Distributed Real-time Architecture for Mixed Criticality Systems

Specification of Simulation Framework
D 5.2.1

Project Acronym	DREAMS	**Grant Agreement Number**		FP7-ICT-2013.3.4-610640	
Document Version	1.0	**Date**	31.03.2015	**Deliverable No.**	5.2.1
Contact Person	Mohammed Abuteir	**Organisation**		USIEGEN	
Phone	+49 271 740 2546	**E-Mail**		mohammed.abuteir@uni-siegen.de	

Contributors

Name	Partner
Mohammed Abuteir	USIEGEN
Roman Obermaisser	USIEGEN
Zaher Owda	USIEGEN
Hamidreza Ahmadian	USIEGEN
Jörn Migge	RTAW
Lionel HAVET	RTAW
Javier Coronel	FENTISS
Marcello Coppola	ST
Arjan Geven	TTT
Grammatikakis Miltos	TEI
Polydoros Petrakis	TEI

Table of Contents

Table of Figures

Abstract

This deliverable presents the specification of simulation framework, which was defined based on the DREAMS architectural style.

The simulation framework includes cluster level, chip level and execution level. Each simulation level has its simulation tool. Therefore, the integration of the different simulation tools is presented in this deliverable. This integration of the different simulation tools requires the presence of communication and coordination interfaces. Furthermore, the coordination of different simulation tools requires the synchronization of the simulation steps since the simulation step on one simulation tool may depend on the result of the other simulation tool.

Additionally, the simulation building blocks in different level may need for the configuration parameters. Therefore, the tool for generation the configuration parameters is required. Moreover, a tool for analyzing the simulation result is necessary.

Additionally the formal verification framework for DREAMS and the fault injection DREAMS emulation platform is presented by this deliverable.

1 Introduction

This deliverable describes the simulation architecture of the DREAMS virtual platform and the dependencies between the simulation building blocks. The DREAMS virtual platform consists of different simulation levels: cluster-level simulation, chip-level simulation and execution environment simulation. The implementation of these different levels is based on different simulation tools. Therefore, integration and communication mechanisms are needed to ensure the efficient transfer of data and control between co-operating simulation tools.

The DREAMS virtual platform will be integrated into the DREAMS toolchain through the model-driven configuration of the simulation blocks. Furthermore, simulation observations are automatically synthesized into analysis results.

Beyond this, a formal verification framework, specifically for protocol verification via a process algebra language for modelling and a model checking tool for verification of temporal logic properties of the STNoC are introduced.

1.1 Relationship to other DREAMS Deliverables

The architectural style document D1.2.1 served as the primary input for the specification of the simulation framework. The simulation building blocks were defined to comply with the architectural style. The specification of the simulation building blocks is the foundation for the subsequent implementation of the DREAMS virtual platform. The interface for the integration of the on-chip simulation building blocks into the virtual platform has been agreed with WP2.

Based on the configuration needs of the simulation building blocks and the DREAMS meta-model (T1.4 and T1.6), T4.2 will develop tools for the model-driven configuration for the simulation building blocks.

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, the 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: This deliverable specifies the simulation and fault injection framework for simulating and verifying the behavior of a mixed-criticality system based on the DREAMS architecture. The deliverable identifies the selected simulation tools and provides specifications of the simulation interfaces between the tools. The deliverable also contains specifications of the simulation components for the architectural building blocks and specifications of the simulation components for the evaluation of timing and reliability.
- D5.2.2 Prototype implementation of simulation framework for DREAMS architecture: The deliverable consists of simulation models for the virtualization building blocks of the DREAMS architecture for the cluster-level and the chip-level. Simulation models at the cluster-level will be based on real-time Ethernet. At the chip-level, a formal verification framework based on process algebra language will support protocol verification and the investigation of safety properties. The simulation models are integrated in an overall simulation framework for the DREAMS architecture. In addition, techniques for the transfer of test case results from the simulation to tests performed on the target hardware are available.

- D5.2.3 Fault injection framework: This deliverable is a fault injection framework extending the simulation framework from D5.2.2. The deliverable includes fault injection components for the injection of operational faults and design faults in the simulation components of the application subsystems and the platform. In addition, simulation components for monitoring significant properties (e.g., reliability) are provided. A trace inspection tool will support the checking of behavioral assumptions made at model level.

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 as follows. Section 2 gives an overview of the structure of the simulation framework, where the following is defined:

- The simulation architecture for different communication levels
- The integration mechanism to link simulations at different integration levels
- The configuration tools, methodology and analysis of simulation traces.
- The formal verification of the STNoC framework
- Transfer of test results from simulation to physical systems.

Sections 3, 4 and 5 describe the simulation building blocks and the simulation tools that will be used as well as the configuration interfaces of other simulation building blocks for the cluster level, chip level and hypervisor level, respectively. To ensure the efficient transfer of data between co-operating simulation tools, the integration and communication mechanisms are presented in section 6. Section 7 explains the principles of the model-driven configuration of simulation blocks and describes how the corresponding configuration file generators will be integrated into the DREAMS model editors provided by WP1. The analysis of simulation traces is presented in section 8. In section 9 the specification of a formal verification framework for DREAMS is provided. A new methodology to verify the robustness against SEUs of a DREAMS Chip instance is described in section 10.

2 Overall Structure of Simulation Framework

One of the main objectives of the DREAMS project is to develop a cross-domain architecture and design tools for networked complex systems where application subsystems of different criticality, executing on networked multi-core chips, are supported. With the increasing complexity of the platforms, testing has become a challenging and costly process.

A simulation environment for the DREAMS platform allows to gain insights into design alternatives and design faults at early development stages, thus decreasing development time and cost. Through the automated execution of test cases in a simulation environment, tests become reusable (e.g., regression tests in different versions of an application), it is possible to perform more tests in less time with fewer resources and testing is more comprehensive and reliable. Figure 1 shows the "Overall Simulation Building blocks".

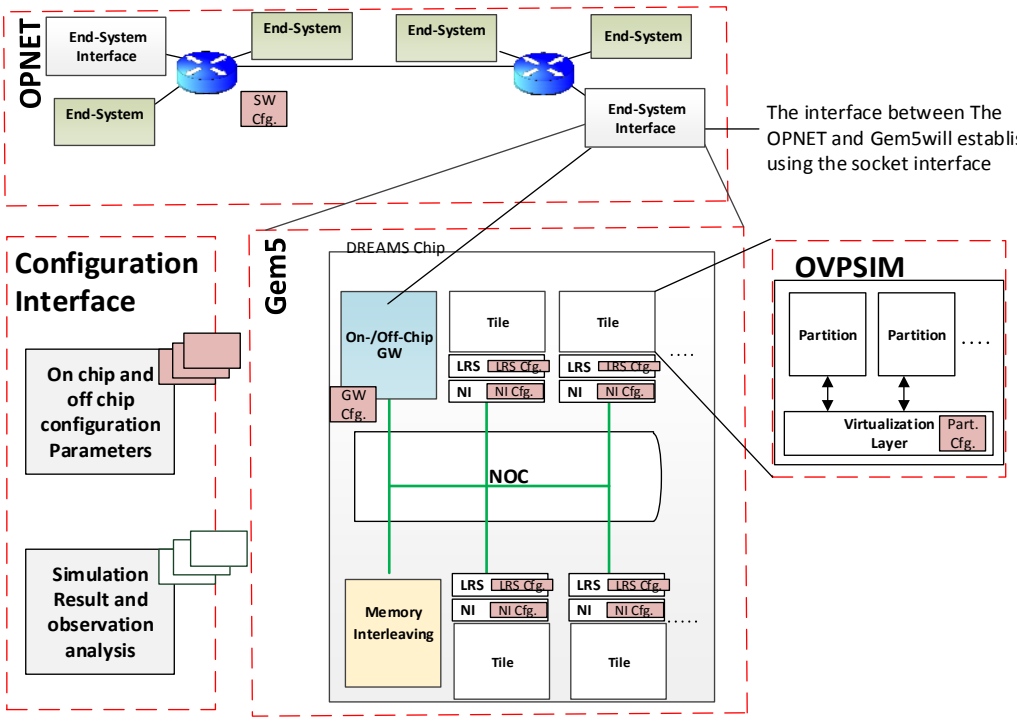

Figure 1: Overview Simulation Building Blocks

2.1 Simulation of Cluster Level

The simulation environment for the cluster level is based on the time triggered Ethernet (TTEthernet). The simulation allows different connection of End-Systems or/and DREAMS chips using one or more off-chip routers in a given topology (e.g., star, ring). The main simulation building blocks at the cluster level are generic building blocks of the infrastructure elements of a TTEthernet-system, which can be configured and extended to create an application-specific simulation model:

- ***Generic model of a TTEthernet switch***. TTEthernet switches are central building blocks of a TTEthernet-based system. A generic simulation model of a TTEthernet switch supporting periodic, sporadic and aperiodic communication will be developed. In order to construct the overall simulation model, the user can perform multiple instantiations of the generic switch, establish connections to end systems, DREAMS chips and other switches and assign to each switch instantiation a corresponding configuration. The switch configuration defines the message timing including a time-triggered communication plan.
- ***Generic model of a TTEthernet end system.*** TTEthernet end systems are the communication end points within the TTEthernet system. The user can perform instantiates of the generic TTEthernet end system and connect each instantiation to TTEthernet switches. End systems can be configured to produce messages according to application-specific parameters (e.g., interarrival time, distributions of sporadic messages, periods of periodic messages). In addition, nodes can be extended with the application behavior (e.g., C++ application code).

More details about the simulation building blocks are provided in section 3.

2.2 Simulation of Chip Level

The chip level simulation environment of DREAMS is composed of the Local Resource Scheduler (LRS) at the on-chip network interface and the STNoC backbone.

The LRS consists of the *Egress Bridging Unit (EBU)*, the *Serialization Unit (SU)* and the *Ingress Bridging Unit (IBU)*. The interface between the cores and the bridging units (i.e., EBU and IBU) is established by means of *Ports* and the interface between the EBU and the SU is provided by the priority queues.

The STNoC is composed of instances of the following STNoC backbone components which can be configured and extended for optimal support of real-time traffic.

- Network Interface (which is known as NoC interface in conjunction with the LRS): The NI is used to interconnect all IP cores (Tiles, Memories, Gateway) to the NoC. Messages arriving at the NI are segmented into packet and flits of a fixed (but configurable) size prior to entering the NoC.
- Router: The router dispatches the flits of a message based on a source-based routing information. These routers retrieve the routing information from the header of the packet (header flit).
- Synchronous Link: The link is used to connect a router to a NI or to specify router-to-router connections.

The gem5 simulator can execute in Full System (FS) mode, including an operating system (e.g. Linux and Android) and hardware devices. System services, shell commands, compiler and debugging tools, are available by connecting to a simulated console via telnet. However, currently there is limited, uniprocessor-only support of Ruby memory system for ARM-based multicore systems, implying the use of external co-simulation tools; only the less accurate and flexible Classic memory system (no instruction pipeline), gem5 can model an ARM uni-or multi-processor running Linux or Android in FS mode. An ongoing effort by TEI aims to connect the Ruby memory controller to the STNoC NI, disabling cache coherent interfaces. This work will be presented along with STNoC and newly proposed access control mechanisms (interleaving, FBA) in DREAMS Deliverable D2.1.3

2.3 Simulation of Execution Environment

The simulation of the execution environment will represent an application tile component within the DREAMS architecture. This environment simulates the execution of DREAMS applications with mixed criticality features. These applications will run on top of the XtratuM hypervisor inside partitions of virtual

machines. The hypervisor and partitions access the services provided by the DREAMS platform and for this, the execution environment must be able to connect to simulation of on-chip and simulation of cluster level. Other simulation environments will simulate those additional DREAMS components such as on-chip interconnect (STNoC), off-chip/on-chip gateways, off-chip interconnect (TTEthernet) and GRM.

