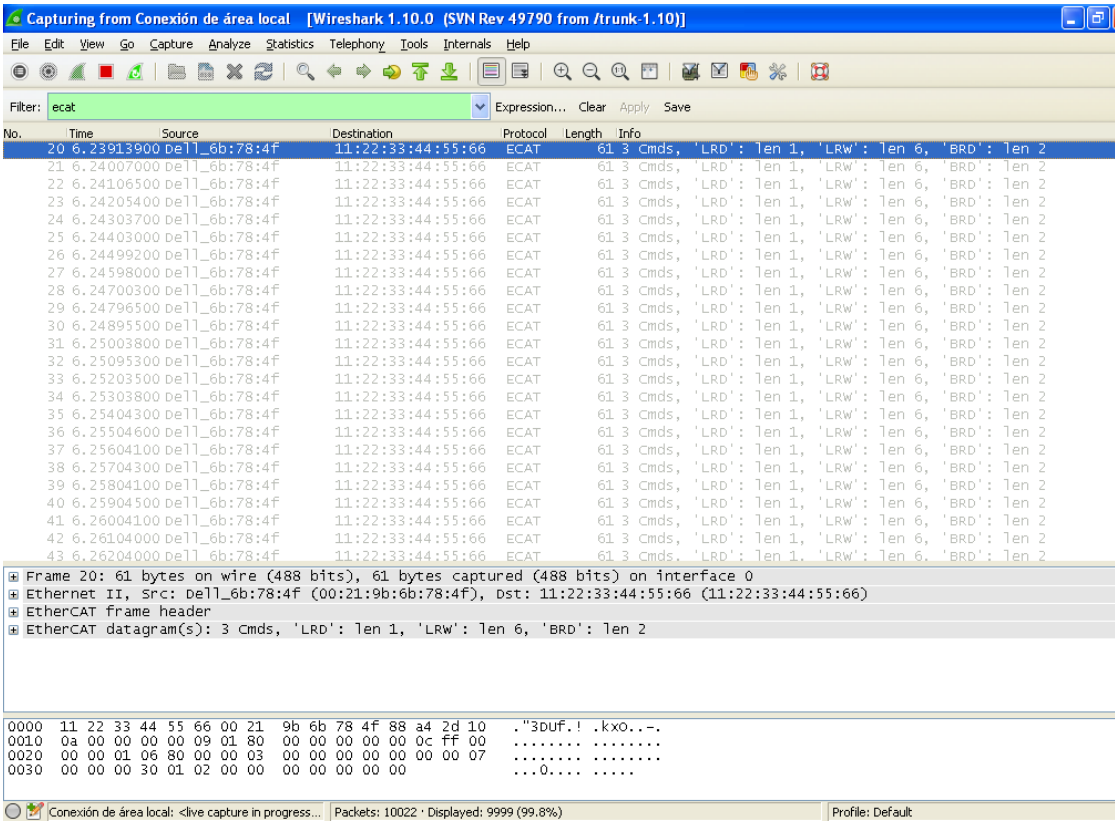

Figure 11 Frame Loss (FL) fault

2.4.3 Real-Time Fault-Injection Framework in the wind power demonstrator

The wind power demonstrator is a powerful tool to evaluate DREAMS technologies and the fulfilment of the project objectives. The wind turbine case study integrates a subset of the DREAMS technologies and among them, the SCL, which is used to transport safety related input/output data between EtherCAT slaves and the safety protection system deployed in the harmonized platform. The demonstrator is comprised of mixed-criticality partitions/applications, where the safety protection system is the most critical piece of software. The verification plan will include tests to validate that faults occurring in other partitions/applications do not compromise the safety level of the protection system (but possibly the availability). The Ethernet fault injector will be used as a tool to validate the impact of different faults in the off-chip communication bus.

Figure 12 shows the proposed implementation of the SCL and the Fault Injector in the wind power demonstrator. The SCL is implemented in both ends of the communication channel (only in the slave that manages safety input and output signals) and the Fault Injector in between. Only safety datagrams will be corrupted.

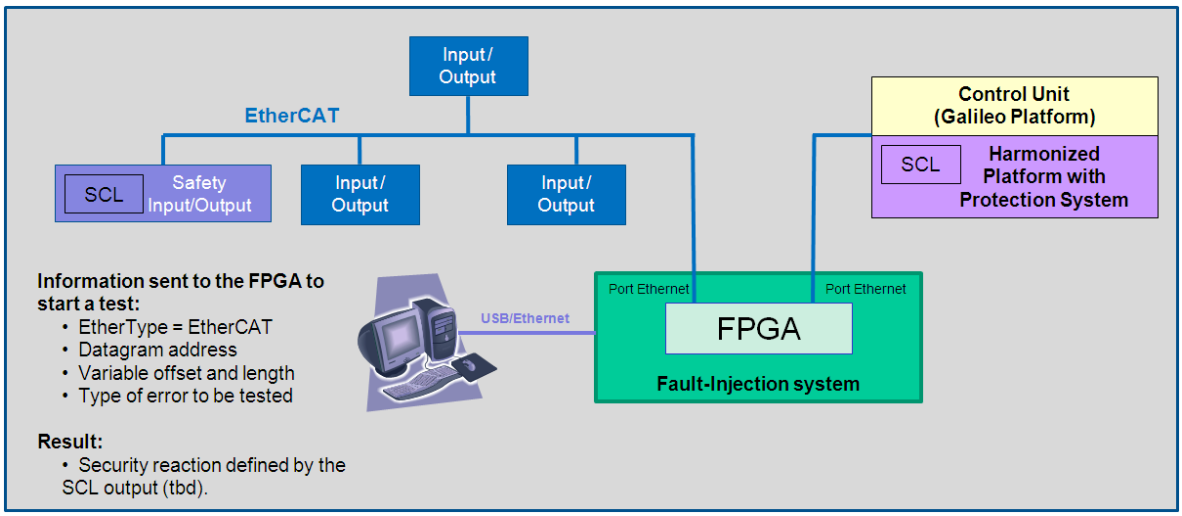

Figure 12 Evaluation of the Real-time Fault-Injection Framework in the wind power demonstrator

The delay introduced by the fault-injection system has been measured in section 2.3.2 and is approximately 7 µs (from the time it receives the first byte till it releases the first byte), which is OK for the EtherCAT communication implemented within the wind power demonstrator that has a cycle of 3 ms.

3 On-chip Fault-Injection Framework

3.1 Fault injection Emulation Platform

Hardware emulation is the next step after simulation. Without emulation, there is no way from the simulation point of view to run concurrently hardware and software. Hardware emulation is the only technology enabling to run applications in an acceptable time. Today there are two different technologies called FPGA-based emulator and emulators based on custom silicon. In the first class, the core element is a custom FGPA designed for emulation applications, while the second class, the core element is an array of simple Boolean processors that execute a design data structure stored in large memories. Regarding the underlying emulation technology used ST has developed a framework to extend the current hardware emulation flow toward the fault injection support. Fault injection is important to evaluate the dependability (the study of failures and errors) of a System on Chip. The dependability of a system is the study of the detection and isolation of failures and errors as well as the reconfiguration and recovery capabilities.

Often during the design phase the dependability of SoC is evaluated using simulation based fault injection which assumes that errors or failures occur according to a predetermined statistical distribution. The main drawback is the difficulty to provide accurate input parameters that can reflect the reality especially in the case of mixed criticality systems where the timing is critical. On the other side, testing a prototype will allow us to evaluate the system without any assumption yielding more accurate results. In fact, in a prototype-based fault injection methodology, faults are injected using random inputs enabling a better understanding of the effectiveness of fault tolerance mechanisms as well as concrete numbers of the coverage related to error detection and recovery mechanisms. Last but not least, implementing the fault injection using emulation it is possible to speed up and cover the huge number of faults that need to be checked.

The injection of faults can be done at hardware level and /or software level. In the first case we consider logical or electrical faults while in the second case we consider code or data corruption. Using the emulation based fault injection is a step further in this direction since instead of considering a prototype we consider a fully operational SoC. Therefore fault injection provides information about the real-life failures. Injecting random faults while running real software provides a way to validate the SoC against the actual errors and failures. However the only blocking point using emulation based fault injection is the amount of software/workloads that we are able to run.

3.2 The Fault Model

In this section we describe a new methodology to validate the functionalities and the robustness against soft errors of DREAMS chip instances. As described in Wikipedia, register-transfer level (RTL) is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. The register-transfer-level abstraction is used in hardware description languages (HDLs) like Verilog and VHDL to create high-level representations of a circuit, from which a lower-level representations and ultimately actual wiring can be generated. Design at the RTL level is typical practice in modern digital design. Considering the amount of details, RTL simulations are very slow implying the impossibility to simulate chip instances such as

DREAMS ones. In order to solve this problem emulation has been introduced enabling the complete verification of the system. Testing of digital circuits has traditionally been done using fault models:

- SEU: Single Event Upset
  - flip the value of a flip-flop (0→1, 1→0)
  - value remains corrupted until next write in register
- STUCK-AT 0/1 (STK)
  - stuck value of a flip-flop to 0/1 permanently
- MBU: Multiple Bit Upset
  - flip the value of several flip-flops (1→*n*)
  - similar behavior to SEU for each individual corrupted flip-flop
- MCU: Multiple Cell Upset
  - flip content of 1 to 6 memory cells according to the content stored in adjacent cells
  - values in cells remains corrupted until next write in memory cell
  - simulate physical propagation of error

3.3 The DREAMS Fault Injection Emulation Platform

As mentioned, soft errors, also referred to as Single-Event Upsets (SEUs), are transient faults caused by various types of radiation and interferences in the flip flops. Neutrons originated by cosmic rays, alpha particles originated by the natural emission of the molding composing of the packages of electronic devices, electro-magnetic interference and electrical noise can abruptly flip the stored state of a system and cause a wrong functionality. This liability is becoming more and more probable with the technology scaling and micro architectural trends. This creates system reliability concerns, especially for chips used in mixed-criticality systems such as automotive, healthcare, networking industries, and generally on electronics systems where functional safety is a requirement.

Figure 13 shows a fault injection environment which typically consists of the Device Under Test (DUT) (aka SoC) plus a fault injection library, a workload generator, workload library, controller, monitor, data collector and data analyzer.

The main component is the fault injection which injects faults into the emulated DUT and executes the software from the workload generator (baseline software with applications). It considers as a sampled space the possible combination of the word formed by the concatenation of the 44k injected flipflops of the DUT. This state space has a cardinality of 2^{44k}, however almost all of these combinations are unreachable in a normal behavior. We perform statistical sampling of this state space by injecting faults in flip-flops, and observe the behavior of the device under this fault. Effects of the fault are classified into 4 classes:

- Undiscovered not Propagating (Safe Undetected, i.e. Masking effect): the fault is inserted, but neither the Fault monitor nor the Application are affected (fault --> no error --> no failure)
- Undetected Propagating (Dangerous Undetected): the fault is inserted, the Fault Monitor is not affected, the Application fails (fault --> error -->failure) (unsafe condition)
- Detected not Propagating (Dangerous Detected): the fault is inserted, the Fault Monitor reacts and resets the device (critical fault detected), which goes in Safe mode; the Application stops (fault > error--> safe mode)

- Detected Propagating (Safe Detected): the fault is inserted, the Fault Monitor and the Application react and manages the error (non critical fault detected), the Application continues and it manages any fault.

Another component of the framework is the monitor that tracks the execution while the data collector tracks data at runtime. The data analyzer performs the data analysis. Since the fault space is then composed of a <flip-flop, cycle> tuple, we use a uniform sampling over both the flip-flop and cycle dimension to cover the fault space. If a state (set of states) occurs more frequently in the test (such as executing a loop), then the uniform sampling will also reflect the fact that the processor is more frequently in these states and the measured Soft Error Resilience (SER) will reflect it. In this way, the result of a fault injection campaign will reflect the real life observed SER of the device.

The controller is a piece of software that runs on a separate computer and controls the overall framework. For this reason it is often denoted as the software controller or software host controller.

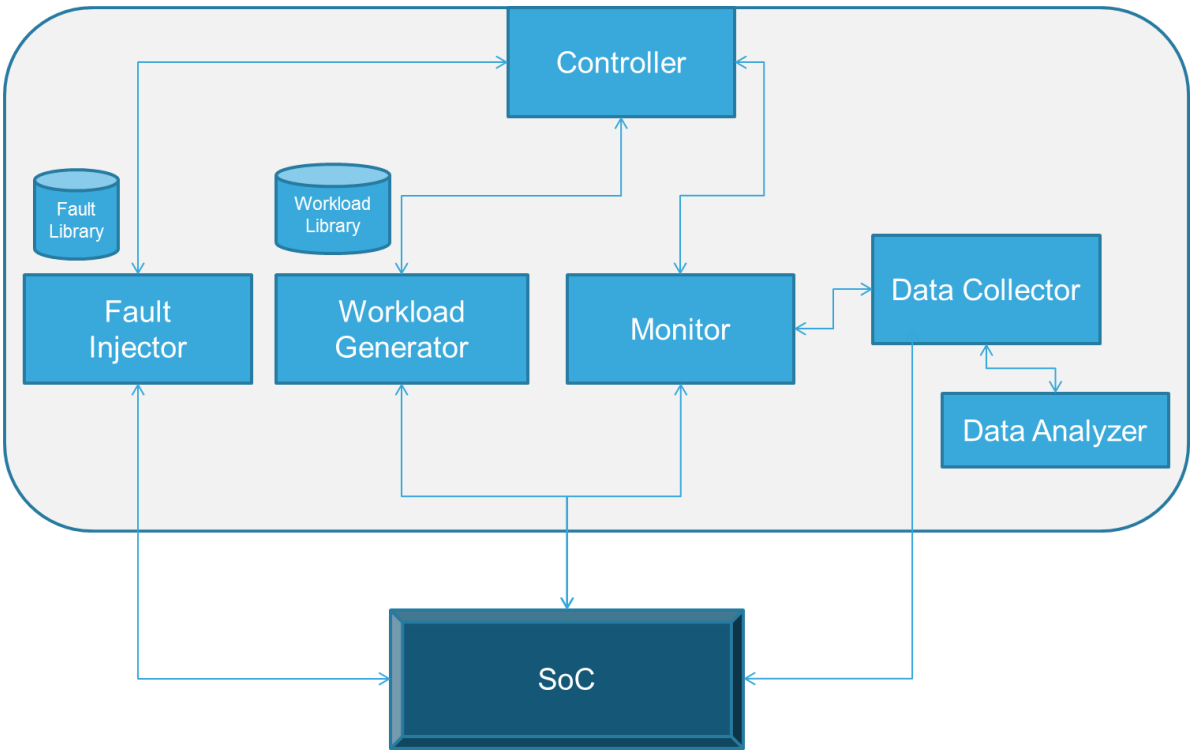

Figure 13 Fault Injection Framework

The fault injection emulation framework developed in DREAMS is composed of several DREAMS hardware blocks (synthesized on the hardware emulator platform) and a fault injection environment that runs on a host platform which allows for greater flexibility and portability.

The hardware emulation platform is based on Mentor Veloce technology which accelerates block and full SoC RTL simulations during all phases of the design process. Veloce2 enables pre-silicon testing and debugging at hardware speeds, using real-world data, while both hardware and software designs are still fluid. The key features are

- Scalable verification platforms with capacities from 16 million to 2 billion gates
- Common configuration and debug software across the Veloce family
- Simulator-like debug environment
- 100% internal DUT visibility
- Network accessible, multi-user systems.

The technology is based on custom silicon using a highly optimized SoC chip technology. The Veloce SoC, while FPGA-like in its computation resources, has a different kind of network for interconnecting the computation resources that is optimized specifically for faster compile time. The constraint in designing programmable logic cores is not to build the most optimized commercial FPGA, but to optimize the logic for emulation applications using a distinctive interconnect network. Veloce SoC supports a capability to independently compile the logic part of the design from the communication part; these are unique steps that use distinct, system-level resources for predictable compiles.

Using the Veloce emulators the synthetized DREAMS Hardware blocks are enhanced with so called flip-flop saboteurs. These flip-flops provide the injection method of the faults. For examples, each register in the DUT is enhanced with extra logic (called flip-flop saboteurs) that emulates a particular fault model by changing the flip flop value. The flip flop saboteurs are used to emulate the different fault models previously described. The flip-flop saboteurs introduce faults into the DUT and they are controlled by the software host controller. The software controller controls the different design-under-test instances and receives/sends fault data/fault results to the software host controller through the so called SCEMI-API interface. Thanks to the SCEMI interface, the software controller can be executed in a separate computer.

The SCEMI-API interface allows system co-emulation between an untimed software test bench linked to a design-under-test (DUT) which is for example mapped into an emulation platform through some communication channels as shown in Figure 14. In addition, it enables an easy integration of new and heterogeneous hardware emulator technology that supports this API. Currently, Cadence Palladium, EVE Zebu and Mentor Veloce are seamlessly integrated in the platform.

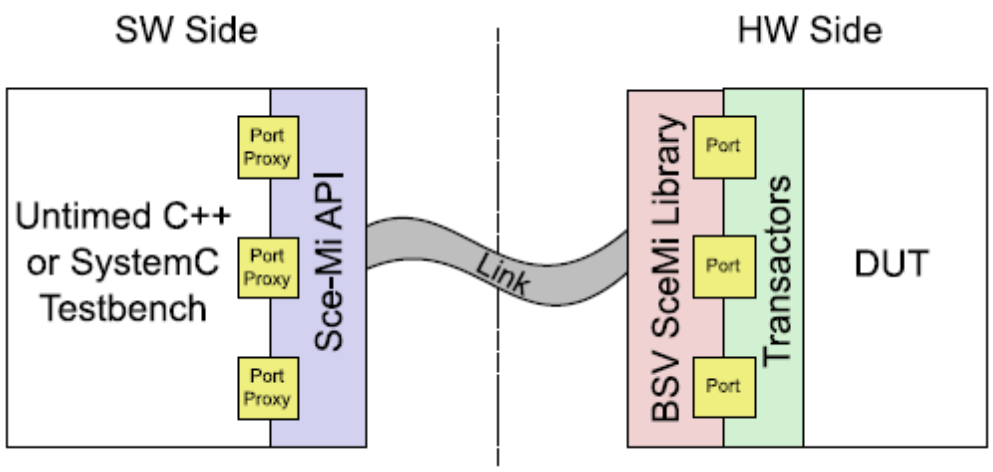

Figure 14 SCEMI API

In order to speed up the fault injection coverage an extra hardware controller is added to the DUT as shown in Figure 15. This hardware component, still controlled by the software controller enables to run in parallel different scenarios of fault injections reducing the time to reach the target level of fault coverage.

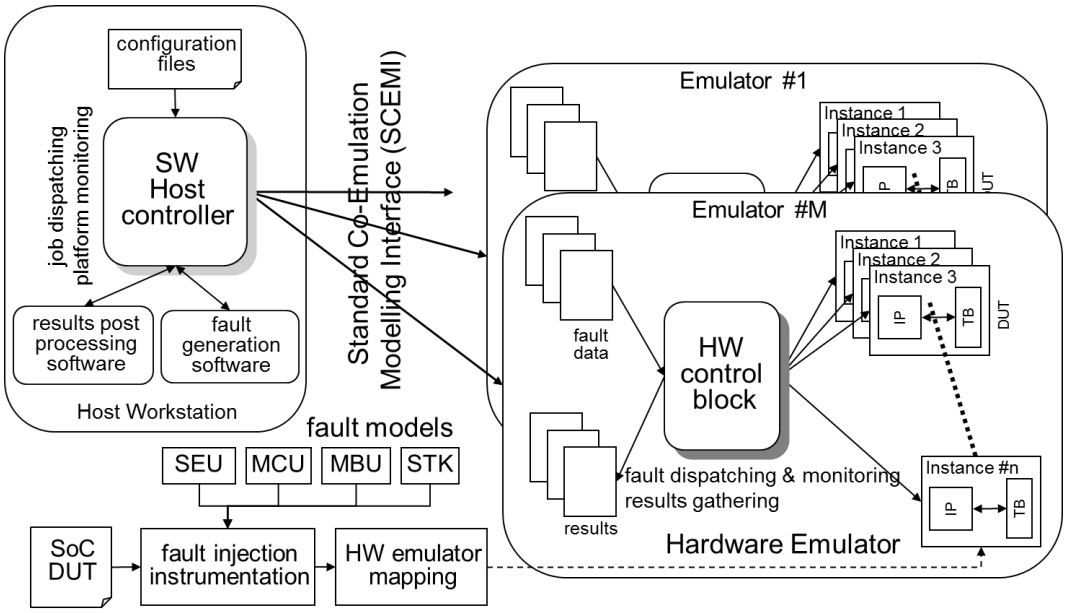

Figure 15 Emulation Platform

In fact the hardware control block enables to handle multiple parallel instances of the same DUT (SoC). It continuously monitors the state of the different instances of the DUT and starts a new fault injection as soon as a previous one is completed. The software controller is multithreaded and is able to control several hardware emulators in parallel.

3.4 Evaluation

3.4.1 Set-up

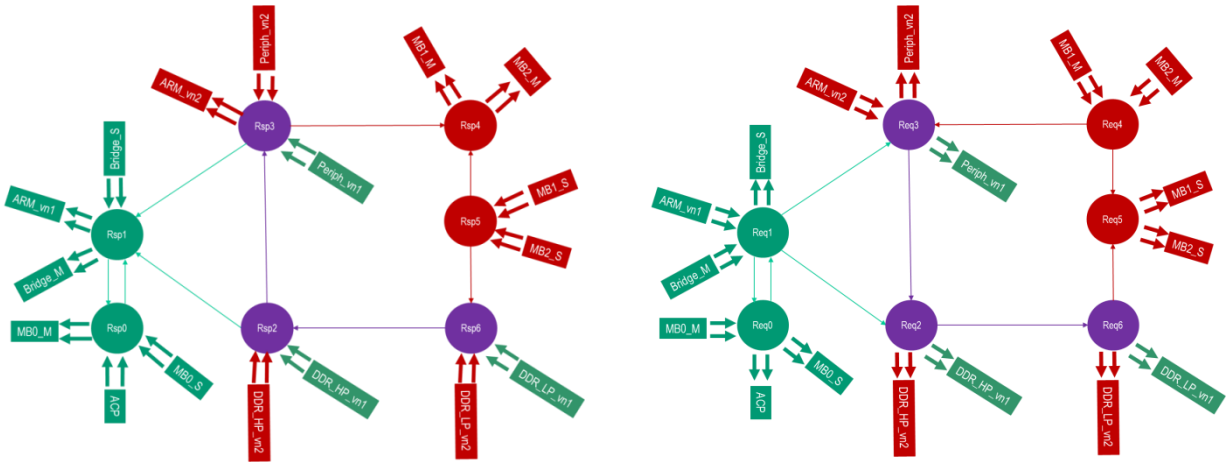

Figure 16 STNOC DREAMS instance

An example of a DUT that we have considered for the fault injection framework is the architecture of the STNoC as defined in DREAMS (see Figure 16) for the harmonized platform. This STNoC architecture was extended with the concept of Sphere of Replication (SoR) which refers to a set of replicated components that in this specific case are all the STNOC components and a Fault Monitor. A systematic check is performed at the output of the target network interfaces (NIs) by the RCCUs (Redundancy Control Checking Units). It is a simple comparator that combines the transactions coming from the 2 NIs to ensure that the same operations or transactions are executed on a clock per clock basis in both STNoC SoRs working in a Lock Step Mode of operation. During the execution, a SEU is injected in one of the STNoC SoR that is monitored by the RCCU modules

Using the fault injection flow we aim to detect faults as they leave the STNoC SoR. In fact, as long as a fault remains confined within the SoR, and therefore does not generate an action visible outside the SoR or influences the effective operability of the periphery, it shall not be considered as a dangerous fault.

The presence of RCCUs at the outputs of the STNoC SoR represents a minimum guarantee that non-common cause faults are detected when the two redundant channels are merged into a single actuator or recipient and the action is being performed. A programmable Fault Collection and Control Unit (FCCU) monitors the integrity status of the DUT and provides the SoC with flexible safe state control. A fault on the monitored components is being signaled by the RCCU to the FCCU. Having the test running automatically and logging the results allows a massive fault insertion, which in turn allows checking the RCCU-FCCU effectiveness. The FCCU constitutes an independent hardware channel to collect errors and to control the device to a safe state when a failure is present. No CPU intervention is requested for collection and control operation.

3.4.2 Test description

The test Application developed to exercise the fault-detection capability consists on a memory to memory data transfer: data contained in memory bank A is transferred through a DMA channel into the bank B, then transferred back to bank A; after this a checksum is calculated on the data, and compared with a reference signature. During the execution, a SEU is injected in one of the interconnect components monitored by the RCCU modules. This procedure is repeated 4000 times to ensure a high confidence interval and low error margin on the result.

The transfer-check test application exercises most of the STNoC components monitored by the RCCU. A fault on the monitored components is being signaled by the RCCU to the FCCU. Having the test running automatically and logging the results allows a massive fault insertion, which in turn allows checking the RCCU-FCCU effectiveness.

The main procedure of the fault injection framework is described hereafter

1. Workstation starts the test (send a power-on reset)
2. When the DUT is ready to receive the fault it notifies the workstation
3. Workstation logs time-stamp (a free running counter) and FCCU status registers.
4. Workstation injects one fault. The fault and the time-stamp are logged in the log-file.
    a. If FCCU is affected by the injected fault, it will notify to the workstation and the workstation logs the time stamp plus the FCCU status registers, then gives a power-on reset and starts a new test.
    b. If it is not affected, the DUT notifies the workstation that it is not anymore ready to receive faults. In addition the DUT sets the safety report variable (containing information on the application checksum integrity) and notifies the workstation about the normal termination of the test. Then, the workstation logs the time-stamp, the safety_report and the FCCU status registers.
5. Go to point #1

In order to measure the RCCU and Sphere of Replication effectiveness, the same test with injected faults is being run with the FCCU disabled. The result comparison of the two test sets (with and without the FCCU activated) will indicate how frequently dangerous faults are detected and neutralized by the RCCU-FCCU mechanism.

Since, by design, it is not possible to completely disable the FCCU, this is done modifying the design on Palladium, simply removing the FCCU connection.

For each component where faults are injected, the number of injected faults is chosen to 4000 to ensure a high confidence interval and low error margin on the result. Using this number of faults, a confidence interval of 98.86% at a ±1% error margin (98.96% at ±1% using continuity correction) is achieved and results can be considered as safe for the considered test application. Considering the length of the test (199282 cycles), the number of injected faults (4000) and the smallest unit of injected size there is no need to apply any Cochran correction on the number of injected faults. Therefore, a result X (percentage taken from the 4 class classifications) can be interpreted as follow: there is a 98.86% chance that the real value (which is unknown to us) lies in the range [X-1%, X+1%] of the measured value X. However, care must be taken when X is close to 0 or 1 (0% or 100%) as the estimator used to draw this conclusion (normal distribution) leads to growing imprecision as we get close to 0 or 1.

3.4.3 Test results

STNoC SoR	Nr Tests	FCCU (enabled / disabled)	Undetected Not Propagating	Detected Not Propagating	UndetectedPropagating TOTAL	Undetecte Propagating Wrong Results	Undetecte Propagating Delayed Results	Undetected Propagating Application Time Out
STNOC 0	4000	on	3896	104	0	0	0	0
STNOC 0	4000	off	3975	0	43	8	18	17
STNOC 1	4000	on	3897	103	0	0	0	0
STNOC 1	4000	off	3998	0	1	0	0	1

Figure 17 Tests results

As expected, with the FCCU enabled no fault is propagated to the application and the fault is either masked, or intercepted by the RCCU-FCCU mechanism. This reflects the 99% coverage figure claimed for the SoR (where the missing 1% coverage takes into account the common causes of failures, not modelled on Palladium). This gives evidence of the RCCU-FCCU mechanism verification.

The comparison of those results with the corresponding results with the FCCU disabled shows the effectiveness of the mechanism: while the "Application Time-out" and the "Delayed" results could be intercepted by a timer watchdog (and therefore could be moved to the Detected-Propagating column with a proper modification of the application), the wrong results can be detected only by a redundant architecture. The test therefore shows how many failures would be undetected on the implemented application without this architecture.

3.5 Conclusions

We have considered the STNoC architecture used in DREAMS enhanced with two redundant SoRs in order to improve safety and to be able to validate the fault injection framework. The focus of this work is mainly on the methodology to measure the effectiveness against soft errors, which are becoming an actual concern on current CMOS logic circuits. The fault injection framework implies the usage of different emulator platforms (Palladium, Zebu, Veloce) with a minimal RTL instrumentation overhead, and a statistical model which allows to achieve sample measurement results.

Part B: Simulation Environment and Fault Injection

4 Validation of the DREAMS Chip Simulation Framework

This chapter provides the final status update of the virtual platform simulation framework for the DREAMS architecture including the chip simulation and the hypervisor. Additionally, the final version of the simulation framework is evaluated using two use cases, a synthetic use case and an avionic use case. The avionic use case is based on the WP6 demonstrator, as described later in this chapter.

4.1 DREAMS Chip Co-Simulation Virtual Platform

The simulation framework uses two simulation tools to establish the desired DREAMS chip-level behaviour. Gem5 is used to simulate the on-chip network interconnect and a set of soft-core tiles of the DREAMS chip. Each tile has its own network interface with the support of the Local Resource Scheduler (LRS). Tiles run user defined trace-based applications, which are based on the exchange of messages with a priori defined setup and communication configurations. On the other hand, OVPSim is used to provide the simulation environment of the XtratuM hypervisor. The number of the partitions and their applications are configurable based on the use case as it will be illustrated.

The combination of these two simulators required the definition of a co-simulation strategy to guarantee the accuracy and integrity of the simulation. Therefore, a tick-based interleaving approach was introduced, in which the simulator tools are allowed to execute one at a time for a defined simulation step. This simulation step is defined based on the minimum execution step of the OVPSim simulator which is the bottle neck for defining the required execution step.

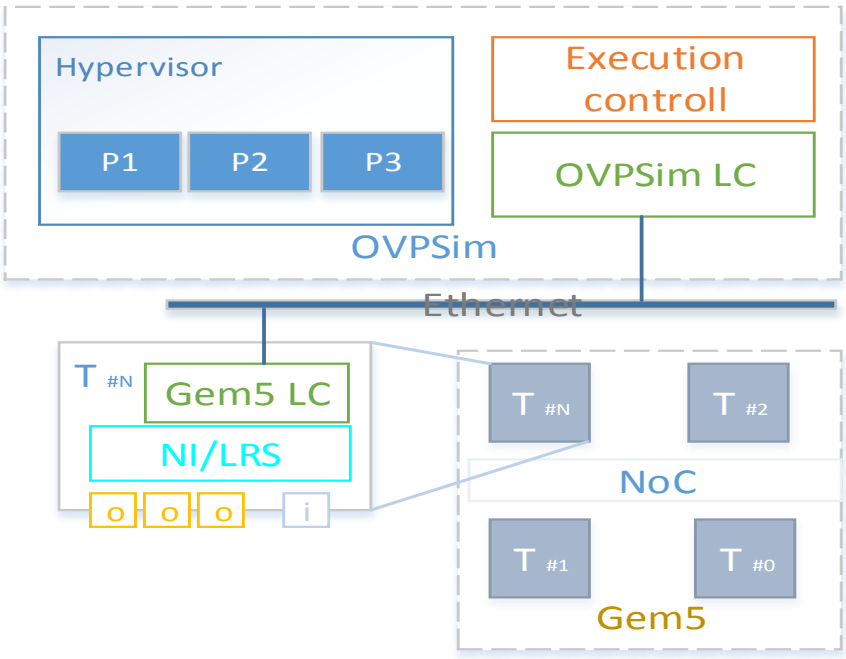

Figure 17: DREAMS Chip Co-Simulation Virtual Platform

This framework uses two simulation controllers, one for each of the simulation tools. These controllers are responsible for alternating the simulation step between the simulation tools by controlling the simulator calendar. Furthermore, they are liable for exchanging collected data and control messages. At Gem5 level, incoming and outgoing messages from and to the local controller are redirected to the corresponding destination based on the port mapping configuration. For instance, consider a message sent from application A1 from Partition P1 at the hypervisor tile destined to tile T2. The message has to

be redirected at the Gem5 local controller to the corresponding output port destined to T2, and vice versa. Further details about the simulation tools and the co-simulation approach are available in D5.2.2 “Prototype implementation of simulation framework for DREAMS architecture” and D4.2.1 “Specification and first implementation of a platform configuration files generator”.