The main simulation building blocks in the execution environment are generic building blocks of a DREAMS application tile, which can be summarized as follows:

- Processor core and local memory: The DREAMS harmonized platform is used as the reference platform for the simulation of the execution environment. Therefore, the simulation of the application tile will be based on the Zynq-7000 series board embedded with an ARM Dual-Core Cortex-A9 processor. From that board, the execution environment will only be focused on the Cortex-A9 processor, memory and the majority of internal devices around the processor. This ARM processor is based on the ARMv7-A architecture, which will be fully simulated. Additionally, the ARM processor to be simulated can be setup up with one or more cores, although for the execution environment only two cores will be simulated.
- DREAMS virtualization layer: The XtratuM hypervisor is used as a reference virtualization layer. The behavior of the hypervisor is completely mimicked in the same way as a real execution on a hardware platform. The configuration and tuning of the hypervisor version to be simulated can be performed using the same toolsets provided by XtratuM in the software development. All XtratuM services and features will be available to be used in the simulation and the DREAMS Abstraction Layer and applications can access to them. . The virtualization layer can implement DREAMS services such as MON (Resources Monitors) and LRS (Local Resource Schedulers) – D3.2.1 High-Level Design of Global Resource Management Services. Although, it will depend about the availability during the simulation tasks.
- DREAMS Partitions: partitions or application containers will run on top of the hypervisor. These partitions will implement application functionalities to complete the DREAMS platform simulation. Each partition is based on the Dreams Abstraction Layer (DRAL), and on top of DRAL, DREAMS Application components are executed. These applications can implement DREAMS services that need access to other DREAMS tiles through other simulation environments. DREAMS partitions could contain:
    - DRAL (DREAMS Abstraction Layer)
    - MON (Local Resources Monitors)
    - LRM (Local Resource Managers)
    - Application components
- OVPSIM interface to DREAMS chip: These building blocks interfaces between the OVPSIM simulation and the other simulation environment. This wrapper should be able to receive information from other simulation and injects it in the OVPSIM simulation and also exports information from OVPSIM simulation to the other simulation.

2.4 Linking of Simulations at Different Integration Levels

Multiple simulation tools (i.e., OPNET, GEM5 and OVPSim) are used in DREAMS to simulate the virtual platform. Those simulators have different execution restrictions and capabilities, therefore a linking framework is required to combine and synchronize these different simulation tools. There are two simulation levels: Chip level, in which the non-cycle accurate OVPSim is used in combination with GEM5 to simulate system core execution services in a multicore network-on-chip setup. Secondly, the cluster

level, which simulate the off-chip communication using OPNET and GEM5. Both tools are discrete event simulators.

Taking into account that discrete events occur at specific points in time, such events result in changes of the simulation which are communicated to the simulation coordinator. A socket based coordination control framework for simulation support of multiple tools is presented in section 6. The multi simulation coordination framework consists of the global coordinator, it is responsible for the overall communicate between simulators and control handling. In addition to that, the coordination system has one local controller for each simulator that is responsible for guaranteeing the discrete event execution and synchronization of the simulators.

Two different simulation approaches are used for chip level and cluster level simulations. The interleaving approach presented in subsection 6.1.1 guarantees the synchronization of the OPNET and GEM5 simulations at cluster level by alternating the execution sequence between the simulators. This is done based on the upcoming execution time of the next events in the simulators calendars. Secondly, the tick-based approach allows OVPSim and GEM5 to run in parallel until a pre-calculated time-tick. After reaching this time-tick the two simulators synchronize their local simulation calendars using the buffered events/instructions that were collected in the global controller from the previous execution tick. Later in this section these two approaches are described in detail by describing the technical implementation of the data and control flows.

Functional mock-up interface (FMI) defines a standardized interface that supports model exchange and co-simulation between simulation tools. The FMI standard is widely used in co-simulations that are based on e.g. Simulink, dSpace and AMESim [1]. On the other hand, not all simulation tools support FMI. For instance, OPNET provides its own co-simulation interface. Additionally, Gem5 and OVPSim are yet not used with FMI nor with the OPNET co-simulation interface. Therefore, the proposed multi simulation architecture combines different discrete event simulation tools (e.g. OPNET, GEM5 and OVPSim) that does not have FMI support by presenting a socket-based communication that manages the co-simulation timing model to provide an virtual platform.

2.5 Model-Driven Configuration of Simulation Environment

One objective of the DREAMS project is the model driven support of the design and validation activities of the development process for mixed criticality systems. One motivation for this objective is the possibility to automate the export/import of design data between different tools so as to avoid error prone manual copying, reworking, and pasting of design data, when moving from one tool to another during the design process.

In the context of the simulation framework these ideas translate into the “model-driven configuration” of simulation blocks, i.e. the tool supported configuration of the simulation blocks out of the model of an instance of the DREAMS architecture. Such a model conforms to the DREAMS meta-model and provides the complete description of an instance of the DREAMS architecture.

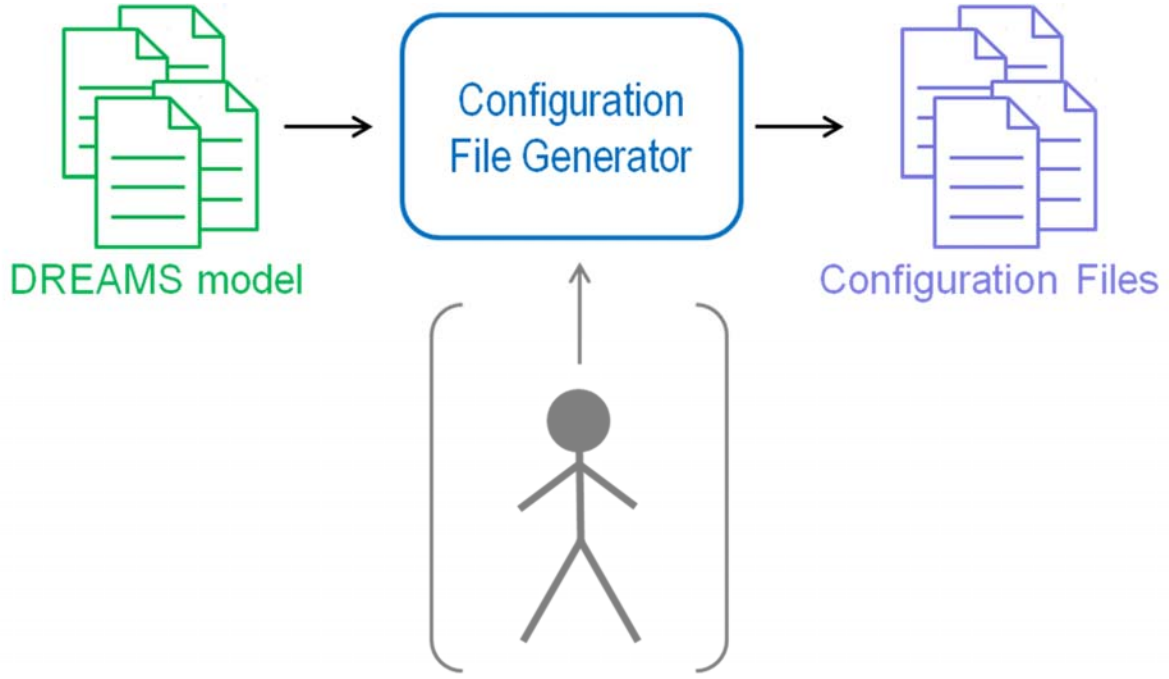

Figure 2: Inputs and outputs of a configuration file generator

Depending on the simulation building block, the Configuration File Generator might use additional inputs in order to bound parameters that are not covered by the DREAMS meta-model.

Details on how the automated configuration is realized are provided in Section 6.

2.6 Analysis of Simulation Traces

The goal of the simulation framework is to help detecting non-conformances of the designed system with respect to the requirements. For this purpose it should produce reports on the observed evolution of properties that are defined by the requirements of the designed system. Typical examples of such properties are (end-to-end) delays, which should conform to latency constraints or the number lost frames in routers, due to insufficient memory.

In discrete event simulators, the evolution of the system model is driven by the occurrences of events that induce state changes. Based on these elementary events (e.g. frame queuing, transmission starts, etc), the high-level properties that need to be reported as a result of the simulation (e.g. end-to-end traversal times) can be deduced, but only if cause and effect relations can be established. All information needed to establish these relations could be put into the traces, but since traces quickly become large files, it is preferable to put only the strict minimum of information into the traces and derive relations as much as possible from the DREAMS system model.

The above described approach leads to the dataflow of the Simulation Trace Analysis Tool as depected in Figure 3. Notice that besides generating the reports, the tool also allows to visualize the results on screen, as tables or graphs.

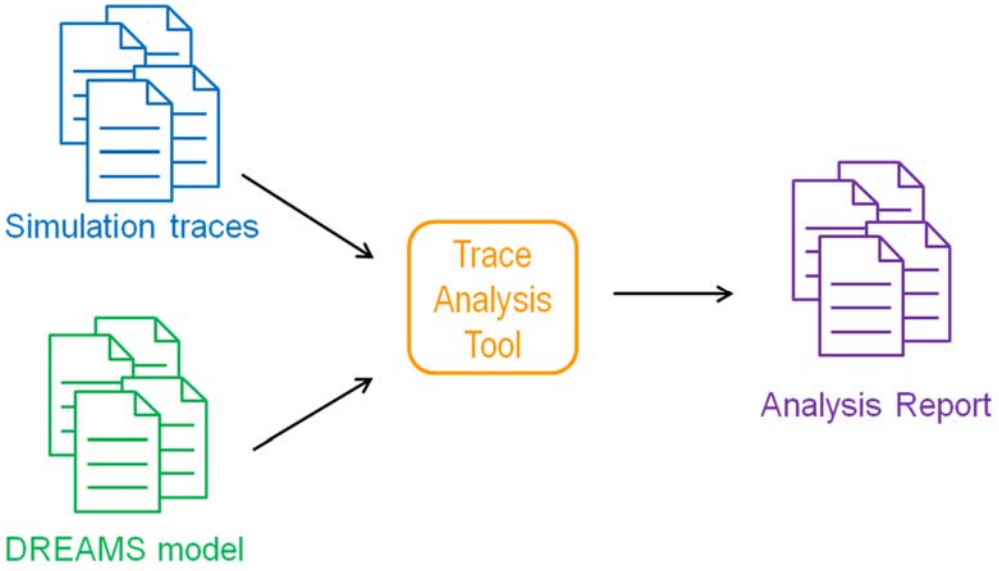

Figure 3: Inputs and outputs of the Simulation Trace Analysis Tool

More details about the formats of the simulation traces and the synthesized properties are provided in Section 8.

2.7 Formal Verification Specification of DREAMS Framework

The introductions of multi-cores architectures leads to an increasing trend to support several applications with different levels of criticality. In addition, handling safely the hardware failures is another design problem that will only increase as multicore will become more and more commonplace.

In this context, formalization is required not only for analysis of safety properties, but also for timing. Formal models play an important role in the analysis of hardware faults and fault correction as well as to know the behavior and timing of conflicts in multicores. In this deliverable, we present the formal verification of the adaptive link of the STNoC which is one of the core elements of the clock synchronization service of STNOC. The adaptive link is located between the STNoC routers: it collects different clocks, and performs a secure crossing of clock regions as described in section 9.

2.8 Transfer test results from simulation to physical system

A specific sub-target of the simulation activities is focused on the simulation based on a very low-level design abstraction in the registration-transfer level (RTL) to perform logic simulation. At this level, typically, FPGA implementation is simulated to predict the behaviour of the logic circuits.

Where model-based test verification activities are focused at finding errors in the toolchain and SW implementation, the low-level simulations models aim to find potential errors in the implementation of protocols and standards in hardware. Particularly in HW/SW co-design, the verification activities of both go hand-in-hand and a simulation of the communication platform, consisting of terminal points (e.g. off-chip gateways) and connection points (e.g. switches) is necessary to provide an end-to-end simulation environment that detects errors that cannot be found in simulation of only the components.

This topic is further addressed in the next stage of the deliverable.

3 Specification of Cluster Level Simulation Environment

At the cluster-level the system is decomposed into DREAMS chips interconnected by a real-time communication network (i.e. TTEthernet) in a corresponding network topology (e.g. mesh, star, redundant star, ring). The connection between off-chip and on-chip networks is established through gateways. A gateway is responsible for the redirection of messages between the NoC and the off-chip communication network.
The specification of the off-chip network is presented in section 3.1.1, where TTEthernet and a time triggered end system are explained. In section 3.1.2, the specification of the gateway is described. Section 3.2 presents the simulation tools that are used to emulate the simulation building block components. The configuration interface for each simulation building block is described in section 3.3.