4.1.1.1 OVPSim Configuration and Execution Instructions

OVPSim is responsible for simulating the execution of the processing system. To this end we have implemented a platform representing the different elements of the physical platform: memory, bus, processors and some devices. Furthermore, the platform contains the necessary elements to communicate with the Gem5 simulator.

4.1.1.2 OVPSim Virtual Platform Deployment.

The platform is supplied in a precompiled manner containing the directory structure shown in the following image. The platform directory contains the source files that make up the platform as well as the binary result of compiling these files.

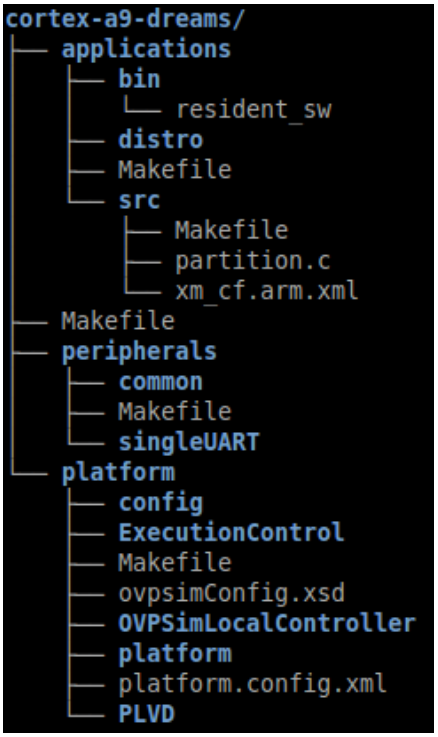

Figure 18: OVPSim contents tree

4.1.1.3 OVPSim Virtual Platform Configuration.

The simulation is configured using an XML file “platform.config.xml”. This file is placed in the “platform” directory. The contents of the xml file looks are illustrated in Figure 19 and described below:

4.1.1.3.1 Platform Description (platform)

This xml block describes the elements included in the platform. This block should not be modified since the platform is static. The shown configuration includes:

- 2 Processors corresponding with the ARM cores.
- 8 Memory areas OCMRAM, OCMROM and DDR representing the memory areas available in the board. The PhX memory areas have been added to fulfil the memory map.

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.fentis.com/ovpsimPlatform" >
  <icm autoconsole="acNONE"/>
  <platform>
    <processor variant="Cortex-A9MPx2" endian="little" coreName="cpu0" debug="false"/>
    <processor variant="Cortex-A9MPx2" endian="little" coreName="cpu1" debug="true"/>
    <memory name="OCMRAM"   start="0x00000000" size="192k"  permission="ICM_PRIV_RWX"/>
    <memory name="OCMROM"   start="0x00030000" size="64k"   permission="ICM_PRIV_RWX"/>
    <memory name="DDR"      start="0x00100000" size="1023M" permission="ICM_PRIV_RWX"/>
    <memory name="Ph0"      start="0xc0000000" size="512M"  permission="ICM_PRIV_RWX"/>
    <memory name="Ph1"      start="0xe0000000" size="4K"        permission="ICM_PRIV_RWX"/>
    <memory name="Ph2"      start="0xe0001060" size="928b"  permission="ICM_PRIV_RWX"/>
    <memory name="Ph3"      start="0xe0001400" size="1019K" permission="ICM_PRIV_RWX"/>
    <memory name="Ph4"      start="0xe0100000" size="511M"  permission="ICM_PRIV_RWX"/>
  </platform>
  <execCtrl/>
  <olc olc_type="olc_type_comms">
    <comms olc_comms_mode="olc_commsMode_Client" port="55441" host="127.0.0.1"/>
  </olc>
  <plvd>
    <mem PL_start_Address="0x40000000" PL_size="0x80000000"/>
  </plvd>
</config>
```

Figure 19: Xtratum platform configuration

4.1.1.3.2 OVPSim Local Controller (OLC)

This XML block describes the configuration parameters needed to configure the Local Controller that interconnects OVPSim and Gem5. The controller requires the following attributes:

- ***@olc_comms_mode***: This attribute indicates the connection role. In this case it is always "olc_commsMode_Client".
- ***@host***: This attribute indicates the IP address where the Gem5 server is placed to establish the connection.
- ***@port***: This attribute indicates the port to establish the connection.

4.1.1.3.3 Programming Logic Virtual Device (PLVD)

This XML block describes the configuration parameters needed to configure the Programming Logic Virtual Device. This attribute describes where the Programming Logic is placed in the Memory Map. Only the memory rage used by the DRNoC ports is needed to be declared here. The description of each attribute is detailed here:

- ***@PL_start_Address***: This attribute indicates the start address where the Programming Logic is placed in the Memory Map.
- ***@PL_size***: This attribute indicates the Programming Logic Size in bytes.

4.1.1.4 OVPSim Virtual Platform Execution.

In order to run the OVPSim virtual platform the user should navigate into the root directory and launch the executable indicating the application file as show in the next image:

```
@xmvm:~$ cd cortex-a9-dreams/
@xmvm:~/cortex-a9-dreams$ ./platform/platform.Linux32.exe -a <path_to_the_application>/resident_sw
```

Note: Since the Gem5 simulator acts as a server, it must be launched before the OVPSim virtual platform.

4.1.2 Gem5 Configuration and Execution Instructions

The simulation framework for the tiles and their interconnect implemented in Gem5 provides a configurable and fully adaptable framework. The configurations are defined based on the simulated use case. Configurations of the Gem5 simulation tool are divided in two major parts: generic framework configurations and DREAMS extension configurations. Generic configurations are related to the overall setup of the use case, and are defined at execution time. This includes the microarchitecture to be used (e.g. ARM, ALPHA), the number of tiles, caches size, topology (e.g. mesh) and the number of simulation cycles. The following figure illustrates the configuration/execution command of the Gem5 generic parameters.

```
./build/ARM/gem5.opt --debug-flags=NetworkTest --debug-start=0  configs/example/
ruby_network_test.py --num-cpus=4 --num-dirs=3 --topology=Mesh --mesh-rows=2 --sim-cycles=20000000 --inje
ctionrate=0.01  --fixed-pkts --maxpackets=200000 --garnet-network=fixed
```

Figure 20: Gem5 Execution Command Including Generic Configurations

In this example, we are running ARM based tiles for 20M cycles in a mesh topology for a fixed garnet network. Debug flags are used to trace the simulation progress. For DREAMS purposes additional flags were added to collect simulation results, to trace the extended network interface and the co-simulation execution (e.g., USiegenStats, USiegenTimeStamp, USiegenMsgTrace).

Configurations related to the DREAMS extensions define the hardware topology, ports, virtual links and configuration parameters. In addition to the time-triggered schedule for both egress bridging unit (EBU) and serialization unit. The simulation framework has a trace-based simulation execution, which means that the trace file configuration has to be defined based on each use case. This also includes the exchange of message configuration parameters.

Configuration files have the format of CSV files with one dedicated file for each configuration set. To define the above mentioned configuration parameters, files located in "/src/mem/ruby/network/garnet/fixed-pipeline/ConfigurationManager/" have to be modified accordingly.

4.2 Synthetic Use Case Validation

A synthetic use case is introduced in this section to validate the on-chip DREAMS virtual platform. This use case consists of four tiles as shown in Figure 21, each tile name consists of three digits: chip ID, partition ID and tile ID. In this use case we are demonstrating a single DREAMS chip with four tiles each with a single partition. Tile "0.0.3" is connected to the OVPSim simulator that will run the hypervisor. Three periodic applications will be running on the hypervisor tile that has a unique input port for receiving data input as a rate constrained data flow from tile "0.0.1". Moreover, ports of each tile are defined in the HL_PortConfig.csv file. This includes the port, core and partition IDs indicating the physical and logical addresses of the port. Finally, port types are defined in addition to the dedicated virtual link IDs that will be connected to this port. Virtual links are defined based on the type (i.e. TT, RC, or BE) of the virtual link. Periods are given in case of time-triggered VLs, while in case of rate constrained ones the MINT is defined. Additionally, the physical name of the resource and destination ports are given.

A simulation period of five microseconds is defined for this use case. Periodic messages are sent from tile 3 at an offset of 4100, 4500 and 4900 microseconds at ports 0, 1 and 2 accordingly. Based on this use case, messages received at tile "0.0.3" are processed and sent later to tile "0.0.1" as rate constrained messages.

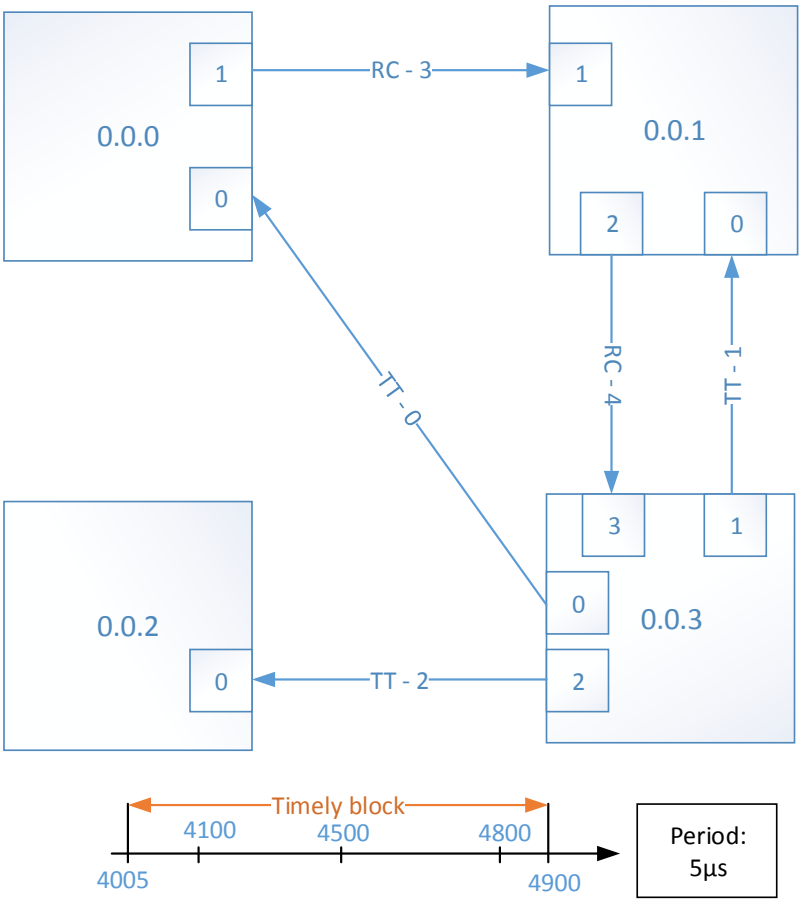

Figure 21: Synthetic Use Case Design and Configuration

The execution results of the use case are illustrated in the following figure.

```
                                rec. core | msgID | td | ts | PQU | t NI | core
NON found!: NON
  29124: system.cpu3: Stats @ Core[0], msg[100]: 124| 29000  29100;29101  29101;29106  29124
  34124: system.cpu3: Stats @ Core[0], msg[100]: 5124| 29000  34100;34101  34101;34106  34124
  34519: system.cpu3: Stats @ Core[1], msg[101]: 3519| 31000  34500;34501  34501;34506  34519
  34819: system.cpu3: Stats @ Core[2], msg[102]: 2819| 32000  34800;34801  34801;34806  34819
  39124: system.cpu3: Stats @ Core[0], msg[100]: 10124| 29000  39100;39101  39101;39106  39124
  39519: system.cpu3: Stats @ Core[1], msg[101]: 8519| 31000  39500;39501  39501;39506  39519
  39819: system.cpu3: Stats @ Core[2], msg[102]: 7819| 32000  39800;39801  39801;39806  39819
NON found!: NON
  44124: system.cpu3: Stats @ Core[0], msg[100]: 15124| 29000  44100;44101  44101;44106  44124
  44519: system.cpu3: Stats @ Core[1], msg[101]: 13519| 31000  44500;44501  44501;44506  44519
  44819: system.cpu3: Stats @ Core[2], msg[102]: 12819| 32000  44800;44801  44801;44806  44819
NON found!: NON
```

Figure 22: Beginning of the Gem5 simulation output

The simulation framework results are demonstrated at both Gem5 and OVPSim sides. Gem5 provides the status and statistics of the exchanged messages, and OVPSim shows the received and sent messages from/to the hypervisor partitions.

The simulation results of the synthetic use case are illustrated in Figure 22. The first column represents the destination core at which the message has arrived. The second column shows the message id. The third column represents the total delay (td) of the message, from the transmission of the message by the

sending core to the arrival of the message at the destination core. The forth column (ts) represents the tick at which the message left the sender core. The next column represents the tick at which the message arrived at the destination core. The next two columns (fifth and sixth) represent the enqueue and dequeue time in the *Priority Queues Unit.* Finally, the seventh and eighth column represent the time at which the message is delivered and the time at which the last flit leaves the network interface.

It can be noticed that at some instances the message “NON” occurs. This kind of message is printed in case that the Gem5 simulation tool has an empty buffer, which means that there are no messages received from the OVPSim simulation tool at this instant. This is due to the interleaving process of the simulation, and depends on the execution progress.

```
(          1) sUART:test !!
--->DATA 0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|0x40000000:0x
ee|
<---NON
--->DATA 0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|0x40000000:0x
ee|0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|0x40000000:0xee|
<---NON
--->DATA 0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|0x40010000:0x
ef|0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|
<---NON
--->DATA 0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|0x40010000:0xef|0x40018000:0x
11014110|0x40020000:0xf0|0x40020000:0xf0|0x40020000:0xf0|0x40020000:0xf0|
```

Figure 23: OVPSim simulation output

In Figure 23, messages buffered at the OVPSim local controller in both directions are tracked. The message format is based on the message format of the DREAMS physical platform as described in the previous deliverable D5.2.2.