3.1 Simulation Building Blocks

3.1.1 Off-Chip Network

The simulation building blocks of the off-chip network are listed in the following subsections.

3.1.1.1 TTEthernet switch

As shown in Figure 4, the TTEthernet switch consists of multiple physical links and a bridge. Each physical link contains a physical layer and a MAC layer. The physical layer is built according to IEEE 802.3 [2]. The MAC layer is based on IEEE 802.3 with the following extensions.

The MAC layer checks the validity of the destination address for an incoming frame. This field is extracted from the destination address using the bit mask 0xffffffff0000. If the field has the predefined value of the critical traffic marker, this frame is either periodic or sporadic. Otherwise, the frame is regarded as aperiodic.

The switch distinguishes between periodic and sporadic frames using the EtherType value. The IEEE Standardization Authority has assigned the values 0x88d7 [3] and 0x0800 [4] for EtherType fields of periodic and sporadic frames.

Figure 4 shows the block diagram of the bridge model. The task of the bridge is to handle and forward ingress frames to the egress ports depending on the traffic type. The bridge model contains five layers:

- ***Packet classification layer:*** it puts the ingress frames into one of the internal queues based on the traffic type and virtual link identifier (VLID). In the case of a periodic message, each periodic virtual link (VL) has one periodic VL buffer, which provides buffer space for exactly one message. In case this buffer is full and another message arrives with the same VLID, the newer message replaces the old one. A sporadic message has one queue for each sporadic VL. All aperiodic messages are stored in one queue since aperiodic messages have no timing constraints on successive message instances and no guarantees.

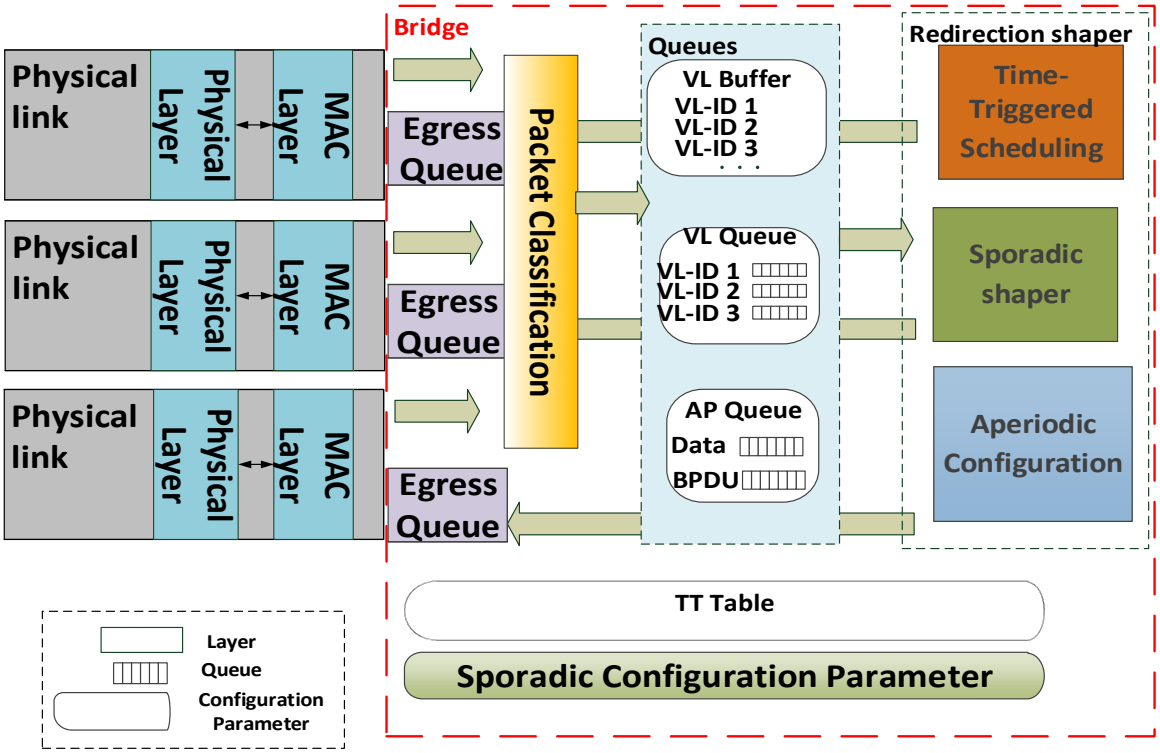

Figure 4: Block Diagram of TTEthernet Switch

- **TT scheduling layer:** It is responsible for relaying periodic frames from the virtual-link buffer to the periodic frame queue at the correct egress port according to the communication schedule. The communication schedule also determines the point in time when the time-triggered frame will be relayed, thereby ensuring deterministic communication behavior.
- **Sporadic shaper layer:** It realizes the traffic policing for the sporadic frame by implementing an algorithm known as token bucket [4]. This layer checks the time interval between consecutive frames on the same VL and moves rate-constrained frames from the virtual-link queue to one of the sporadic egress queues. We distinguish between two priority classes of sporadic frames using two corresponding sporadic egress queues.
- **Aperiodic Configuration layer:** It is responsible for relaying the aperiodic message, where the spanning tree protocol is used to establish a loop-free topology for communication of aperiodic messages [2]. The supported aperiodic messages include Bridge Protocol Data Units (BPDU) and aperiodic data messages. BPDU messages are exchanged between off-chip routers to determine the network topology, e.g., after a topology change has been observed.
- **egress queue layer:** It forwards the frames from the egress queues to the MAC layer according to the priority. The highest priority is assigned to time-triggered frames, whereas aperiodic messages have the lowest priority.

3.1.1.2 End system switch

Figure 5 shows the block diagram of a TTEthernet end system. The end system consists of the following layers:

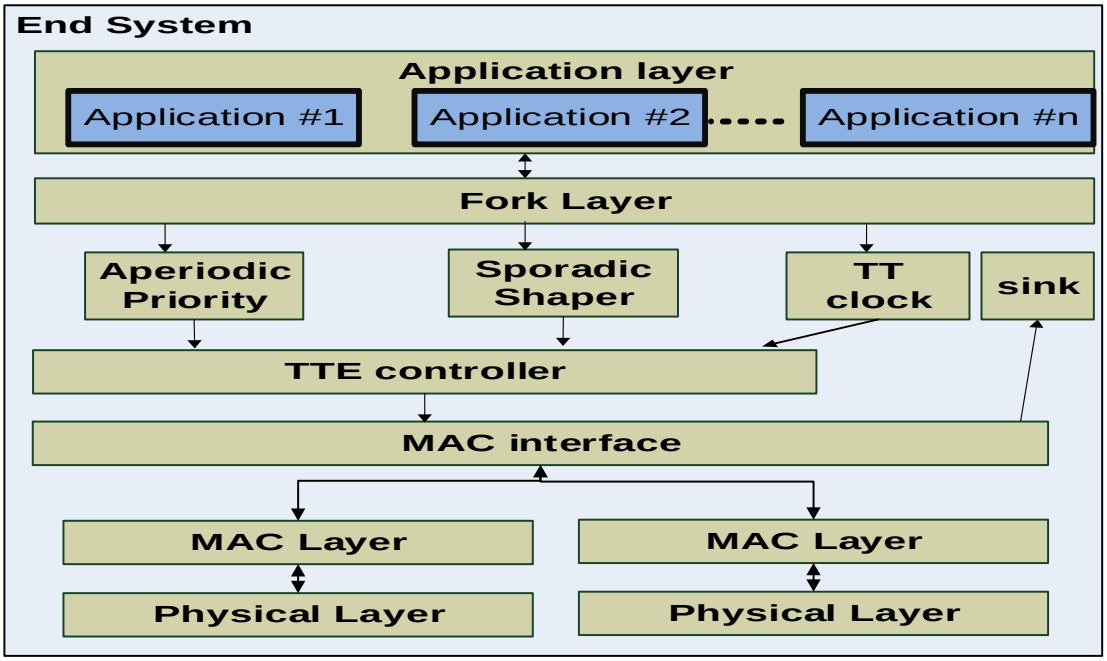

Figure 5: Block Diagram of the TTEthernet End System

- ***application layer:*** it generates the payload traffic with a timing depending on the application specifications (i.e. periodic, sporadic and aperiodic). This payload frame includes *meta data* with payload info such as the source ID, the sender ID and the receiver ID. The meta-data is used to configure the lower layers.
- ***Fork layer:*** it is the interface layer between the technology-independent generic source and the technology-dependent lower layers. According to the configuration parameter in the fork layer, it passes the incoming frame to one of the following layers: TT clock, sporadic shaper or aperiodic Priority.
- ***TT clock:*** it is responsible for the deterministic behaviour of the time-triggered frames. In the TT clock layer, the incoming frame is buffered in a corresponding virtual-link buffer and then the frame is sent to the TTE controller layer at the time specified in the static communication schedule.
- ***Sporadic shaper:*** it is responsible to guarantee the minimum time interval between two consecutive instances of sporadic frames on the respective VL.
- ***Aperiodic priority:*** it forwards the frames to the TTE controller layer according to the priority. There are two priority levels of the aperiodic frames.
- ***TTE controller layer:*** The TTE controller layer contains three queues, namely one for each traffic type. It sends the frames to the lower layer according to their priority. Periodic frames have the highest priority, whereas aperiodic frames are assigned the lowest priority.
- ***MAC interface layer:*** it provides services for the outgoing frames from the upper-layer and the incoming frames from the MAC layer. The frames that come from the MAC layer are directly sent to the sink layer.
- ***Sink layer:*** the received message ends at the sink. The sink is responsible for logging and gathering statistical results.
- ***MAC layer:*** it implemented based on IEEE 803.2.

3.1.2 Gateways

In this section, we will describe the gateway specification model based on the on/off-chip gateway service that is described in the deliverable D1.2.1 [5].

The class diagram for the gateway is illustrated in Figure 7, where all information required for the instantiation of the generic gateway service is available. The constituting classes of the gateway are the bridge, serialization, ingress, egress and VL Queues.

The ingress class will be invoked by an incoming message from the on-chip network or an off-chip network such as TTEthernet. In the ingress class, the incoming messages will be send to the bridge successively.

The bridge class classifies the incoming messages based on the message type (i.e. periodic, sporadic and aperiodic). We explain the processing for each message type in the following subsection.