```
NON found!: NON
 464124: system.cpu3: Stats @ Core[0], msg[100]: 405124| 59000  464100;464101  464101;464106  464124
 464519: system.cpu3: Stats @ Core[1], msg[101]: 403519| 61000  464500;464501  464501;464506  464519
 464819: system.cpu3: Stats @ Core[2], msg[102]: 402819| 62000  464800;464801  464801;464806  464819
NON found!: NON
 469124: system.cpu3: Stats @ Core[0], msg[100]: 410124| 59000  469100;469101  469101;469106  469124
 469519: system.cpu3: Stats @ Core[1], msg[101]: 408519| 61000  469500;469501  469501;469506  469519
 469819: system.cpu3: Stats @ Core[2], msg[102]: 407819| 62000  469800;469801  469801;469806  469819
 474124: system.cpu3: Stats @ Core[0], msg[100]: 415124| 59000  474100;474101  474101;474106  474124
 474519: system.cpu3: Stats @ Core[1], msg[101]: 413519| 61000  474500;474501  474501;474506  474519
 474819: system.cpu3: Stats @ Core[2], msg[102]: 412819| 62000  474800;474801  474801;474806  474819
 479124: system.cpu3: Stats @ Core[0], msg[100]: 420124| 59000  479100;479101  479101;479106  479124
 479519: system.cpu3: Stats @ Core[1], msg[101]: 418519| 61000  479500;479501  479501;479506  479519
 479819: system.cpu3: Stats @ Core[2], msg[102]: 416819| 63000  479800;479801  479801;479806  479819
 484124: system.cpu3: Stats @ Core[0], msg[100]: 425124| 59000  484100;484101  484101;484106  484124
 484519: system.cpu3: Stats @ Core[1], msg[101]: 423519| 61000  484500;484501  484501;484506  484519
 484819: system.cpu3: Stats @ Core[2], msg[102]: 417819| 67000  484800;484801  484801;484806  484819
 489124: system.cpu3: Stats @ Core[0], msg[100]: 430124| 59000  489100;489101  489101;489106  489124
 489519: system.cpu3: Stats @ Core[1], msg[101]: 428519| 61000  489500;489501  489501;489506  489519
 489819: system.cpu3: Stats @ Core[2], msg[102]: 422819| 67000  489800;489801  489801;489806  489819
Exiting @ tick 490000 because Network Tester completed simCycles
```

Figure 24: use case termination statistics (Gem5 side)

The synthetic use case executed and terminated correctly for 490 thousand cycles (cf. Figure 24).

5 Simulation Fault-Injection Framework

Fault injection (FI) is a widely used technique, to test fault-tolerant systems or components. In DREAMS, the simulation building blocks for fault injection allow to investigate the system behavior in the presence of component failures. Generic fault injectors can be instantiated and configured to inject specific message failures (e.g., babbling idiot, masquerading failure ...). The fault will be injected and observed in specified time intervals, and it is possible to inject the same fault several times. The fault type is defied based on following criteria:

- Emulate real world fault behaviour
- Emulate faults within the fault hypothesis (assuming single fault hypothesis, i.e., one fault can occur in one time interval). Further details about this assumption are in deliverable D1.2.1, section 2.

5.1 Fault Injector Implementation

The fault injection simulation building blocks are implemented using the OPNET modeler.

5.1.1 Omission Failure

These failures simulate the case where a sender is not able to generate a frame and/or a receiver is not able to receive a frame. The ”Packet discarder” provided by OPNET is used to implement the omission failure (see Figure 25).

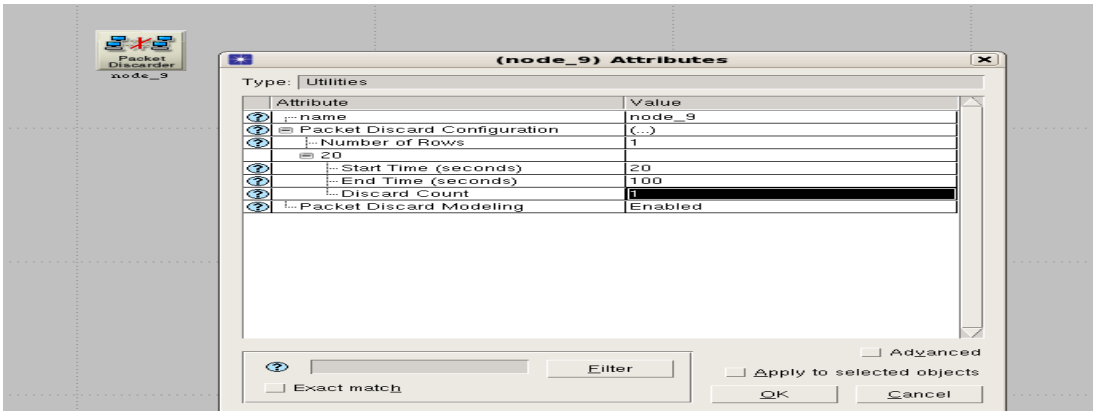

Figure 25: Packet Discarder with Attributes

The parameters for this module are the start time, the end time and the discard count. In addition, the place of this module must be defined, i.e., between which devices it is located.

This fault can be injected several times by using multiples “Rows” with every row having these parameters (Start time, End time and Discard count).

5.1.2 Corruption

This failure emulates errors during transmission (e.g., emulation of EMI disturbances), which introduce unintended changes to the original data. This type of failure can be applied to any link. The focus in the analysis of the corruption failure is to analyze its effects on the communication services.

In order to emulate this failure, OPNET provides the capability of modifying the link configuration module. The Bit Error Rate (BER) can be set for a specific link and with a specific start time (see Figure 26).

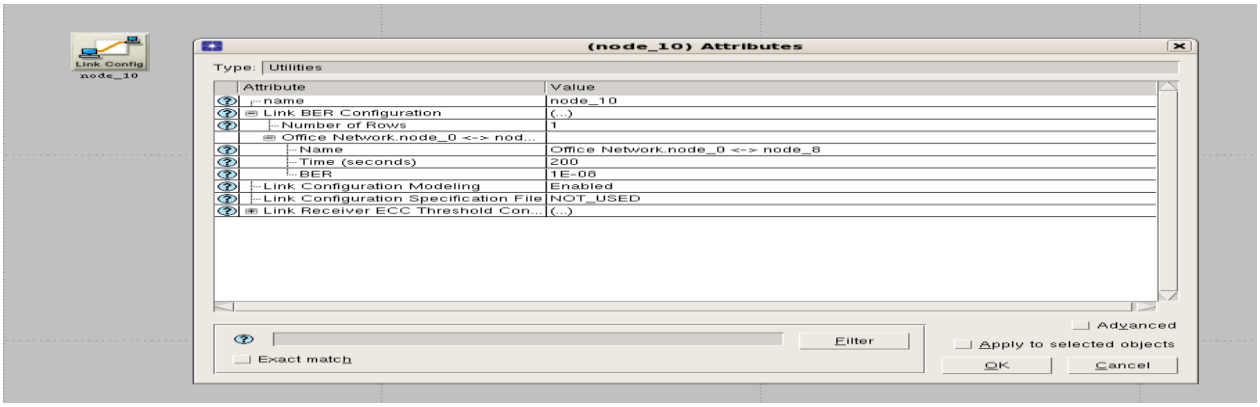

Figure 26 : Link Configuration with Attributes

The parameters for this module are the name of the link and the BER. In addition, the start and the end time define the intervals of the corruption failure.

5.1.3 Link Failure

The goal is to emulate transient crash faults of links for a specific time interval relative to the start of the simulation. The focus in this case is on testing the redundancy mechanisms of the different technologies. These failures are emulated by using the "Failure Recovery" provided by OPNET (see Figure 27). This involves the following attributes of this module: name of the link, time and status. The status parameter can be set to fail or recovery condition. Additionally, multiple injections of this failure can be initiated in this module by adding the corresponding rows.

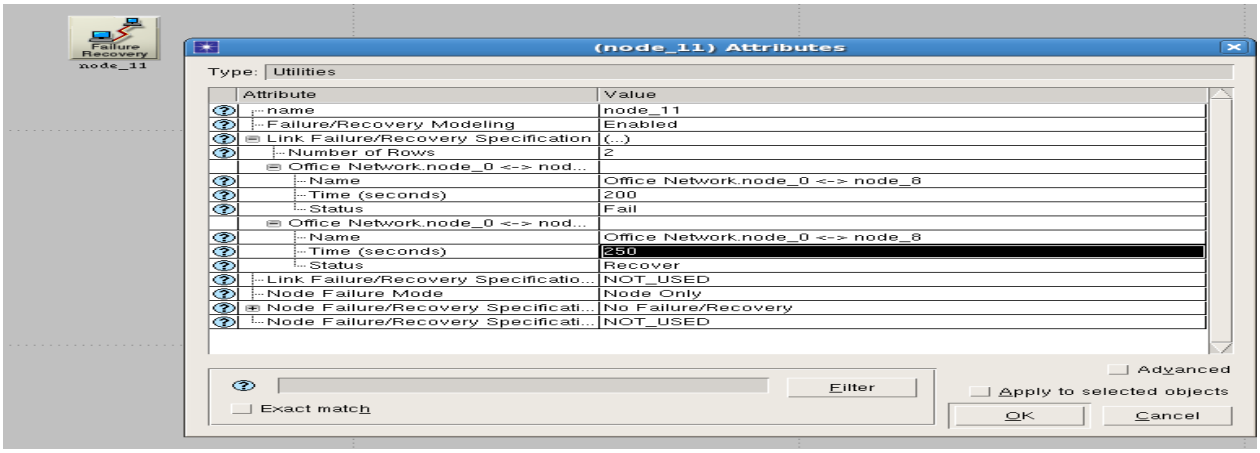

Figure 27: Failure Recovery with Attributes

5.1.4 Crash Failure

These failures simulate the crash failure of a device, i.e., the device shows no functionality by not generating output and by not receiving any input. Depending on the chosen fault injection time interval, this fault emulates the case when a device suffers a permanent fault or is being reset (transient fault). This failure type can be applied to different devices. A major system effect typically occurs, if this fault is injected at the switches.

These failures are emulated by using the “Failure/ Recovery” module for a detected device. In this module, we can define the time, the component or link, and the status.

5.1.5 Delay Failure

A faulty node or switch delays the start of the transmission of a frame (for all traffic types). At the node this failure will occur after the shaper layer. The following effects will be experienced in the different traffic types:

- Time-triggered frames: the frame will be dropped by the switch
- Rate-constrained and best-effort frames: frames exhibit longer end-to-end delays

These failures are emulated with the “Delay Failure” module. The parameters that control the delay failure are the start/end time and the delay time that is usually a constant.

All incoming frames to this module between the start and end time will be delayed by using the OPNET function “op_pk_send_delayed()”. Outside the fault injection interval the incoming frame will pass this module without any side effects.

5.1.6 Babbling idiot Failures

Babbling idiot failures occur when a node or a switch starts sending untimely frames or even generates additional frames, thereby causing more traffic load. In the simulation model, this behavior is implemented by activating a specific traffic generation pattern at a specific time interval.

These failures are emulated with the “Babbling Idiot Failure” module. The parameters that control the babbling idiot failure are the start and end time.

This module is a “queue process” module. This means that during the simulation, this module maintains queues and creates copies of incoming frames to use them later in the emulation of the behavior of the babbling idiot failure. When the interval for injecting the babbling idiot failure starts, the module starts to track one random packet from the queue and copies it. Thereafter, the module returns one of the copies to the queue and then continues sending to the other ones. This process is repeated until the fault interval finishes. During the fault interval, the process discards all incoming frames.

5.1.7 Masquerading Failure

A masquerading failure occurs if an erroneous node assumes the identity of another node. In the case of Time-Triggered Ethernet, a faulty node will send frames with incorrect VL-ID values for time-triggered and rate-constrained traffic. In case of best-effort traffic, the node will send frames with incorrect MAC addresses.

5.2 Evaluation

In this section we will introduce a simulation scenario of an avionic use-case based on an example of the DREAMS architecture (from WP6). To evaluate the safety aspects of the DREAMS architecture, we inject faults according to the fault assumptions.

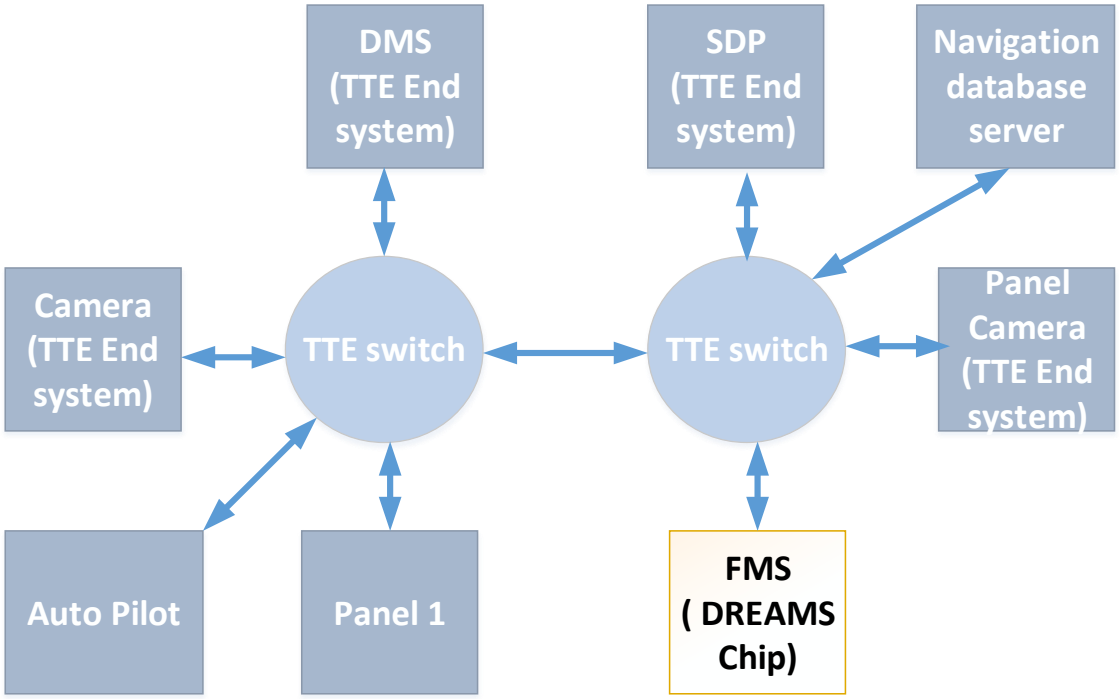

Figure 28: Avionic Use-Case Topology

5.2.1 Simulation Framework in the avionic demonstrator

Figure 28 depicts the topology of the simulated avionic use-case. The off-chip topology is composed of eight nodes, namely a navigation database server, sensor data provider (SDP), display management system (DMS), flight management system (FMS), auto pilot, panel and two cameras, interconnected by two time-triggered switches in a star topology. The FMS is composed of five interconnected tiles, each of which containing a single or more cores. Tiles are connected by a 2*2 mesh topology as shown in Figure 29. Each tile contains one or more cores that run tasks as listed in Table 1. As shown in Table 2 the data exchange of the applications is performed using periodic time-triggered messages, sporadic rate-constrained communication, and aperiodic messages.