3.1.2.1 Processing of periodic messages

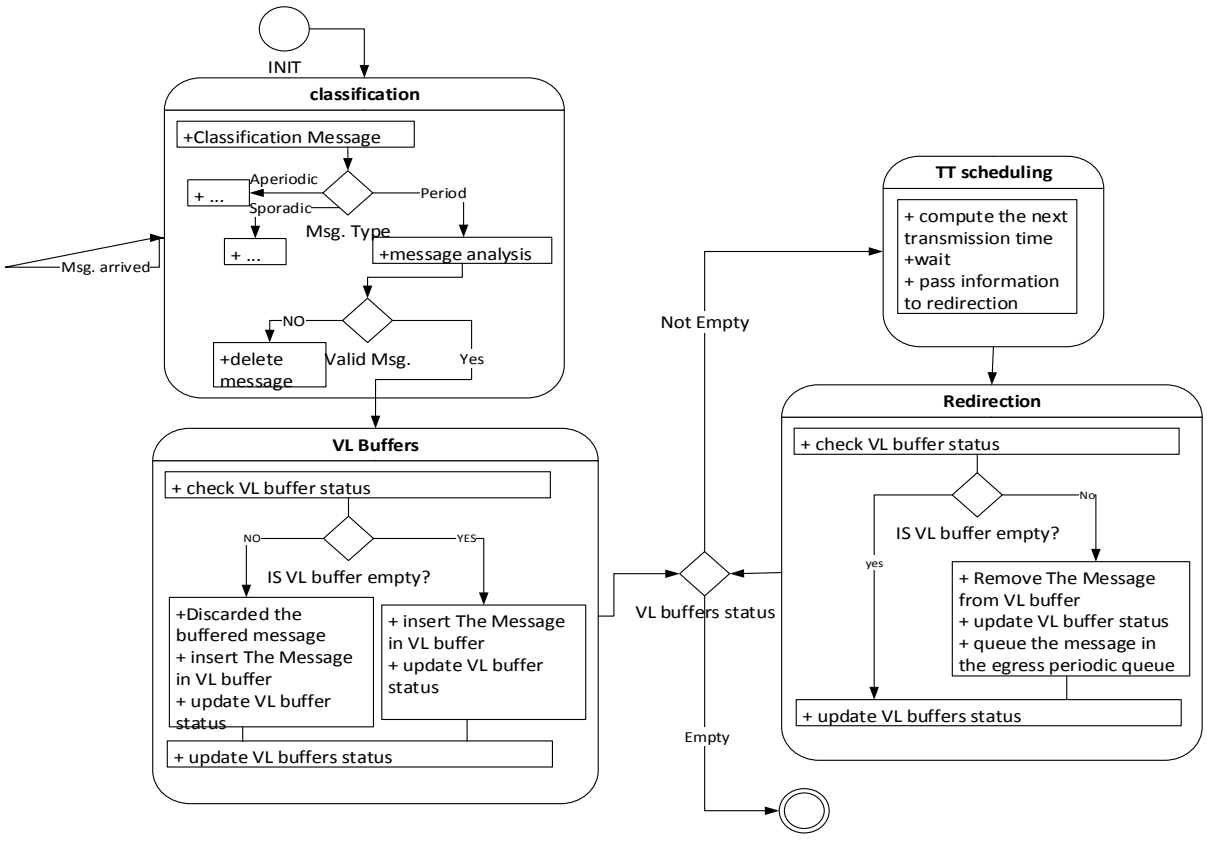

Figure 6: Periodic State Machine

Figure 6 depicts the flowchart for periodic message transmissions. In the classification state, a message analysis function extracts from the periodic message the VLID. In case the incoming message does not have a defined VLID in the configuration parameters, the message is considered as invalid. Invalid messages are dropped in the classification state, while valid messages result in a transition to the VL buffer state.

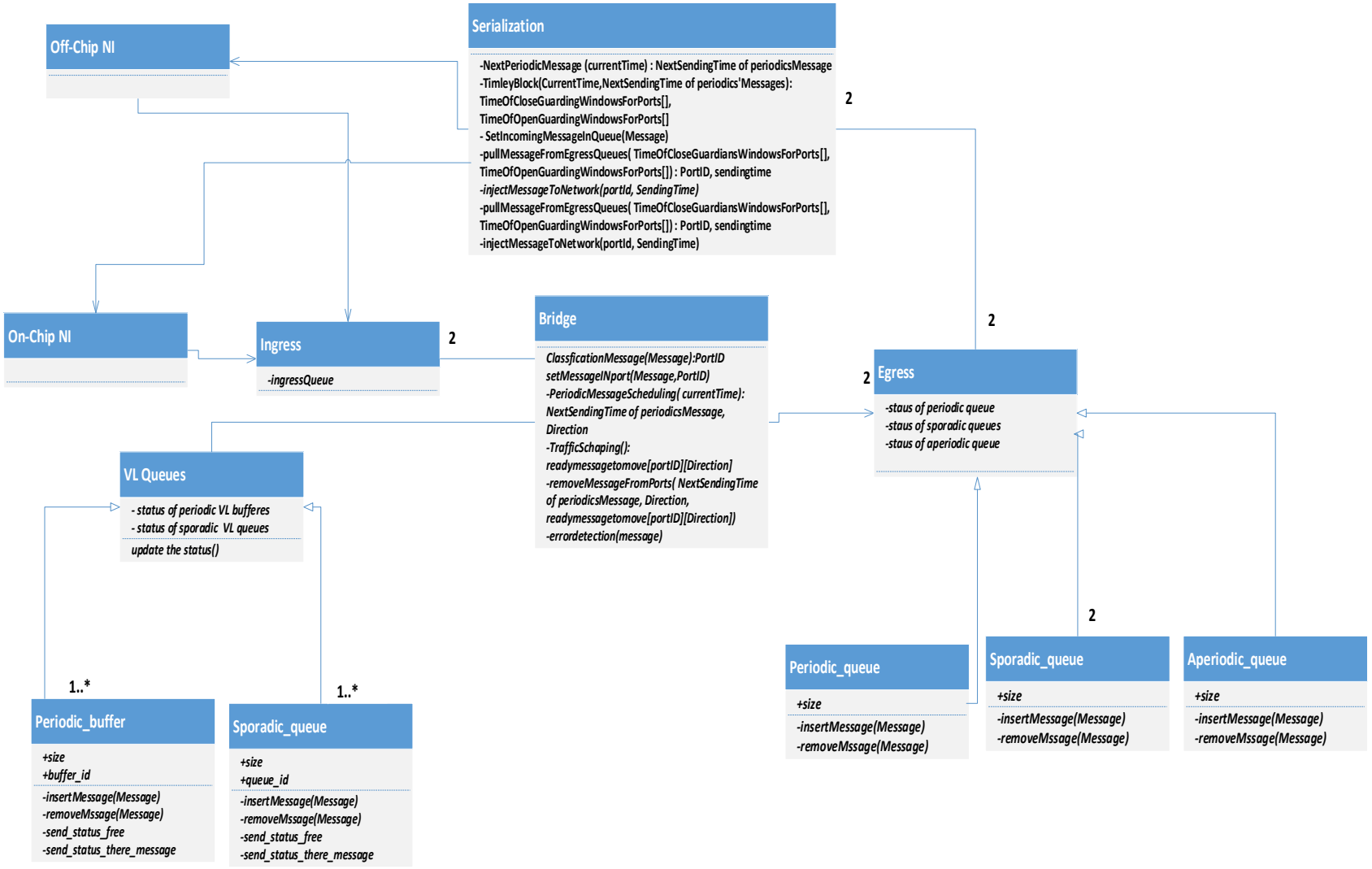

Figure 7: Gateway Class Diagram

Based on VLID, the check VL buffer status function retrieves the buffer identifier from the configuration parameters. Then, it puts the message into the VL buffer, which provides buffer space for exactly one message. In case this buffer is full and another message arrives with the same VLID, the newer message replaces the old one. When the message is buffered, the “VL buffer status” for the corresponding VLID and the “VL buffers status” are updated.

If the “VL buffer status” denotes that the buffer is “not empty”, the next transmission time function in the time-triggered scheduling state determines the point in time when the periodic message is relayed according to communication schedule, thereby ensuring the deterministic communication behavior. At the next transmission time, the *pass information to redirection* function sends the information (i.e., VLID, buffer identifier and direction) to the redirection state. In the redirection state, the *check VL buffer status* function checks whether the corresponding VL buffer contains a message. This message is then sent to one of the egress objects according to the direction parameter, where the message is enqueued in a periodic egress queue. When the message is removed, the “VL buffer status” for the corresponding VLID and the “VL buffers status” are updated. These procedures are performed according to the communication schedule until the “VL buffer status” indicates that the buffer is “empty”.

The serialization is responsible for forwarding the message from the egress queues to the on-chip or off-chip network interface according to the priority. The highest priority is assigned to periodic messages, whereas aperiodic messages have the lowest priority. Using these priorities, the serialization supports two mechanisms to resolve collisions between the different types of messages:

- Timely block: According to the time-triggered schedule, the serialization knows in advance the transmission times of the periodic messages. Timely block means that the gateway reserves so-called guarding windows before every transmission time of a periodic message. The behavior of the timely block mechanism is illustrated in Figure 8.

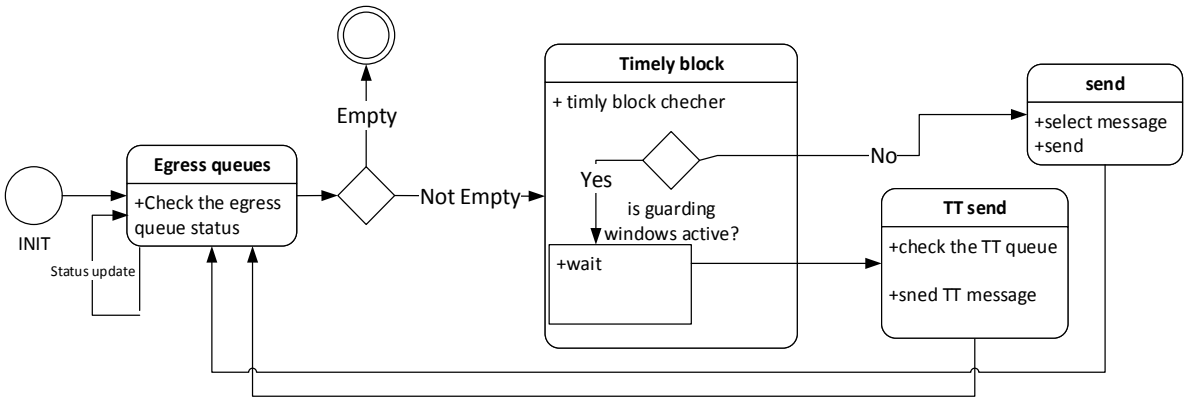

Figure 8: State Machine for Timely Block Mechanism

The egress queues have four egress queues with decreasing priorities: one queue for periodic messages, two queues for sporadic messages (each one for a different priority class) and one queue for aperiodic messages. The egress-queue status is updated when a message is enqueued in one of the egress queues. In case the status of the egress queue is “not empty”, the *timely block checker* function in the timely block state verifies that no guarding windows is active.

In case of guardian windows, the *wait* function imposes a delay until the next transmission time of the periodic message. If there is any periodic message, this message is sent. Otherwise, the process of the flowchart returns to the egress queue state.
In case there is no guarding windows, the *select message* function in the send state selects one message out of the sporadic and aperiodic queues based on the priority and this message is sent. If the status of the egress queues is still "not empty", the procedure is repeated until the egress queues are "empty".

- Shuffling: If a low priority message is being transmitted while a high priority message arrives, the high priority message will wait until the low priority message is finished. Figure 9 shows the flowchart for the shuffling mechanism within the serialization object.

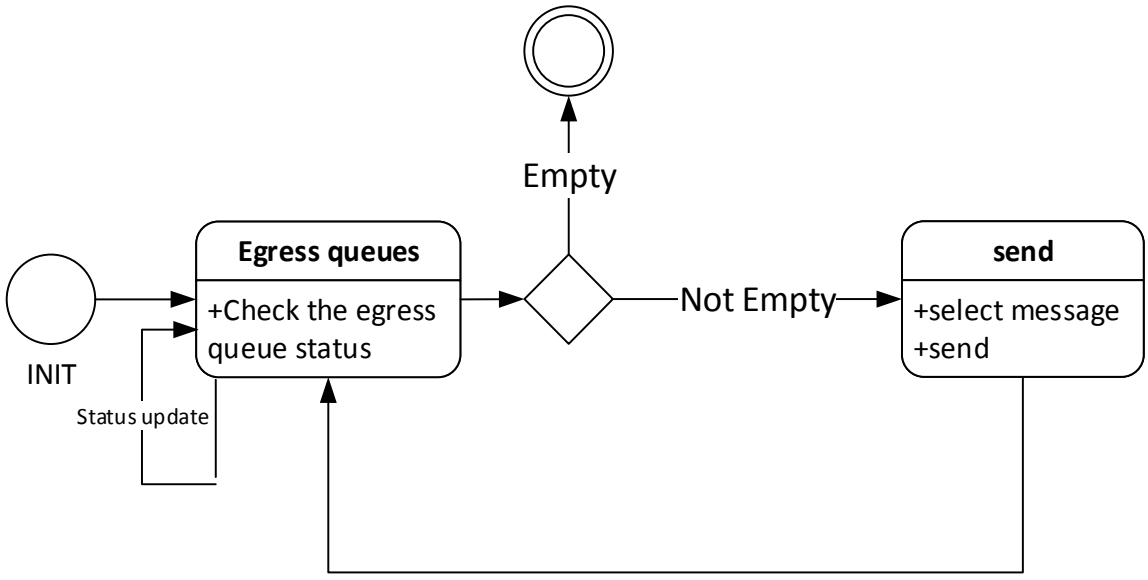

Figure 9: State Machine for Shuffling Mechanism

The egress queue status is updated when a message is enqueued in one of the egress queues. In case the status of the egress queue is "not empty", the *select message* function removes one message from the egress queues based on the priority. The *send* function forwards the message to the network interface of the on-chip or off-chip network interface. If the status of the egress queue is still "not empty", the procedure is repeated until the egress queues are "empty".