Tile ID	Core ID	Description of the applications
Tile#0	Core#0	It presents different pilot actions corresponding to the management of the flight plan.
	Core#1	It has different tasks describing the pilot actions with respect to the localization tasks.
Tile#1	Core#0	It is a guidance task that is responsible for computing the parameters required to guide the aircraft.
	Core#1	Responsible for trajectory tasks
	Core#2	Nearest airports task builds during the flight a list of the nearest airports.
Tile#2	Core#0	It describes the different pilot actions that are translated into inputs for the sensor task.
Tile#3	Core#0	This task is responsible for gathering every sensor output to be passed to the localization task.

	Core#1	Different sensors used to manage and generate the most probable position of the aircraft.

Table 1: Applications Description

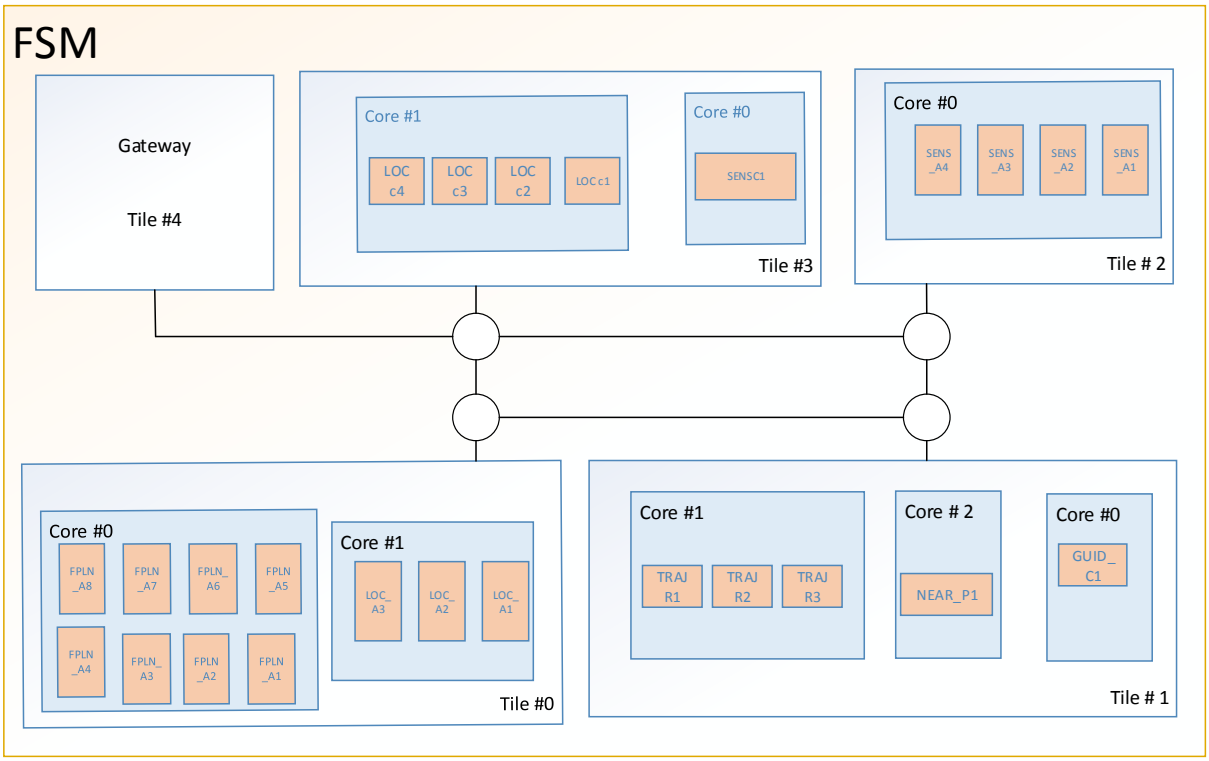

Figure 29: FMS

MSG. ID	Sender	Receiver/s	Type	time	meg size
1	SENS_A1	SENS_c1	Aperiodic	(0-0.2)s	72
2	SENS_A2	SENS_c1	Aperiodic	(0-0.2)s	72
3	SENS_A3	SENS_c1	Aperiodic	(0-0.2)s	72
4	SENS_A4	SENS_c1	Aperiodic	(0-0.2)s	72
7	LOC_A1	LOC_c1	Aperiodic	(0-0.2)s	72
8	LOC_c2	Trajectory, nearest Airport	Periodic	1.6 s	72
11	LOC_A2	LOC_c3	Aperiodic	(0-0.2)s	72
12	LOC_A3	LOC_c4	Aperiodic	(0-0.2)s	72
13	LOC_C4	DSM	Periodic	1 s	72
14	NAVi.	F plan	Aperiodic	(0-2)s	72
15	Temporary F plan	DSM	Periodic	0.3s	72
16	secondary F plan	DSM	Periodic	0.3 s	72

17	Active F plan	Trajectory	Periodic	0.2 s	72
18	Active profile	DSM	Periodic	0.2 s	72
19	Temporary profile	DSM	Periodic	0.3s	72
20	secondary profile	DSM	Periodic	0.3 s	72
21	NEAR _p1	DSM	Periodic	1 s	72
22	GUID_c1	Auto pilot	Periodic	0.2s	72
23	SPD	DMS	Periodic	200 ms	72
24	Panel	DMS	Sporadic	(0-.2)s	72
25	SDP	SENS_A1	Periodic	0.1s	72
26	DMS	Panel	Periodic	0.2 s	72

Table 2: Message Exchange in the Avionic Use-Case

Table 3 lists the simulation results for the evaluation of the avionic use-case. We observed significant discrepancies of the end-to-end jitter for different traffic types, i.e., the difference between the maximum and minimum end-to-end latency between the applications.

Msg. ID	Latency	Jitter
1	389 ns	87 ns
2	346 ns	60 ns
3	374 ns	76 ns
4	320 ns	55 ns
7	419 ns	81 ns
8	249 µs	0
11	453 ns	85 ns
12	519 ns	93 ns
13	231 µs	0
14	348 µs	83 µs
15	215 ns	0
16	240 ns	0
17	179 ns	0
18	238 µs	0
19	221 µs	0
20	247 µs	0
21	273 µs	0
22	236 µs	0
23	145 µs	0
24	283 µs	105 µs
25	162 µs	0
26	136 µs	0

Table 3: Simulation Results

6 Implementation of Model-Driven Configuration for the Simulation Framework

In Section 6.1 we show how the generation of the configuration files for the virtual platform can concretely be executed, whereas in Sections 6.2 and 6.3, we explain the changes and extensions that have been performed at the level of the format and contents of the configuration files.

6.1 How to Configure and Execute the Generation

6.1.1 Description

The configuration tool is implemented as a plugin of Autofocus. It allows exporting the scheduling parameters defined in Autofocus for the on-chip and off-chip communication, as a set of configuration files in textual format (CSV) defined in Section 6.2 and 6.3 of D5.2.2 [1], in addition to the following subsections of the deliverable. These CSV files are parsed by the corresponding simulator components during initialization.

6.1.2 Relations and dependencies with other tools

The inputs required by the tools must have been created before, manually with the help of model editors or through dedicated tools, see D4.4.1 [2].

6.1.3 Inputs

The generated configuration parameters are drawn from the following system description items:

- Logical architecture consisting of applications (top level components) and tasks (leaf-level components) that exchange data.
- Technical architecture, that describes the available processing and communication resources
- System software
- Deployment, which describes the virtual links
- System schedule, which contains the actual scheduling parameters

Since a “System Schedule” has direct and indirect references to all other require inputs, this schedule is the only parameter that needs to be directly passed to the timing analysis tool.

6.1.4 Outputs

The output is a set of textual configuration files. As can be seen in Figure 30, the produced folder contains the configuration files for the message injection and two subfolders are generated, one for the off-chip network and one for the on-chip network.

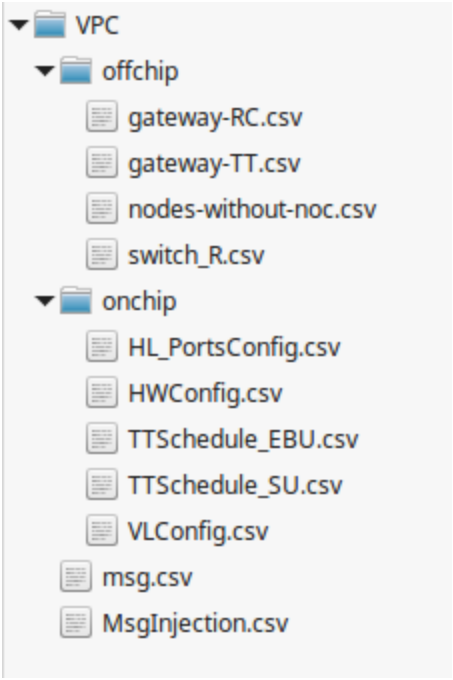

Figure 30: Structure of virtual platform configuration file set

6.1.5 Running the generator

The generation of the configuration files is a two-step process:

1. Generation of the “Configuration Model”, based on the system model
2. Generation of the configuration files, based on the configuration model

The configuration model is defined in D1.6.1 [3] (Section 4.3 for on-chip communication and Section 4.4 for the off-chip communication).

In order to generate **the configuration model**, right -click on System Schedule node that contains the on-chip communication scheduling parameters, and select the “Generate ‘Virtual Platform …’” entry:

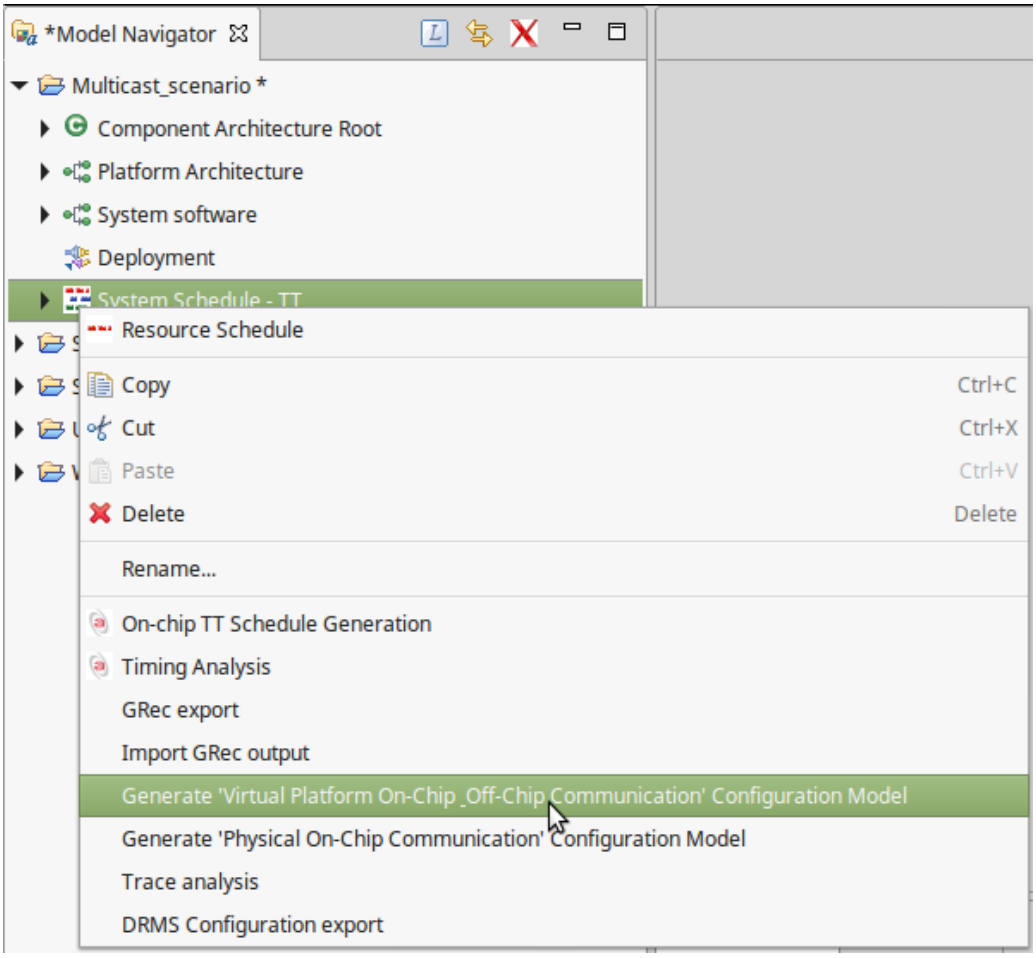

Figure 31: Context menu for generating the configuration files for the virtual platform

As a result, a dialog is shown for entering the length of the time interval in seconds, for which the injection of messages (MsgInjection.csv) is automatically generated:

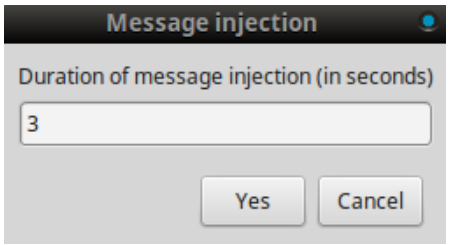

Figure 32: Specifying the duration of message injection.

The injection time of the first instance is randomly chosen between 0 and the period or the MINT of the virtual link. The following instances are injected after a time equal to

- the period of time-triggered virtual links
- the MINT + random fraction of the MINT (up to up to 10%) in order to create some drift in the injections and different scenarios on the bus.

Finally the “Configuration Model” corresponding to the system model is created or updated in the “Storage view”. In order to make the configuration model visible or to make the updates effective, you must first apply “Save” to the system model:

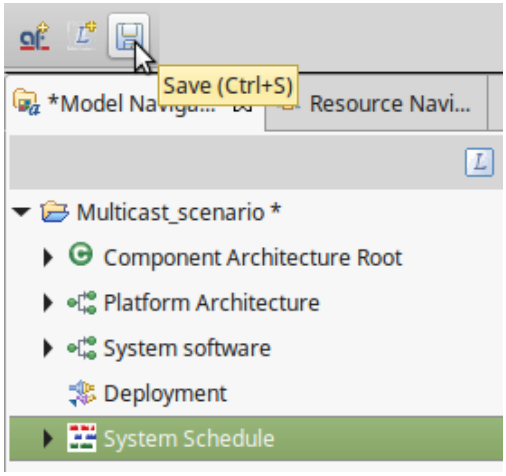

Figure 33: Persisting the generated configuration model

Then, switch to the “Resource Navigator” view, by clicking on the tab besides the “Model Navigator”:

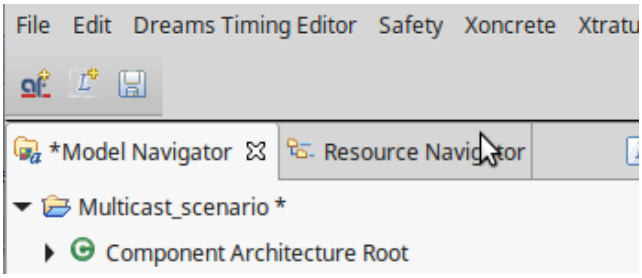

Figure 34: Switching to the "Resource Navigator"

Finally, double click on the corresponding “Configuration Model” node:

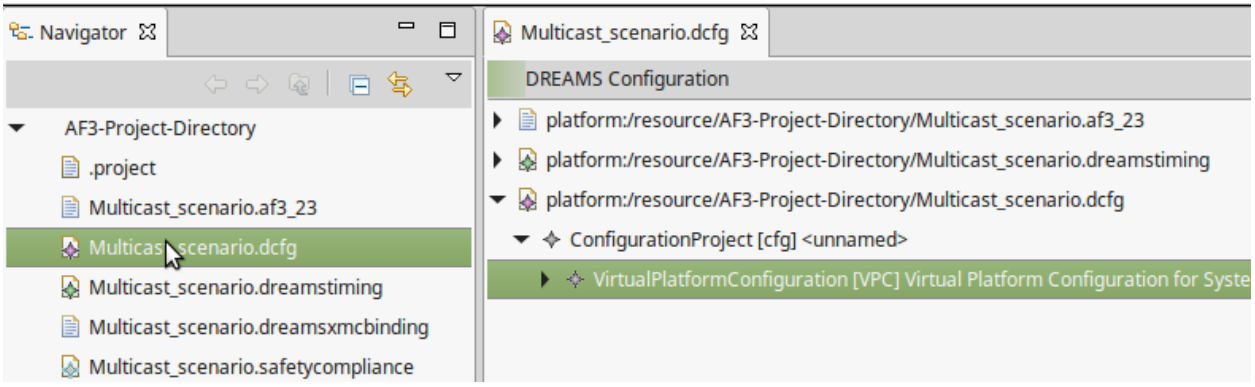

Figure 35: Visualizing the generated configuration model

Below the “VirtualPlatformConfiguration” nodes are accessible in all sub-configurations. Some of the properties may be set or modified in the properties view:

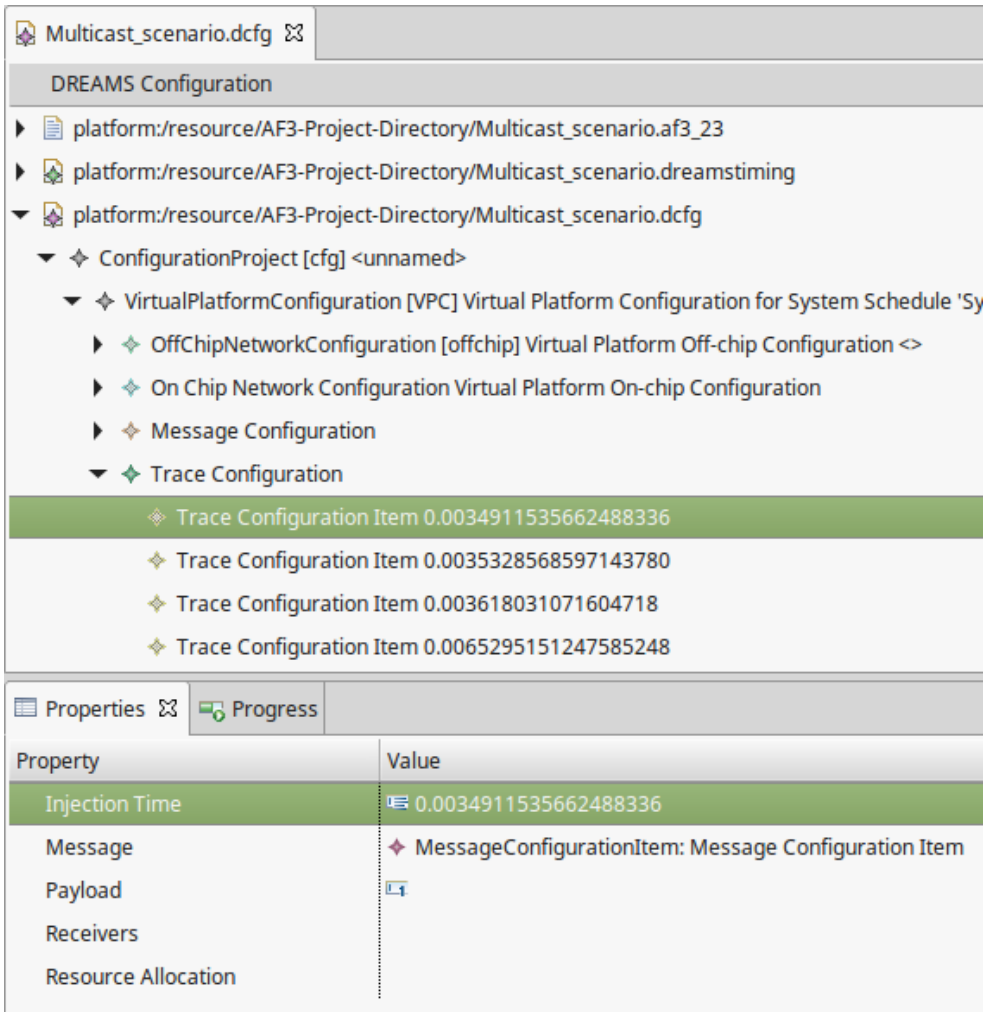

Figure 36: Visualization of configuration model items

In order to perform the last step, select the “Run Configuration Generation Framework” entry from the context menu of the model:

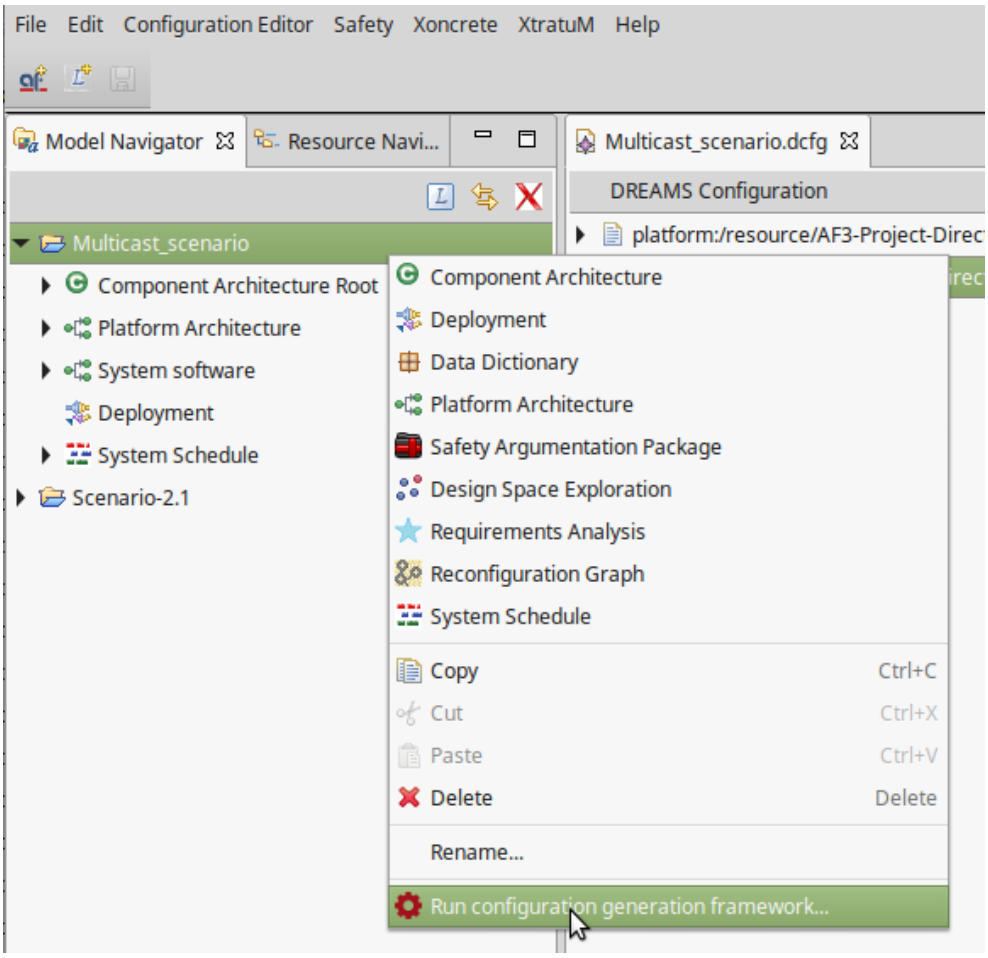

Figure 37": "Run Configuration Generation Framework" menu entry

As a result, the configuration menu is opened:

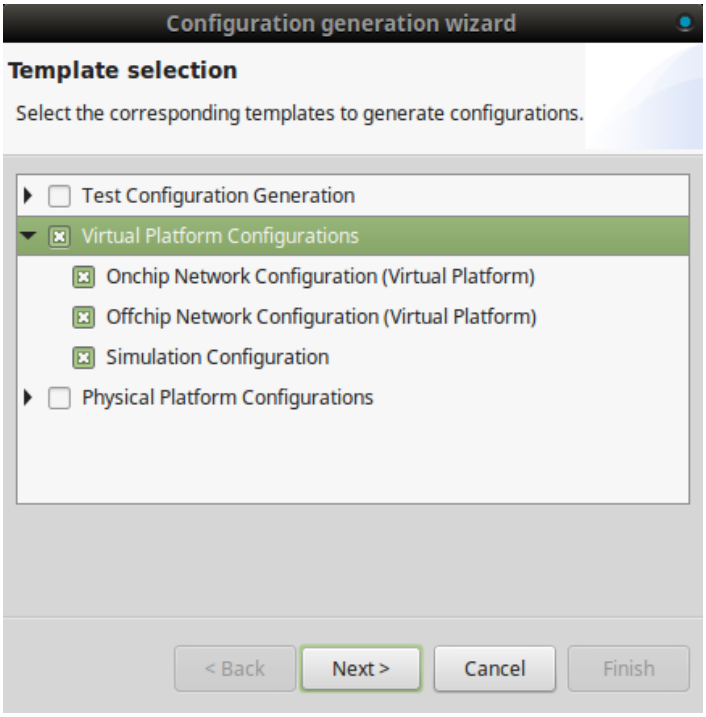

Figure 38: Selection of the configuration files to generate.

Select the "Virtual Platform Configurations" group and click next, in order to select the folder where the "cfg" folder (see Section 6.1.4) with all configuration files will be created; finally click "Finish" to perform the generation.

6.2 Configuration Files for the Cluster Level

6.2.1 Specification of Configuration File Formats with Examples

An additional parameter for queue lengths has been added to all configuration files at cluster level, at the end of the lines:

- **Queue length:** maximal number of messages that can be stored.

A corresponding getter method has been added to all concerned configuration meta-model entities. Examples of the concerned configuration files are shown in the following sections.

6.2.1.1 Simulated TTEthernet Switch

switch_R.csv:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
SW	R													
Trafic Type	VL	Period (us)	Phase(us)	Sender port	Reciver port	Reciver port	Reciver port	Reciver port	Reciver port	Reciver port	Reciver port	Size	Queue ID	queue_length
TT	0	10000	1000	1	2	3	-1	-1	-1	-1	-1	100	50	1
RC	2	15000	500	1	2	3	-1	-1	-1	-1	-1	200	51	1
TT	4	10000	1000	2	1	3	-1	-1	-1	-1	-1	300	52	1
RC	5	15000	500	2	1	3	-1	-1	-1	-1	-1	400	53	1

Figure 39: Example of a switch configuration file.

6.2.1.2 Simulated Node with a Core

nodes-without-noc.csv:

A	B	C	D	E	F	G	H
# Node Nam	VLID	Period/MIN	start Time	Traffic Type	Message siz	Queue ID	queue_length
Node2	4	20	0	TT	300	0	1
Node2	5	30	0	RC	400	2	1

Figure 40: Example of the configuration file for nodes without NoC.

6.2.1.3 Simulated On-chip/Off-chip Gateway: Time-Triggered Virtual Links

gateway-TT.csv:

A	B	C	D	E	F	G
vl_id	period(us)	phase(us)	msg_size	direction	queue_id	queue_length
0	10000	500	100	2	1	1
4	20000	1000	300	1	3	1

Figure 41: Example of the on-chip/off-chip gateway configuration file for RC virtual links.

6.2.1.4 Simulated On-chip/Off-chip Gateway: Rate-Constrained Virtual Links

gateway-RC.csv:

A	B	C	D	E	F	G
vl_id	bag(us)	priority	msg_size	direction	queue_id	queue_length
2	15000	1	200	2	2	1
5	30000	1	400	1	4	1

Figure 42: Example of the on-chip/off-chip gateway configuration file for TT virtual links.

6.3 Configuration Files for the Chip Level

6.3.1 Specification of Configuration File Formats

An additional parameter for queue lengths has been added to all configuration files at the cluster level, at the end of the lines:

- **Queue length:** maximal number of messages that can be stored.

A corresponding getter method has been added to all concerned configuration meta-model entities. Examples of the concerned configuration files are shown in the following sections. Furthermore, since the configuration file for the “TT schedule for the serialization unit” has been slightly changed and since the automatic generation of the contents of the “Message injection” has been added, the corresponding sections are also shown.

6.3.1.1 Ports configuration parameters

HL_PortConfig.csv:

A	B	C	D	E	F	G	H	I	J
# Port ID	Core ID	Partition ID	Phys. Addr.	Log. Addr.	Type	VL ID	Direction	Semantics	Queue Length
# Port 1-1-1	Core 1-1-1	Partition 1-1-1	Cluster.Node1.Tile 1-1.Port-1-1-1	Criticallity.Subsystem.Component.Message		VL_1_TT			
0	0	0	0.0.0.0	0.0.0.0	TT	0	OUT	STATE	1
# Port 1-1-3	Core 1-1-1	Partition 1-1-1	Cluster.Node1.Tile 1-1.Port-1-1-3	Criticallity.Subsystem.Component.Message		VL_1_TT_NOC			
2	0	0	0.0.0.2	0.0.0.1	TT	1	OUT	STATE	1
# Port 1-1-2	Core 1-1-1	Partition 1-1-1	Cluster.Node1.Tile 1-1.Port-1-1-2	Criticallity.Subsystem.Component.Message		VL_2_RC			
1	0	0	0.0.0.1	0.0.0.2	RC	2	OUT	EVENT	1
# Port 1-1-4	Core 1-1-1	Partition 1-1-1	Cluster.Node1.Tile 1-1.Port-1-1-4	Criticallity.Subsystem.Component.Message		VL_2_RC_NOC			
3	0	0	0.0.0.3	0.0.0.3	RC	3	OUT	EVENT	1
# Port 1-1-5	Core 1-1-1	Partition 1-1-2	Cluster.Node1.Tile 1-1.Port-1-1-5	Criticallity.Subsystem.Component.Message		VL_3_TT			
4	0	1	0.0.0.4	0.1.0.4	TT	4	IN	STATE	1
# Port 1-1-6	Core 1-1-1	Partition 1-1-2	Cluster.Node1.Tile 1-1.Port-1-1-6	Criticallity.Subsystem.Component.Message		VL_4_RC			
5	0	1	0.0.0.5	0.1.0.5	RC	5	IN	EVENT	1
# Port 1-2-1	Core 1-2-1	Partition 1-2-1	Cluster.Node1.Tile 1-2.Port-1-2-1	Criticallity.Subsystem.Component.Message		VL_1_TT_NOC			
0	0	0	0.0.1.0	0.2.0.0	TT	1	IN	STATE	1
# Port 1-2-2	Core 1-2-1	Partition 1-2-1	Cluster.Node1.Tile 1-2.Port-1-2-1	Criticallity.Subsystem.Component.Message		VL_2_RC_NOC			
1	0	0	0.0.1.1	0.2.0.0	RC	3	IN	EVENT	1

Figure 43: Example of the port configuration file.

6.3.1.2 TT Schedule for the Serialization Unit

The configuration with the global parameters of the time triggered communication of the on-chip network has been slightly changed: the timely blocking option is now a global parameter.

6.3.1.2.1 Definition of required configuration parameters

On-chip global parameter:

- **Timely block/shuffling*:*** activation of timely block in the entire chip. In case of non activation, i.e. activation of shuffling, the information given for the guarding window will be ignored *(1 activated, 0 deactivated)*

Guarding window parameters

- **Tile id *:*** numerical identifier *(integer)* of a tile
- **Period:** period of the guarding window, expressed in ticks ***(integer)***
- **Opening phase:** Starting phase of the guarding window, expressed in ticks ***(integer)***
- **Closing phase:** Closing phase of the guarding window, expressed in ticks ***(integer)***

6.3.1.2.2 CSV File Format

The first cell of the first line contains the parameter “Timely block/shuffling”. The following lines of the CSV file correspond each to a guarding window. The cells contain the values of the parameters in the order in which they are described in the previous section.

6.3.1.2.3 Sample file

TTSchedule_SU.csv:

	A	B	C	D
1	# Timely block / shuffling			
2	1			
3	Tile ID	Period (tick=ns)	Opening phase (tick=ns)	Closing phase (tick=ns)
4	# Tile 1-1	10 ms	0 ns	500 µs
5	0	10000000	0	500000
6	# Tile 1-2	10 ms	1 ms	1,5 ms
7	1	10000000	1000000	1,5

Figure 44: Example of the serialization unit configuration file.

6.3.1.3 Message Injection Configuration File

One configuration file is used to specify the times when message instances are injected into the partition ports. The generation of these injection times for a certain time horizon has been integrated into the configuration file generator.

MsgInjection.csv:

	A	B	C	D	E	F
1	# Tile ID	Tick (tick=ns)	Msg ID	Port ID	Logical Adress	Payload
2	# Tile 1-1		Msg_TT	Port 1-1-1		
3	1	50	0	0	-1	0
4	# Tile 1-1		Msg_TT_Noc	Port 1-1-3		
5	1	10000	1	2	-1	0
6	# Tile 1-1		Msg_RC	Port 1-1-2		
7	1	100000	2	1	-1	0
8	# Tile 1-1		Msg_RC_Noc	Port 1-1-4		
9	1	110000	3	3	-1	0
10	# Tile 1-1		Msg_TT	Port 1-1-1		
11	1	10000050	0	0	-1	0
12	# Tile 1-1		Msg_TT_Noc	Port 1-1-3		
13	1	10010000	1	2	-1	0
14	# Tile 1-1		Msg_RC	Port 1-1-2		
15	1	15100000	1	1	-1	0
16	# Tile 1-1		Msg_RC_Noc	Port 1-1-4		
17	1	15110000	3	3	-1	0
18	# Tile 1-1		Msg_TT	Port 1-1-1		
19	1	20000050	0	0	-1	0
20	# Tile 1-1		Msg_TT_Noc	Port 1-1-3		
21	1	20010000	1	2	-1	0
22	# Tile 1-1		Msg_RC	Port 1-1-2		
23	1	23100000	1	1	-1	0
24	# Tile 1-1		Msg_RC_Noc	Port 1-1-4		
25	1	23110000	3	3	-1	0

Figure 45: Example of the message injection configuration file.

7 Analysis of Simulation Traces

In this section we describe the extensions related to fault injection, implemented in the RTaW-Timing tool for the analysis and visualization of communication traces produced by the DREAMS simulation tools.

7.1 Trace Files

In this subsection we describe the extensions of the trace file format related to frame drops, where two different reasons are considered.

7.1.1 Off-chip Network Related Events

In the off-chip network domain, frame drops are considered in the on-chip/off-chip gateways and in the routers.

<TIME> FrameDropped <DROP_CAUSE> Gateway <GATEWAY_ID> <FRAME_ID> <FRAME_INST_ID>

<TIME> FrameDropped <DROP_CAUSE> Router <ROUTER_PORT_ID> <FRAME_ID> <FRAME_INST_ID>

Two causes are considered:

1. **Lack of memory**: there was no free slot in the frame queue for emission
2. **Frame Check Sequence Error**: at reception some bits have been detected to be altered

Depending on the cause of the drop and the concerned entity, the frame drop event may replace a different nominal event:

Drop cause	Entity	Replaced event
LackOfMemory	Gateway	FrameQueued event, since the frame is not queued in case of insufficient memory.
	Router	FrameTx event in router ports, since the queuing event is not considered and the frame will never be emitted if dropped.
FCSError	Gateway	FrameRx event, since the FCS error is detected at the reception.
	Router	

Figure 46: Frame drop events

7.1.2 On-chip network related events

In the off-chip network domain, only frame drops due to lack of memory and only the network interfaces are considered. For instance the wormhole routing does only forward flits if buffers are free.

<TIME> MessageDropped < DROP_CAUSE> OutPort <PORT_ID> <MESSAGE_ID> <MSG_INST_ID>

The “FrameDropped” event replaces the MessageQueued, since the packet is not queued in case of insufficient memory.

7.2 Export of DREAMS System Description to RTaW-Timing Tool

In this section we describe how to export the description of a DREAMS model to the RTaW-Timing tool, so that the tool acquired the necessary knowledge of the system under study. The screen-shots are based on the “Virtual-Platform Scenario”, described in Section 6.1 of [1].

To be able to perform the import and analysis of DREAMS simulation traces, the description of the simulated system needs first to be exported into the RTaW-Timing tool. For this purpose, an Autofocus plugin has been developed in T4.4 as part of the tool-chain (see D4.4.1), which performs the export and launches the RTaW-Timing tool.

To perform the export, right-click on the “System Schedule” entity of the system model and select the "Export 'System Description' for 'Trace Analysis'" entry.

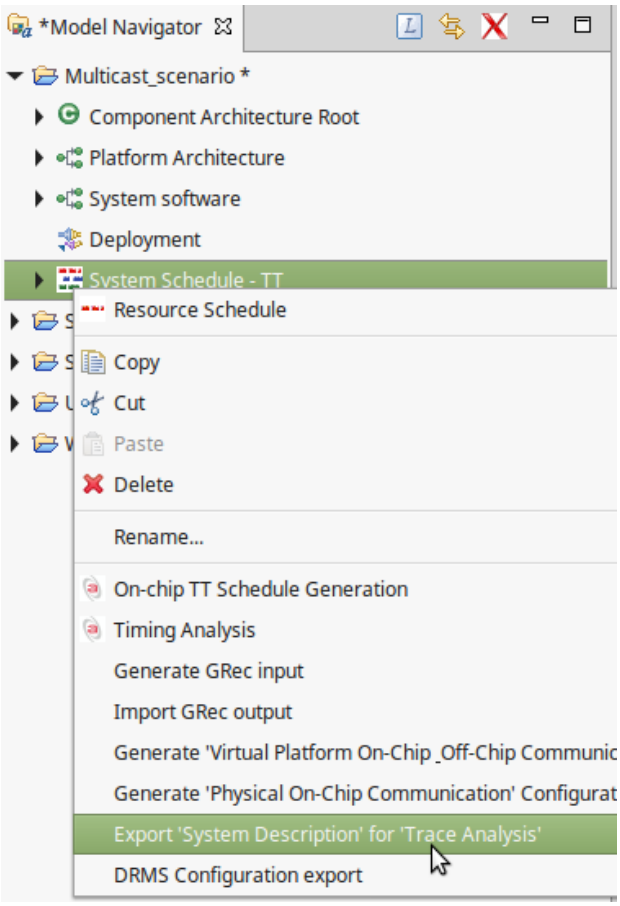

Figure 47: Export of system description for trace analysis.

As a result, an RTaW-Timing window pops up, showing the exported system description. From this point on, you just need follow the instruction on how to import the trace files, provided in D5.2.2.

7.3 Visualization of Trace Analysis Results

In this section we describe the views that have been added to the RTaW-Timing tool for visualizing frame drop statistics.

7.3.1 Frame Drop Tables

The “FrameDrops” tab of the topology panes of the on-chip and the off-chip network displays the frame drop counters:

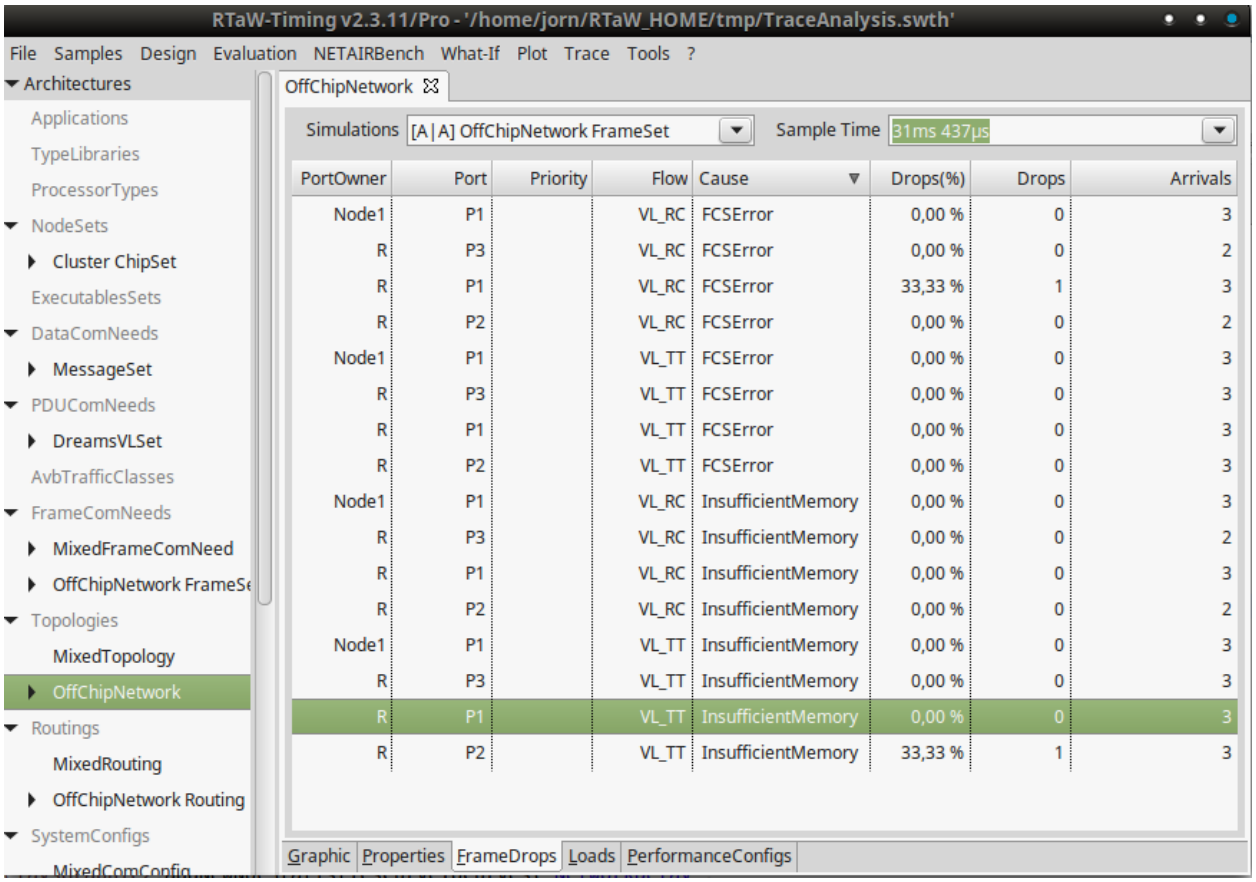

Figure 48: Visualization of frame drop statistics.

If not done so automatically, make sure to select the “simulation” corresponding to the imported trace and a “Sample Time” (= end of the trace).