3.1.2.2 Processing of Sporadic messages

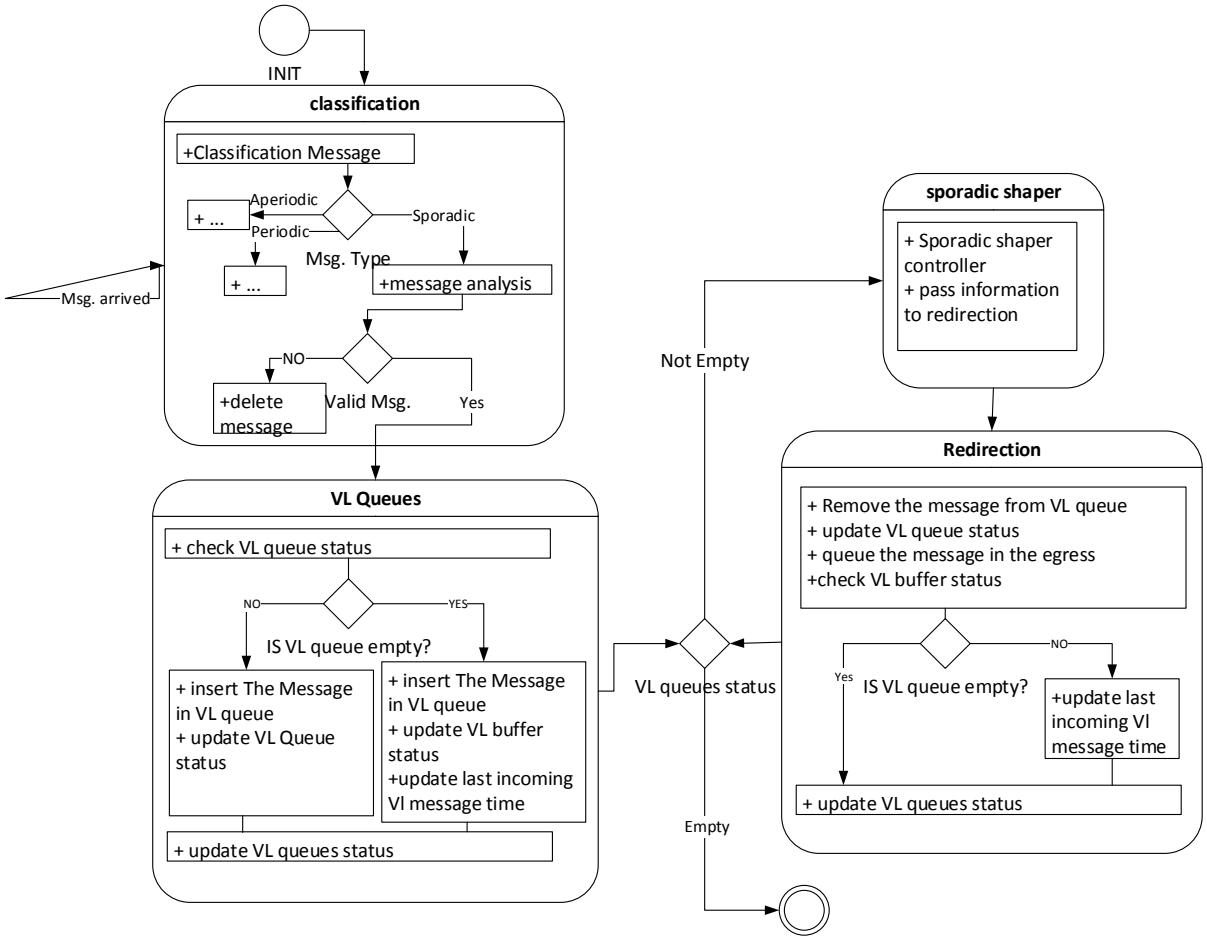

Figure 10: Sporadic State Machine

Figure 10 depicts the flowchart for sporadic message transmissions. The *message analysis* function in the classification state checks an incoming messages. The size of the message must be below the maximum message size according to the configuration parameters of the VL. A valid message is enqueued into the corresponding VL queue. When the message is enqueued, the "VL queue status" for the corresponding VLID and the "VL queues status" are updated. In case the VL queue was empty, the *update time* function updates the reception time of the last incoming VL message. This timestamp is essential for traffic shaping and temporal partitioning.

In the sporadic shaper, the *sporadic shaper controller* function guarantees the minimum interarrival time between two consecutive instances of a sporadic messages on the respective VL. The *sporadic shaper controller* function computes the necessary waiting time for each message based on the time of the latest incoming VL message. When the waiting time has expired, the *redirection* function passes the information (i.e., VLID, buffer identifier and direction) to the redirection state. In the redirection state, the *remove message from VL queue* function forwards the message from the VL queue to one of the sporadic egress queues according to the direction and priority parameters. In case, the VL queue has another message, the time of the last incoming VL message is updated. This step allows the *sporadic shaper controller*

function to send the next message after the minimum interarrival time. This procedure is repeated until the “VL queues status” is “empty”.

Thereafter, the serialization is responsible to forward the message to the network interface of the off-chip or on-chip network as explained in the previous subsection.

3.1.2.3 Processing of aperiodic messages

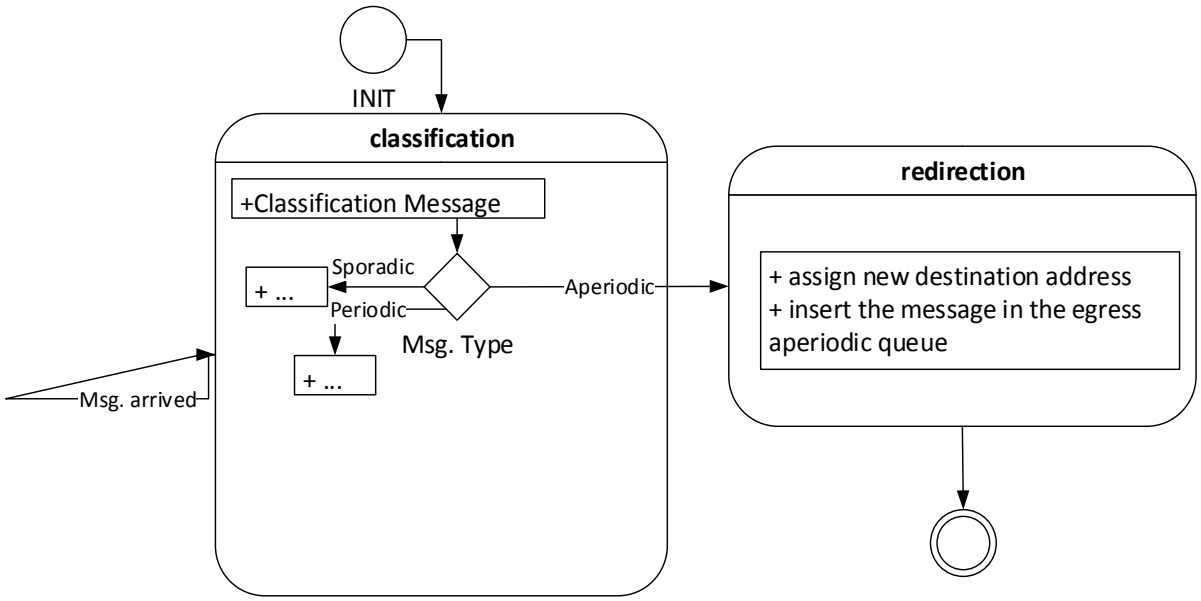

Figure 11: Aperiodic State Machine

Aperiodic messages have no timing constraints on successive message instances and no real-time guarantees. Therefore, the incoming messages are inserted into the corresponding aperiodic egress queue (see Figure 11). When the message is enqueued, the “egress queue status” is updated. Thereafter, the serialization is responsible to forward the message to the network interface of the off-chip or on-chip network.

3.2 Simulation Tool

3.2.1 OPNET

OPNET is a tool suite for discrete event simulations of communication networks. Simulation models in OPNET are organized hierarchically consisting of three main levels: the simulation network, node models and process models. These three levels represent the structure of real-world distributed systems consisting of clusters with networked nodes (e.g., end systems, switches) and different protocol layers.

The top level refers to the complete distributed system and is used to create a network model using building blocks from the standard library and user-defined components. At this level, statistics about the network are collected, the simulation is executed and results are viewed. The node models are at the

second level in the hierarchy and have a modular structure. The node is defined by connecting various modules with packet streams. The connections between modules allow packets and status information to be exchanged between modules. The modules in the nodes are implemented by using process models, the lowest level in the hierarchy. Process models are represented by finite state machines definitions of model functions, and a process interface that defines the parameters for interfacing with other process models and configuration attributes. Finite state machine models are described in embedded C or C++ code blocks. The hierarchical structure of the models, coupled with support for C and C++ code, allows for easy development of communication or networking models.

3.2.2 GEM5

Gem5 is a modular self-contained discrete-event simulation framework resulting from a combination of previous M5 and GEMS simulators which share many years of effort [M5, Gems]. Alike other virtual platforms, such as OVPsim [OVPSIM], Gem5 is free and its BSD License covers most components for Academia and Industry use. The Gem5 infrastructure targets multiprocessor system-on-chip (MPSoC) simulation and design space exploration, reducing the time or cost to apply different configurations in a real system and collect important system performance data. Gem5 uses object-oriented design which provides flexibility and easy refinement through parameterizable, interchangeable and interoperable models. Although there is a steep learning curve due to intricacies of the installation and complexity of the models, users can anticipate to quickly gain experience and design more with less effort.
In this context, Gem5 also provides a built-in, configurable mechanism for system monitoring and design space exploration. Its discrete-event simulation kernel provides high performance and relatively good accuracy using Instruction Set Simulation (ISS).

3.3 Configuration Interfaces

3.3.1 Configuration Parameters for TTEthernet Switch

The configuration of a time triggered Ethernet switch includes parameters for each periodic and sporadic VL that traverse the switch:

- Parameters for a periodic VL: time window for periodic message (message will be dropped if it arrives outside of the window), the identifier, the timing parameters (i.e., period, phase), the sender port, receiver ports, message size and buffer identification.
- Parameters for a sporadic virtual link: the identifier, the minimum interval time, jitter, sender port, receiver ports, message size and queue identification.

3.3.2 Configuration Parameters for TTEthernet end system

The configuration of a TTEthernet end-system consists of the following parameters:

- VLID: The virtual link identifier is used in the periodic and sporadic messages.
- Destination_ID: The destination identifier is used in aperiodic messages.
- Type: periodic, sporadic or aperiodic.
- Periodic timing parameters: These parameters include period, phase, starting time of the application and the end time of the application.
- Sporadic timing parameters: These parameters include interarrival time, the minimum and maximum interarrival time, start time of the application and the end time of the application.
- Aperiodic timing parameters: These parameters include the start and end time of the application.
- Message size: The message size of periodic messages.
- Maximum message size: The maximum message size of the sporadic and aperiodic messages.
- Minimum message size: The minimum message size of the sporadic and aperiodic messages.
- TT clock queue ID: the periodic VL queue identifier
- Sporadic shaper queue ID: the sporadic VL queue identifier

3.3.3 Configuration Parameters for on-/off-chip Gateway

The configuration of an on-/off-chip gateway consist of the following parameters:

- VLID: used in case of periodic and sporadic messages
- DIR: the message direction (an input port provides data from the off-chip network, an output port provides messages to the off-chip network)
- Type: periodic, sporadic or aperiodic
- Timing parameters: In case of periodic messages, the parameters include the period and phase. For sporadic messages, the interarrival time and the jitter are specified. In case of aperiodic message, no timing parameters are required.
- Priority: the priority is used only for sporadic messages
- Message size: the maximum message size for the periodic and sporadic messages
- VL Queue ID: The periodic and sporadic messages need unique VL queue identifiers.

4 Specification of Chip Level Simulation Environment

The chip level simulation environment of DREAMS is composed of the Local Resource Scheduler (LRS) at the on-chip network interface and the STNoC backbone, both implemented using Gem5 simulator system[1]. The STNoC backbone realizes the Network-on-chip (NoC) which doesn't support the time-triggered behavior. The LRS has been defined as an additional layer on top of the simulated STNoC network interface in order to provide the support for the time-triggered transmission of the message as well as different criticalities.

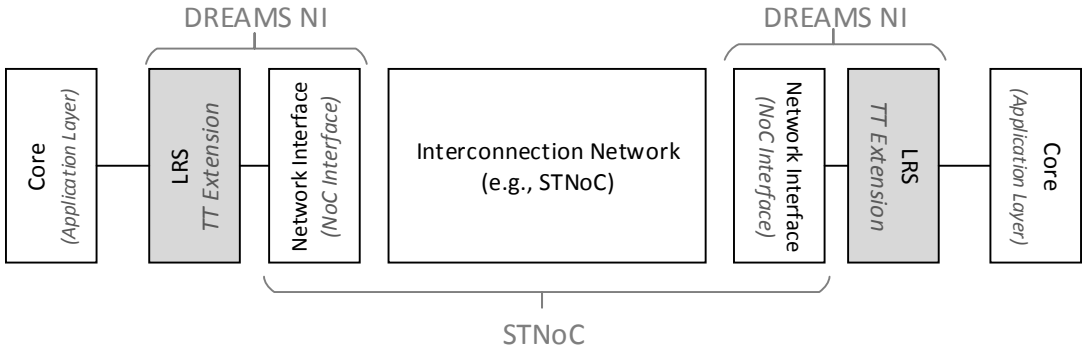

Figure 12 DREAMS Chip level Simulation environment

This chapter is dedicated to chip level simulation environment and is composed of three sections. The first section describes the simulation model of the LRS by introducing basic concepts of gem5, describing the modifications applied to the native gem5 classes and the configuration parameters for the simulation model. This section will end with the simulation result of a sample scenario. The second section describes the STNoC backbone and the configuration interface of the model. The third section shows how the LRS simulation model is integrated with the STNoC model technically.

4.1 The LRS at the on-chip NI

This section describes the simulation model of the LRS at the on-chip NI. It provides early insight for a designer before going to the physical platform, thus acts as the basis for RTL implementation in WP2. Moreover, it aids in analyzing and evaluating the generated offline schedules as well as verifying the effect of modifications in the hardware and the schedule by Local Resource Managers. The integrated simulation of the LRS and garnet exhibits the network on-chip and acts as a building block in WP5.

[1] http://gem5.org

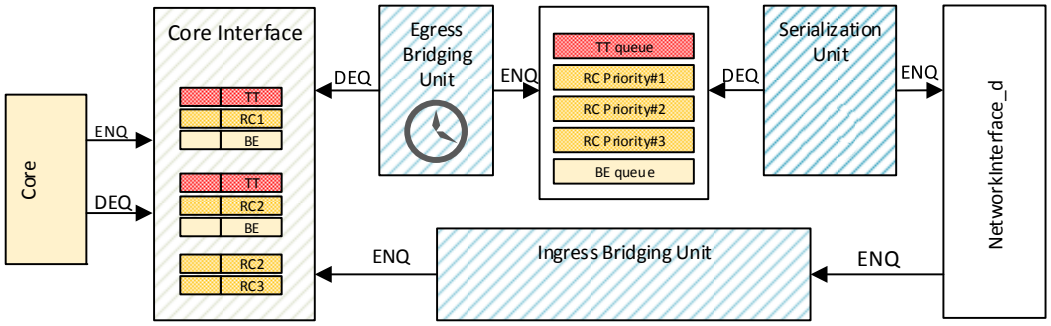

Figure 13 The simulation building blocks of the LRS at the on-chip NI

As described in D1.2.1 the LRS performs the runtime scheduling of resource requests such as allocating bandwidth or processing of queued messages. The LRS acts as an additional layer on top of the plain network interface (for example garnet NetworkInterface_d or shell and kernel of the STNoC) to bridge the gap for supporting mixed-criticality and different communication paradigms, i.e., time-triggered, rate-constrained and best-effort. Figure 13 illustrates different building blocks of the LRS and the interconnection of them. More detailed information about the architecture and specification of the LRS can be found in D1.2.1 Part II, sec 1 and D2.1.2, section 2.

4.1.1 Basic concepts of the LRS simulation model

The proposed simulation model has been embedded into the open-source interconnection network as a part of Ruby[2]. The Ruby Network Tester provides a rich framework for simulating the interconnection network with controlled inputs. In order to make the provided system capable of time-triggered actions, some modification have been applied to some abstract classes. Moreover, several classes have been developed and added to the garnet network. These modifications are described below.

4.1.1.1 Time basis in Gem5

In order to realize the time-triggered communication, a mechanism for timely execution of the actions based on the schedule is needed. An implementation based on the real-time system, however, is not possible because the simulation time does not necessarily run consistently with the system time.

In Gem5 the global time base, or the global logical time is based on *ticks*. A tick has no fixed relation to the real time. Constant defined in `src/sim/core.hh` relate ticks to real time.

Based on the initial configuration of Gem5, the relationship of 1 x 10^{12} ticks per second is defined. This means 1 tick = 1 ps (picosecond). However, this fixed relationship can be lost by a high load on the host system. [3]

In addition to the ticks, Gem5 defines *Cycles*, which in turn are a configurable number of ticks, i.e., 1 $Cylce = n * Tick$. All events and activities in Gem5 are executed on the basis of ticks or Cycles.

4.1.1.2 Events

An event is represented by the Event class (shown in Figure 14), which is an abstract class. This means, the event class can be only inherited by other classes in order to provide the needed methods and attributes needed for any event.

[2] http://www.m5sim.org/Ruby_Network_Test

[3] This relationship can be justified by respective python file (`/src/python/m5/ticks.py`)

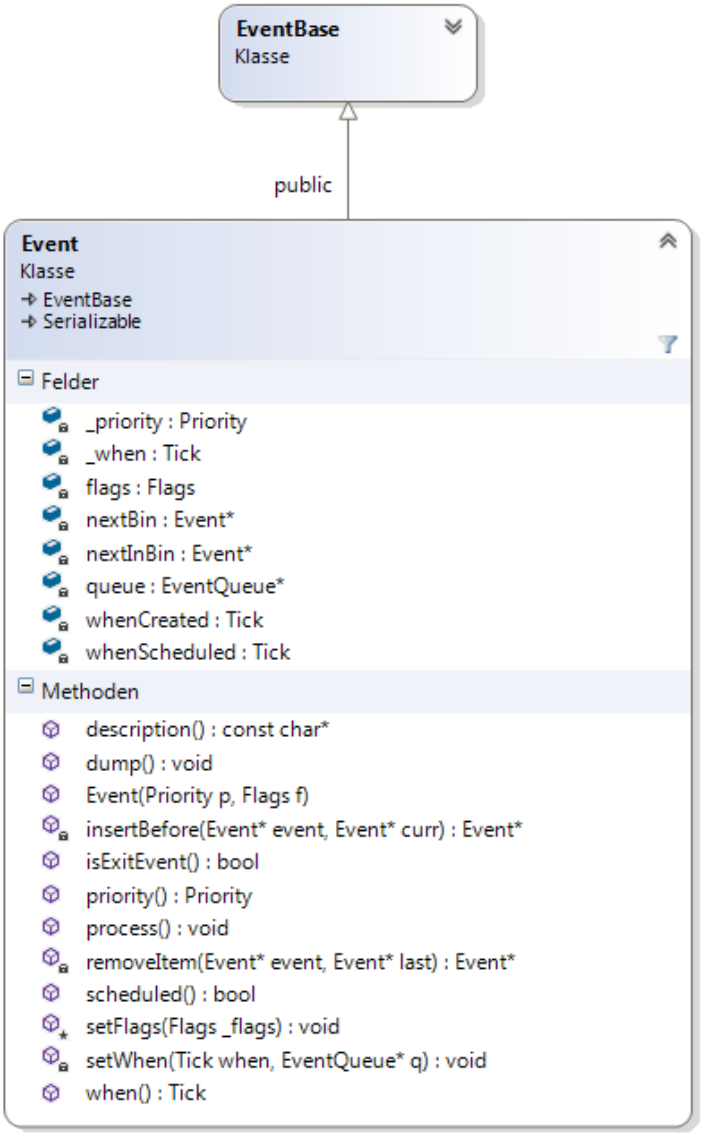

Figure 14 Event class diagram

4.1.1.3 Event management

Scheduling of the events is one of the most common and important issues in an event driven simulator. This is vital in our application, since there are plenty of time-triggered events which need to be scheduled. This has been solved using particular abstract classes. For TT activities, which need scheduled wakeup, the `consumer` class has been altered and employed. This class introduces a virtual `ttwakeup()` method which can be scheduled in the constructor and will be called at the defined schedule. `EventWrapper` would be the best approach for those of TT events, which don't need a recursive wakeup, but rather a single occurrence. These two approaches have been introduced below.

4.1.1.3.1 Consumer

Each event in the simulation is followed by a new decision and possibly a change to a system's state. This has been implemented in Gem5 using `wakeup()` method of the *Consumer* class. The consumer

class (cf. Figure 17) is an abstract class and multiple classes inherit from this class. Moreover, the `wakeup()` method is a pure virtual method, meaning, the respective derived class needs to define the `wakeup()` class locally.

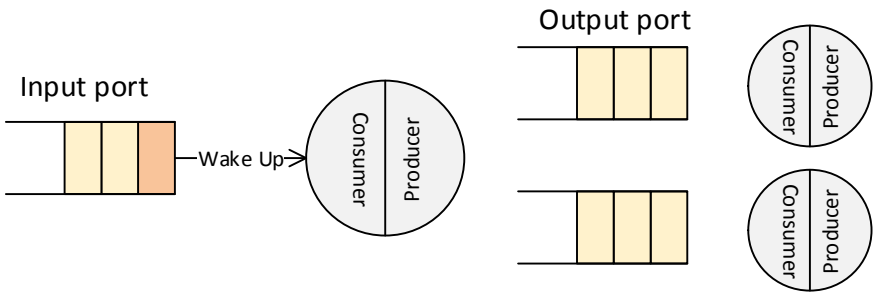

Figure 15 Consumer and events in Gem5

As shown in Figure 15 each consumer class is connected to at least one input (as consumer) and one output port (as producer). Once a new message is enqueued into the input port, the producer will call the `wakeup()` method of the consumer class. When the `wakeup()` is called, the derived class can determine the event type based on the current state of the system and message and decide the next system state. For example, in the case shown above, the consumer on the left hand side can be the EBU which wakes up by the incoming message from the core. After the EBU examined the `portStatus`, based on the port type it can enqueue the message into the respective priority queue.

4.1.1.3.2 Modified Consumer class

Based on the requirements of DREAMS minor modifications (i.e., a time-triggered wakeup (`ttwakeup`)) have been applied to this class. For this purpose a new `ConsumerTTEvent` has been defined in order realize a TT event in `consumer` class. This class duplicates the normal event, but in contrast to the ET events, on one hand, it calls the `ttwakeup()` method and on the other hand the defined `event` has highest priority in the priority queue.

The `ttwakeup()` wakes the derived class up at the scheduled tick in order to perform the planned activity (e.g., enqueue, dequeue, etc.). In case this method is required for the next period, it can reschedule itself at each cycle for the next period (cf. Figure 16).