One line of the table corresponds to a “link” identified by the emitting router or on-chip/off-chip gateway port. The columns display the essential information such as the concerned virtual link and the drop percentage but also the absolute numbers of frames.

Part C
Validation Framework

8 Validation Framework

This section introduces the overall validation framework based on the low-level simulation of the communication between different DREAMS nodes utilizing a mixed-criticality network on the basis of TTEthernet.

The simulation environment helps to evaluate the functionality of the hardware components before their actual implementation in hardware and reduces the need for the setup of an extensive hardware testing environment for various use cases as these tests can be executed in the simulation. Based on the correct behaviour of the simulated components, a more efficient hardware test environment can be created in order to verify the consistency between the simulation and real-world tests.

For the simulation we make use of **Bus Functional Models** for the simulation of the **Design Under Test (DUT)**, not only on the device level but particularly also at the level of the system, in order to evaluate the performance of the device interactions (e.g. multiple end systems connected to one or multiple switches).

The section is consisting of these parts:

- Validation Methodology
- Implementation of the Validation Methodology
- Validation Results

8.1 Validation Methodology

8.1.1 Hardware Testing Strategy

Figure 49 depicts the overall validation strategy as a single process consisting of several steps.

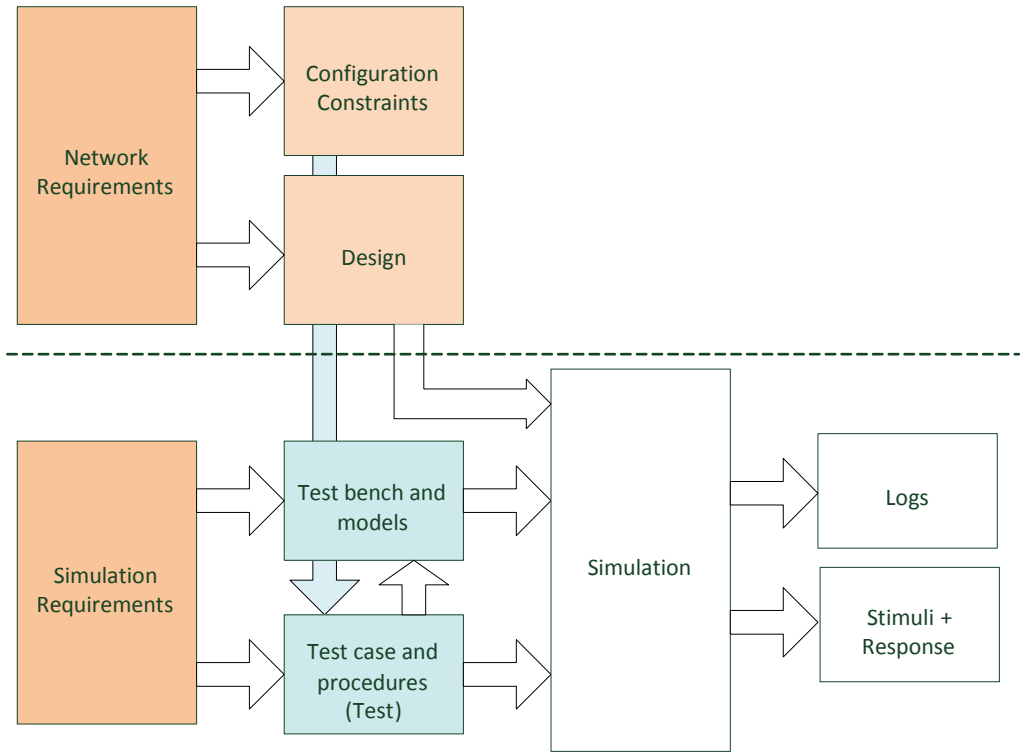

Figure 49: Overall Validation Strategy

A generic test bench, including models of the interfaces, is connected to the DUT (VHDL design of the TTTech switch and two end-system controllers). The models are simulated interfaces, which provide procedures and functions to be used in a test case, thus enabling the test bench to apply stimuli on the DUT. Within the VHDL simulation of the DUT, these stimuli are processed and the output is recorded. The test case itself is checking the output of the simulation (response) against the pass/fail criteria and the result is logged by the simulator (transcripts file). For example, a PCI model implements a ”read” and a ”write” function and converts calls to these functions to PCI bus access signals. A monitor checks if all signals are asserted and de-asserted correctly and checks for a correct response on the bus, if present.

8.1.2 Test Bench Concept

The process of device-level testing is intended to show that the final device is compliant to all defined requirements. This is done by applying requirement-based tests to the DUT. To write more complex and repetitive tests, it is necessary to design a suitable test bench environment. The test bench can be divided into three parts (see Figure 50):

- **Input Generator:** Creates stimulus to the DUT.
- **DUT:** The Device Under Test (IP model of the switch and end system modules).
- **Responses Checker:** Checks for correct DUT responses.

Figure 50: Overview of the basic Verification Concept

The basic concept of the test bench, as described in the previous section, is also used in this implementation. Components that drive stimuli and components that verify the DUT responses are integrated into one component. These components represent the outside connection of the DUT. Commands to the model are coming from procedures of the test.

8.1.3 Modeling Concept

The chosen method of setting up the test bench incorporates the use of bus functional models to apply stimuli and verify functional results from a Device Under Test. It is possible to use models with reduced implementation, i.e., the models do not need to represent the full functionality of components but have to imitate the required signal characteristics. This type of modeling the outside connections to the DUT allows creating more complex functional tests, enables faster runtime of simulations and allows using all the syntax/semantic power of VHDL for test creation. This is depicted in Figure 51.

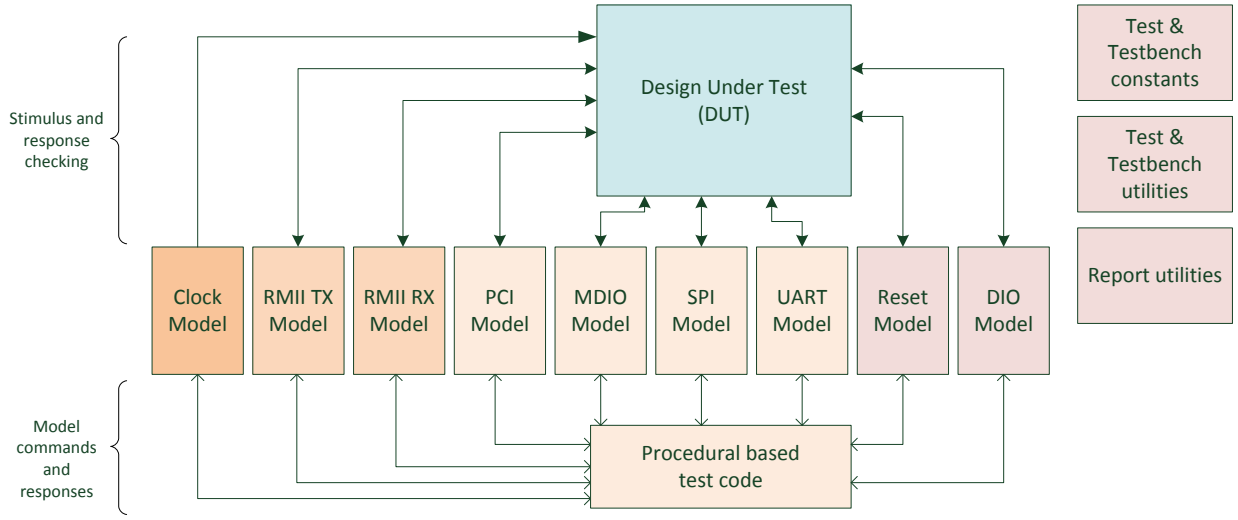

Figure 51: Simulation models and interaction with DUT

There are several models that represent different activities around the DUT. Among them there are:

- **CLK model** Generates the clock.
- **RMII TX model** Writes packets to Ethernet ports.
- **RMII RX model** Monitors the outputs on the Ethernet ports.
- **MDIO model** Interfaces the MDIO serial management interface for the Ethernet transceivers.
- **PCI model** Interfaces the PCI bus.
- **SPI model** Interfaces the SPI/QSPI ports.
- **UART model** Interfaces the UART via RS485.
- **Reset model** Generates resets to the DUT logic.
- **DIO model** Monitors DUT output pins and drives input pins.

Test procedure call methods of the models stimulate and verify the DUT. Moreover the test bench also contains globally accessible packages that contain constants, frequently used procedures (init, reset, ...) and higher level abstraction procedure calls. Several modules used in the ASIC, e.g. Pin Config or Interrupt Controller do not have models, hence they do not have directly associated test cases. Given that their functionality is mandatory for the other modules, their correctness is evaluated by performing the test cases of the other modules. For example, if all test cases pass, Pin Config has to be correct. Otherwise, at least one of the test cases would have failed due to the fact that the execution of all test cases covers the complete pin multiplexing behavior.

8.1.4 CLK model

Clock Generator model: Generates the clock with the specified parameters.

The generics of this model are listed in Table 4 . The ports are listed in Table 5.

Parameter	Default value; Description
G_INST_NAME	**"CLOCK_BFM"**; instance name
G_HANDLE_NO	**0**; natural number that provides the identification of instances
G_PERIOD	**8 ns**; clock period
G_DUTY_CYCLE	**0.5**; clock duty cycle

Table 4: CLK model Generics

Parameter	Default value; Description
[in] tb_stop [out] clk	test bench global stop generated clock

Table 5: CLK model Ports

8.1.5 RESET model

Reset Generator model: Generates reset.

The generics of this model are listed in Table 6. The ports are listed in Table 7.

Parameter	Default value; Description
G_INST_NAME G_HANDLE_NO	**"RESET_BFM"**; instance name **0**; natural number that provides the identification of instances

Table 6: RESET model Generics

Parameter	Default value; Description
[in] tb_stop [in] clk [out] reset	test bench global stop input clock to which reset is synchronized generated reset

Table 7: RESET model Ports

8.1.6 DIO model

DIO model: Drive/read discrete input/output signals.

The generics of this model are listed in Table 8. The ports are listed in Table 9.

Parameter	Default value; Description
G_INST_NAME G_HANDLE_NO	Instance name Natural number that provides the identification of instances

Table 8:DIO model Generics

Parameter	Default value; Description
[in] CLK [in] DIO_IN [out] DIO_OUT [in] TBSTOP	Input clock Pins that are monitored by DIO Pins that are driven by DIO Test bench global stop

Table 9: DIO model Ports

8.1.7 GMII MON model

GMII Monitor GFM: The GMII Monitor model is not directly connected to the DUT's output signals. It is instantiated by the RMII Monitor model to monitor GMII based Ethernet traffic, provided by the RMII Monitor model.

The generics of this model are listed in Table 10. The ports are listed in Table 11.

Parameter	Default value; Description
G_INST_NAME	Instance name
G_HANDLE_NO	natural number that provides the identification of instances

Table 10: GMII MON model Generics

Parameter	Default value; Description
[in] tb_stop	test bench global stop
[in] clk	GMII clock to be used
[in] port_speed	port speed (SP_1000, SP_100, SP_10)
[in] rxd	receive data byte
[in] rxdv	receive data valid (for current byte)
[in] rxerr	receive error (for current byte)
[out] s_statistics	statistics

Table 11: GMII MON model Ports

8.1.8 GMII GEN model

GMII Generator model: The GMII Generator model is not directly connected to the DUT's input signals. It is instantiated by the RMII Generator model to generate GMII based Ethernet traffic, further used by the RMII Generator model.

The generics of this model are listed in Table 12. The ports are listed in Table 13.

Parameter	Default value; Description
G_INST_NAME	instance name
G_HANDLE_NO	natural number that provides the identification of instances

Table 12: GMII GEN model Generics

Parameter	Default value; Description
[in] clk	GMII clock to be used
[in] port_speed	port speed (SP_1000, SP_100, SP_10)
[out] speed	current speed (SP_1000, SP_100, SP_10)
[out] bxm_en	model enabled
[out] txd	transmit data byte
[out] rxen	transmit data enable (for current byte)
[out] txerr_align	insert alignment error
[out] txnibbleodd	insert odd number of preamble nibbles
[out] txerr	transmit error (for current byte)

Table 13: GMII GEN model Ports

8.1.9 RMII MON model

RMII Monitor model: Enables the monitoring of DUT transmitted RMII data. The monitored RMII data stream is converted to GMII and forwarded to a GMII Monitor model, which provides enhanced monitoring capabilities.