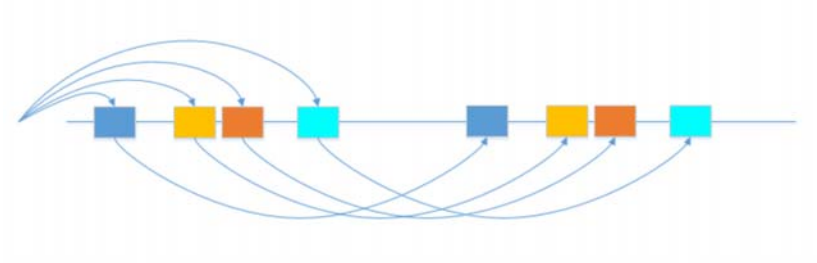

Figure 16 proposed mechanism for infinite wakeup schedule

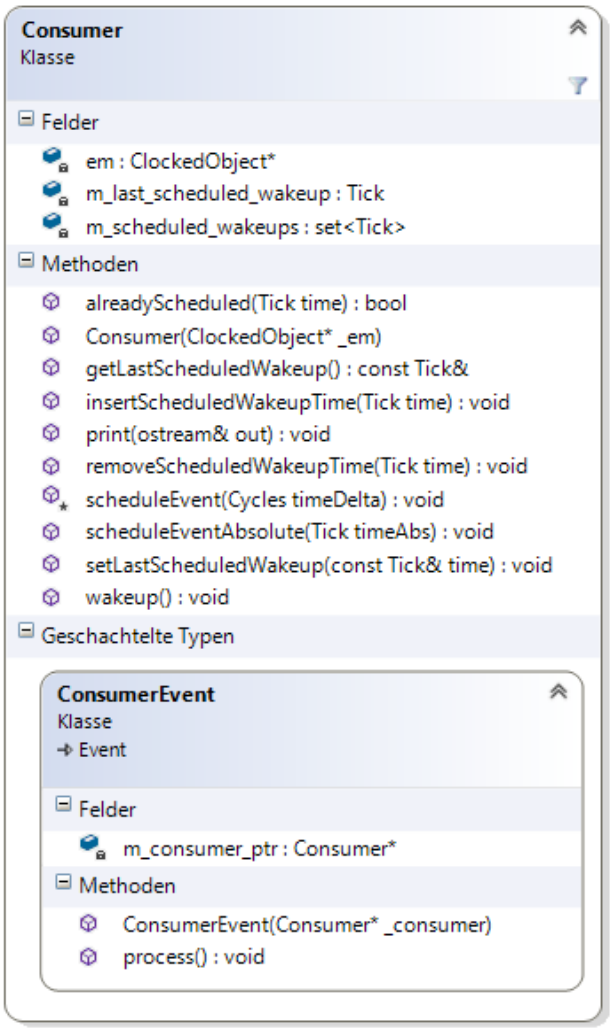

Figure 17 Consumer class diagram

4.1.1.4 Clocked object

The `ClockedObject` class enables us to benefit from the `EventWrapper`. The `EventWrapper` joints a method within the derived class with an event and enables us to define the tick at which the method need to be invoked. This approach provides us more flexibility for scheduling individual events. Code 1 represents, the way that `EventWrapper` can be employed.

```
class Klasse : public SimObject {
      void event_method;
      EventWrapper<Klasse, &Klasse::event_method> myEvent;
};
```

Code 1 EventWrapper declaration

Figure 18 represents an overview of the simulation model and how different building blocks of the simulation model inherited from these two abstract classes. Figure 19 depicts this inheritance of the `consumer` and `clockedobject` class in another form.

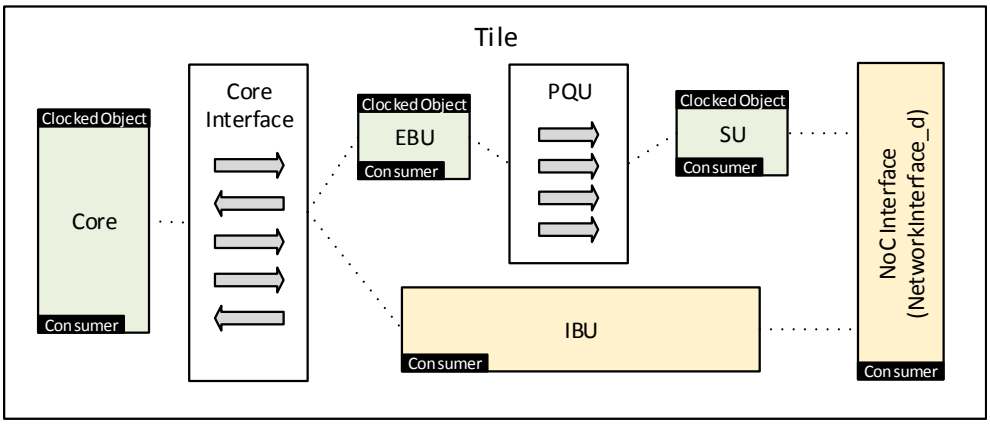

Figure 18 LRS Simulation building blocks

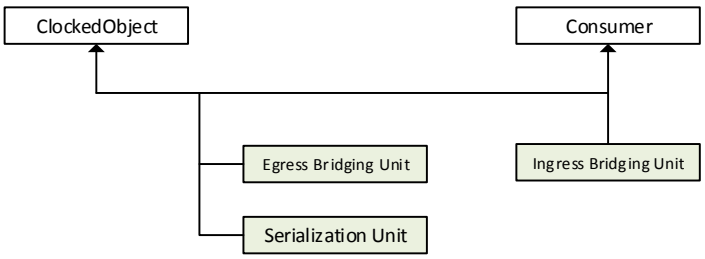

Figure 19 inheritance of the EBU, IBU and the SU

4.1.1.5 Messages

The transmission of the message will be triggered by a core. This transmission is scheduled through the configuration parameters, using csv files (will be explained later in this text). The message itself is an object which is instantiated before the simulation starts. Several attributes hold the simulation information at each message object. While the message is traversing different units and network elements, the content of the message in the memory will not be copied, but rather a pointer to the message will be handed over from one element to the other one.

In order to satisfy DREAMS needs the native Message class introduced by Gem5 had to be changed. In other words, some attributes and methods have been added to this class. In addition to that, for instantiation of different message types, the following (shaded boxes) have been developed and inherited from this class. Figure 20 represents the relations and this inheritance.

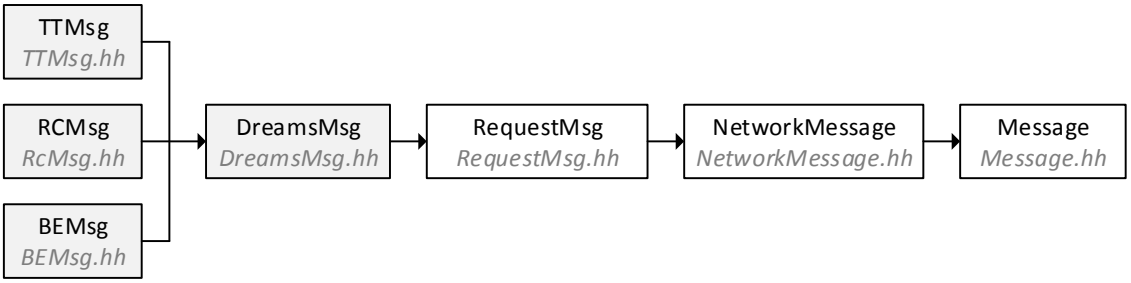

Figure 20 Message class

4.1.1.6 Message buffer

Gem5 provides `MessageBuffer` equipped with a collection of handy methods, which is employed untouched in the simulation of the LRS. The data queue of the ports and the priority queues have been instantiated using this class. As explained earlier, only the pointer to the message is circulated through the LRS. This class deals only with the pointers to the messages. Therefore, all methods in this class deal with the pointer of the message and no modification was needed, though some modifications had to be applied in `Message` class.

4.1.1.7 Mixed-Criticality Address

As discussed in architectural style (D1.1.1), based on the structure of the application and platform, we introduce the following namespace:

$$\underbrace{Criticality.Subsystem.Component.Message}_{Logical\ Name\ (Application)} = \underbrace{Cluster.Node.Tile.Port}_{Physical\ Name\ (Platform)}$$

Each port is associated with a (hardware specific) physical name mapped with a (hardware agnostic) logical name. These names have been defined by `PhyName` and `LogName` classes and build together a mixed-criticality address (`MCAddress`). We defined a class for each of these concepts, each of which embodies the above introduced elements (see Figure 21)

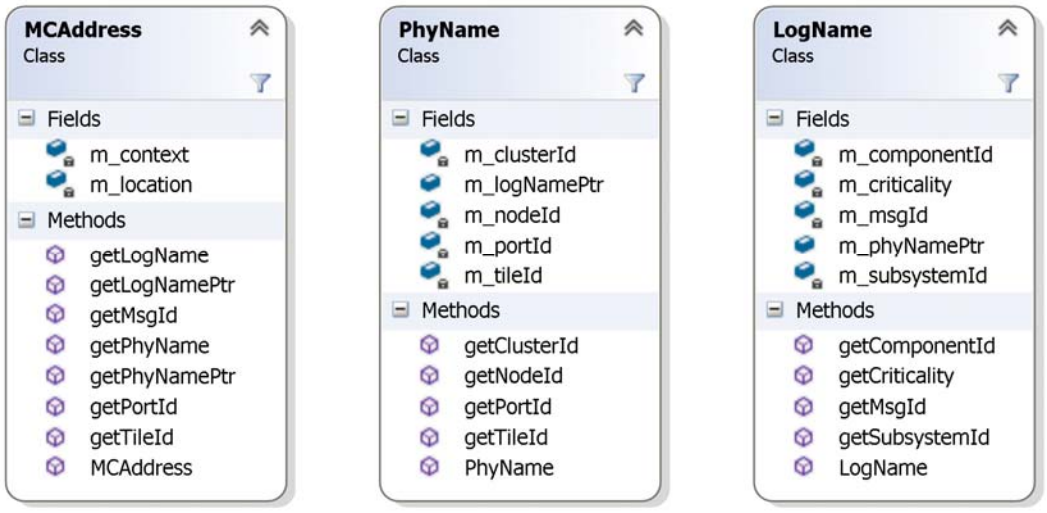

Figure 21 class diagrams of different classes defined for namespace

4.1.1.8 Virtual Links

Virtual Links (VLs) are an abstraction over the on-chip and off-chip networks and hide the physical system structure of the platform from the components. A VL is an end-to-end multicast channel between the output port of one sender component and the input ports of multiple receiver components. Each VL is identified by VLID as well as the source port. It contains the list of destination ports and temporal specification of the link (e.g., in case of a TT link, the period, the phase and in case an RC link, the MINT and the jitter).

In the simulation model, virtual links will be instantiated and mapped with the source and destination ports, once the simulation starts. Furthermore, they are part of the `portConfiguration`, since a part of these configuration have been defined by the mapped VL.

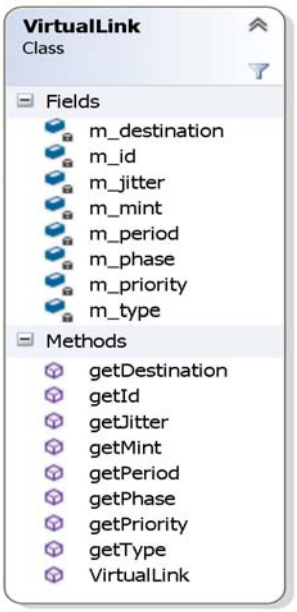

Figure 22 VirtulLink class diagram

4.1.2 The organization of the simulation model

Figure 23 demonstrates the hierarchical representation of the simulation building blocks. Shaded boxes represent the native Gem5 classes used in this implementation and the remaining part shows the developed part, which will be explored in this section. The name on top of each box addresses the name of the class and the name at the bottom shows the filename including the class declaration. Arrows represent the pointers which realize the connections between classes.