The generics of this model are listed in Table 14. The ports are listed in Table 15.

Parameter	Default value; Description
G_INST_NAME	instance name
G_MAC_MODE	**true**; MAC mode (true) / PHY mode (false)
G_HANDLE_NO	natural number that provides the identification of instances

Table 14: RMII MON model Generics

Parameter	Default value; Description
[in] tb_stop	test bench global stop
[in] port_speed	port speed (SP_100 or SP_10)
[in] gmii_clk	GMII local clock
[in] rmii_ref_clk	reference clock used during RMII transfer
[in] rmii_tx_en	data enable signal
[in] rmii_tx_err	data error signal
[in] rmii_txd	data received by the RMII model from the DUT

Table 15: RMII MON model Ports

8.1.10 RMII GEN model

RMII MUX Generator model: Enables Ethernet traffic generation based on RMII. A GMII Generator model is used to generate GMII Ethernet traffic, further converted to a RMII data stream, which is finally fed to the DUT's RMII input pins.

The generics of this model are listed in Table 16. The ports are listed in Table 17.

Parameter	Default value; Description
G_INST_NAME	instance name
G_MAC_MODE	**true**; MAC mode (true) / PHY mode (false)
G_HANDLE_NO	natural number that provides the identification of instances

Table 16: RMII GEN model Generics

Parameter	Default value; Description
[in] tb_stop	test bench global stop
[in] port_speed	port speed (SP_100 or SP_10)
[in] gmii_clk	GMII local clock
[in] rmii_ref_clk	reference clock used during RMII transfer

[in] rmii_tx_en	data enable signal
[in] rmii_tx_err	data error signal
[in] rmii_txd	data received by the RMII model from the DUT

Table 17: RMII GEN model Ports

8.1.11 AHB HOST model

AHB Host model: Models AHM Master Interface

The generics of this model are listed in Table 18. The ports are listed in Table 19.

Parameter	Default value; Description
G_INST_NAME	**"AHB_HOST_BFM"**; Instance name
G_HANDLE_NO	**0**; Natural number that provides the identification of instances
G_AHB_CLK_DIV	**1**; Clock divider

Table 18: AHB HOST model Generics

Parameter	Default value; Description
[in] tb_stop	test bench global stop
[in] clk	system clock
[in] reset	system reset to AHB slave
[out] activate_driver	enables the modelsim's signal_driver to take control over the switch AHB interface
[in] ahbi	AHB slave-master signals
[out] ahbo	AHB master-slave signals

Table 19: AHB HOST model Ports

8.2 Implementation of the Validation Methodology

For the validation of the proposed solution, only the most basic network topology and data flow inside of the DUT is used. This topology and data flow is described in Figure 52.

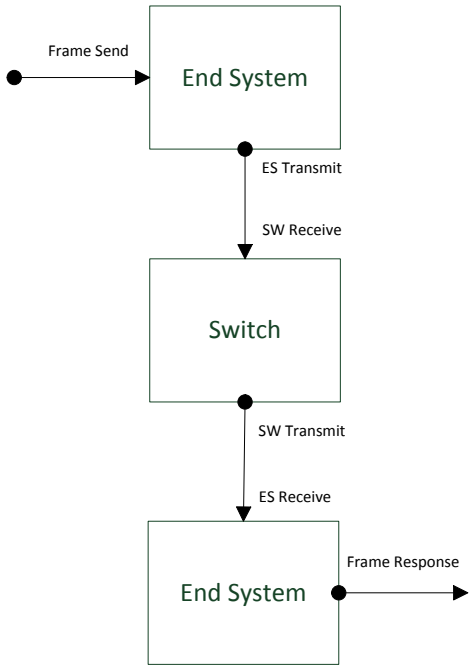

Figure 52 Network topology and data flow in DUT

The network topology consists of two End System modules and one Switch module. The End System modules are identical (copies), which means that validation of the Transmission functionality in ES 0 implies that the Transmission functionality in ES 2 is validated too. The same implication can be also applied to the Receiving functionality in the End System modules. Therefore it is sufficient that the direction of the data flow is only in one direction, from ES 0 to the ES 1. All test procedures are using this network topology and data flow implicitly.

8.2.1 Test Procedure Synchronization

This test procedure validates that synchronization is stable. The period of the fastest VL is between 500 us and 10 ms in order to validate the synchronisation procedure in different environments. The integration cycle duration range tested ranges between 500 us and 10 ms. Transmission speeds range between 100Mbit/s and 1Gb/s.

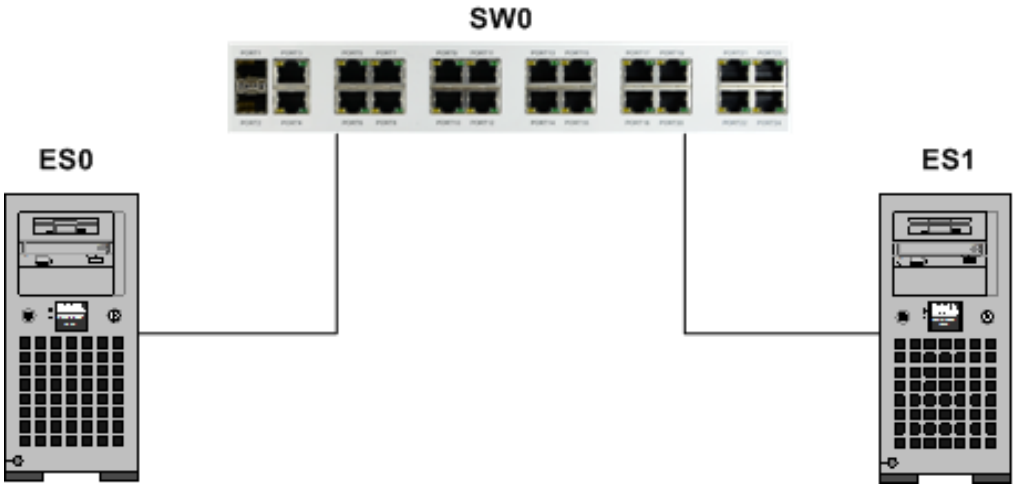

Figure 53: Simulated network topology and data flow for synchronisation validation

The test is passed when synchronisation is stable and the sync-loss counter doesn't increase during the test-runs. The total number of test cases with the mentioned variations is 18 variations.

8.2.2 Test Procedure Freedom from Interference

In this test procedure the latency of TT-VLs on the bus with and without additional RC-traffic and BE-traffic flooding is tested. These TPs also check for missing frames and whether the latency stays within the expected limits (shuffling etc.). The following values are varied: period of fastest TT-VL, period of fastest RC-VL, integration Cycle Duration, window shuffling, switching latency, payload size, transmission speed, and frame-types used.

The test is passed if no frames are lost and the latency is bounded within expected limits. The total number of test cases with the mentioned variations is 25 variations.

8.2.3 Test Procedure Security

8.2.3.1 TP_MACSEC_BE_1

This test procedure validates that the best effort (BE) frames are being processed correctly if the same symmetric keys are used. This is done by sending one BE frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send).

8.2.3.2 TP_MACSEC_BE_2

This test procedure validates that the best effort (BE) frames are being processed incorrectly if different symmetric keys are used. This is done by sending one BE frame to the data flow and comparing the original data with the output data of the End System 2. The expected behavior is that either the received data are changed (different) or the data are not received at all (dropped in the Switch module).

8.2.3.3 TP_MACSEC_BE_3

This test procedure validates that the best effort (BE) frames are being processed correctly if the same symmetric keys are used for the communication between the End System 1 and Switch modules, however the communication between the Switch and End System 2 modules is not using the MACsec ports. This is done by sending one BE frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send). This test procedure ensures that the network communication can be also partially secured as not all ports will probably support the MACsec functionality.

8.2.3.4 TP_MACSEC_TT_1

This test procedure validates that the time-triggered frames are being processed correctly if the same symmetric keys are used. This is done by sending one TT frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send).

8.2.3.5 TP_MACSEC_TT_2

This test procedure validates that the time-triggered frames are being processed incorrectly if different symmetric keys are used. This is done by sending one frame to the data flow and comparing the original data with the output data of the End System 2. The expected behavior is that either the received data are changed (different) or the data are not received at all (dropped in the Switch module).

8.2.3.6 TP_MACSEC_TT _3

This test procedure validates that the time-triggered frames are being processed correctly if the same symmetric keys are used for the communication between the End System 1 and Switch modules, however the communication between the Switch and End System 2 modules is not using the MACsec ports. This is done by sending one frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send).

8.2.3.7 TP_MACSEC_PCF_1

This test procedure validates that the protocol control frame (PCF) frames are being processed correctly if the same symmetric keys are used. This is done by sending one PCF frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send).

8.2.3.8 TP_MACSEC_PCF_2

This test procedure validates that the protocol control frame (PCF) frames are being processed incorrectly if different symmetric keys are used. This is done by sending one PCF frame to the data flow and comparing the original data with the output data of the End System 2. The expected behavior is that either the received data are changed (different) or the data are not received at all (dropped in the Switch module).

8.2.3.9 TP_MACSEC_PCF_3

This test procedure validates that the protocol control frame (PCF) frames are being processed correctly if the same symmetric keys are used for the communication between the End System 1 and Switch modules, however the communication between the Switch and End System 2 modules is not using the MACsec ports. This is done by sending one PCF frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send). This test procedure ensures that the network communication can be also partially secured as not all ports will probably support the MACsec functionality.

8.2.3.10 TP_NOMACSEC_BE_1

This test procedure validates that the best effort (BE) frames are being processed correctly by using standard (NOT MACsec) frames for MACsec ports. This is done by sending one BE frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send).

8.2.3.11 TP_NOMACSEC_PCF_1

This test procedure validates that the protocol control frame (PCF) frames are being processed correctly by using standard (NOT MACsec) frames for MACsec ports. This is done by sending one PCF frame to the data flow and comparing the original data with the output data of the End System 2. The frame data at the output of the End System 2 (Frame Response) are expected to be the same as the data that were put to input of the End System 1 (Frame Send).