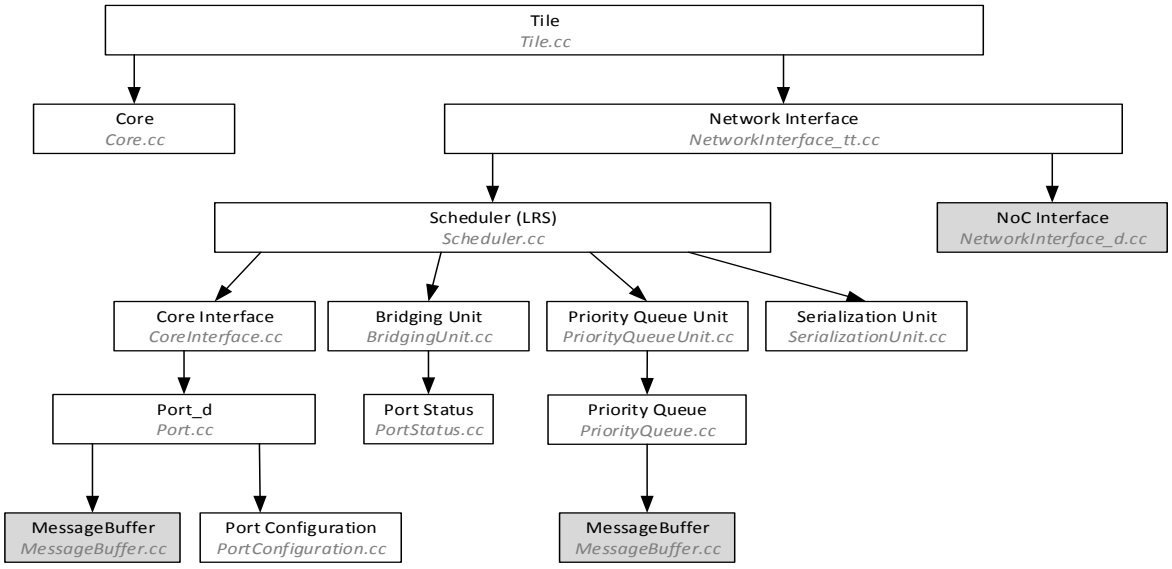

Figure 23 correspondence between classes

In the following subsections these building blocks will be explained.

4.1.2.1 Cores (application layer)

In order to generate the on-chip traffic, the core class has been defined. This class simulates a typical application running on a single partition of a core on the tile. Moreover, cores act as a sink for the generated traffic by other cores and add the timestamp to the messages to provide the needed statistical data for later analysis.

Cores contain a vector which include all scheduled send activities (`core_schedule_table`). This vector will be initialized based on the configuration files once the simulation starts. Furthermore, they have required links to all messages which need to be sent through the NoC (`m_msgs` in Figure 24).

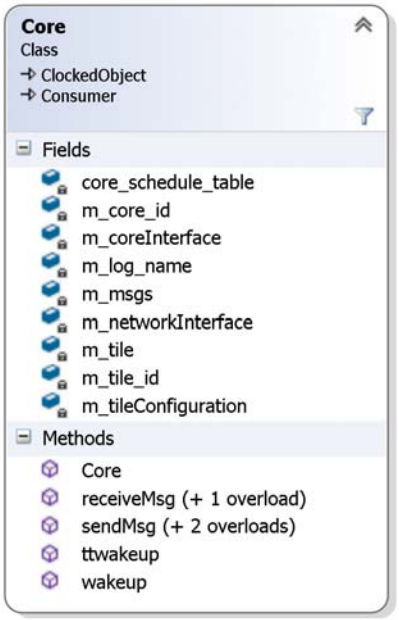

Figure 24 Core class diagram

4.1.2.2 Core interface

The core interface provides the required access to the ports for cores, egress bridging unit and the ingress bridging unit. This access comprised enqueue-ing, dequeuing, updating and retrieving the configuration parameters. This simulates the management services provided by the core interface in order to prohibit any possible interference between components.

As shown in Figure 13, cores do not have any access to individual ports, i.e., the core calls the provided method from the core interface by passing the port id and the pointer to the message and the core interface enqueues the message into the given port (`enqueueCpu`). This provides the core interface possible access controls. Likewise, the egress and ingress bridging unit do not have any access to the individual ports. `PortStatus` objects at the EBU, in contrast, have the required link to the individual ports, in order to be able to update themselves and provide the EBU with the needed information. Figure 25 represents the class diagram of the core interface.

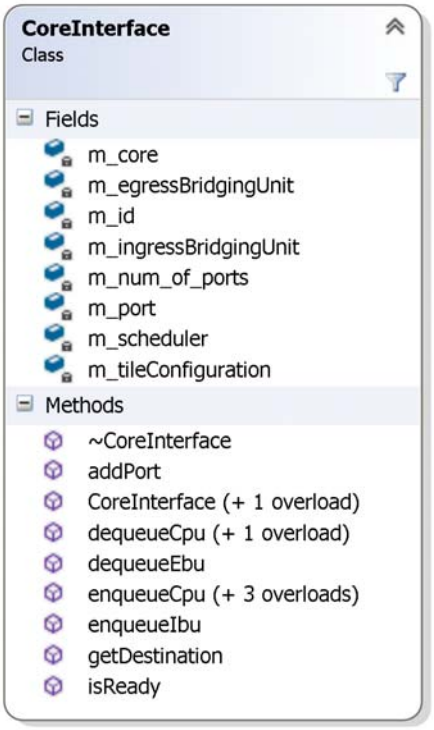

Figure 25 CoreInterface class diagram

4.1.2.3 Ports

Ports are the key elements and starting points to establish the partitioning in a mixed-critical platform. We distinguish between periodic, sporadic and aperiodic ports. These traffic types imply the priority of the message and consequently, the criticality of the respective application subsystem.

Ports are composed of the data queue, the port configuration and the port status.

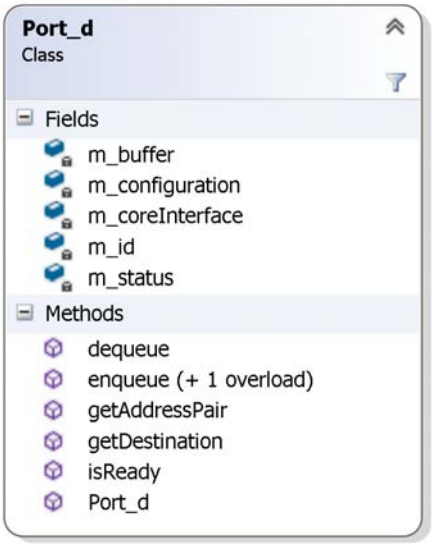

Figure 26 Ports class diagram

4.1.2.3.1 Data queue

This part has been realized by native gem5 `MessageBuffer` class. This class provides needed methods such as `isReady`, `enqueue` and `dequeue`. These methods will be called by the port object using the pointer (`m_buffer` which is shown in Figure 26). Figure 27 describes how an enqueue or dequeue call is initiated from a core or the EBU or the IBU.

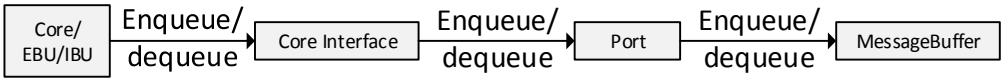

Figure 27 enqueue/dequeue call procedure

4.1.2.3.2 Port Configuration

The `PortConfiguration` class (shown in Figure 28) keeps the configuration parameters of the respective port and embodies all configuration parameters needed for instantiation of the port. This object will not be changed during the simulation. A pointer to an already instantiated `portConfiguration` object is necessary for the instantiation of a port. The `portConfiguration` object is instantiated based on the configuration defined by csv files once the simulation starts.

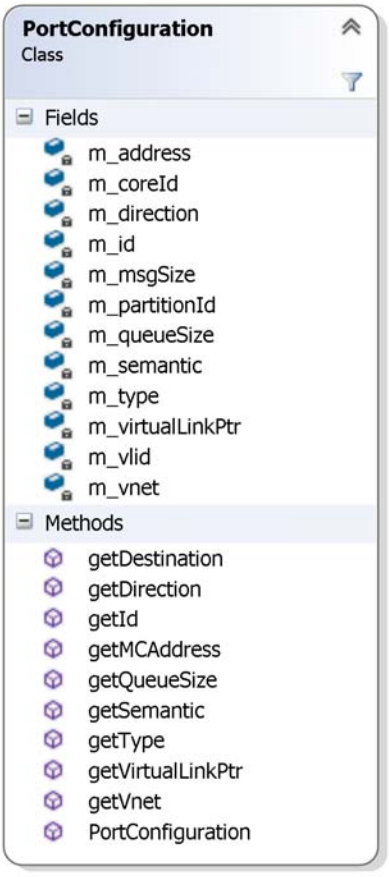

Figure 28 PortConfiguration class diagram

Each TT and RC port is accessible by a unique partition in order to segregate the criticalities. The id of this partition (`m_partitionId` in Figure 28) is part of the `portConfiguration` and this access will be checked by the core interface.

4.1.2.3.3 Port Status

`PortStatus` (shown in Figure 29) reflects the status of each port. The port status will be read by the EBU und updated by the port. This prohibits the EBU from applying unwanted changes into the ports.

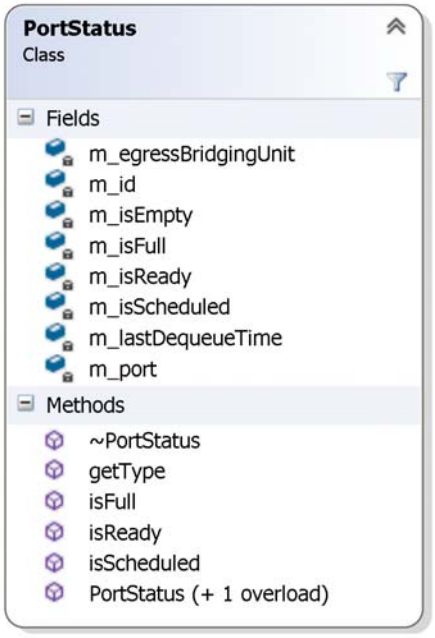

Figure 29 PortStatus class diagram

4.1.2.4 Egress Bridging Unit

The egress bridging unit (EBU) ensures timely dequeuing of TT ports, dequeuing of RC ports only when the minimum interarrival time (MINT) is elapsed and dequeuing of BE ports only if there is enough bandwidth left over by other two types of messages.

The EBU is defined as `Consumer` in order to perform the transfer of TT messages into the priority queue based on the schedule defined in `ebu_schedule_table` (initialized based on the configuration file). On the other hand, for handling the RC and BE messages, there is some situations, in which dequeuing of the respective port is not doable, since at this tick a TT action is already scheduled. In these situations, the EBU must schedule a dequeue process for the future, in order to not miss this event. For this purpose, the `EventWrapper` will be employed to perform the dynamically scheduled dequeue events in `ebu_dq_schedule`. Utilization of the `EventWrapper` requires the inheritance from the `ClockedObject`.

Before enqueue-ing the TT and RC messages into the priority queue, the EBU retrieves the destination of the message, based on the configuration of the virtual link and embeds this address into the message. In contrast, there is no need for embedding the destination address for the BE messages, since the core has already inserted this address into the message. Subsequently, the NoC interface specifies the NoC specific route based on the destination embeds this route into the message.

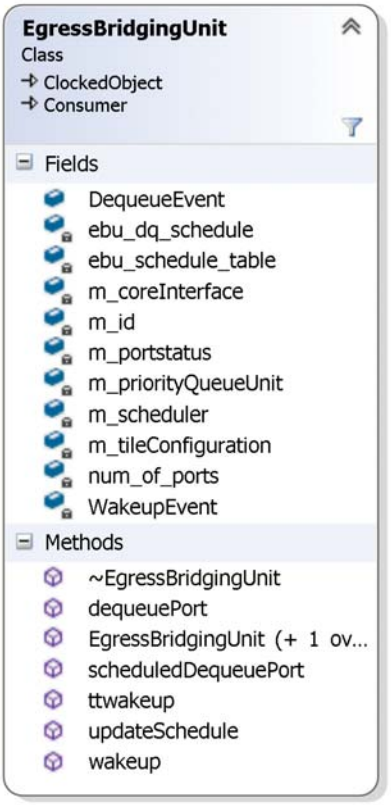

Figure 30 EgressBridgingUnit class diagram

4.1.2.4.1 Transmission of TT messages

Periodic ports will be dequeued based on the TT schedule provided by the configuration parameters. This is done by the `ttwakeup()` method. The EBU wakes up each cycle a TT port dequeue is due. The scheduled `cycle` and the `portId` is available in the configuration of the EBU.

4.1.2.4.2 Transmission of ET messages

RC and BE ports send a wakeup to the EBU once a new message is enqueued. This will be done automatically by `Consumer` class, each time a new message is enqueued into the `MessageBuffer`. Afterwards, the EBU handles the messages based on the state machine described in D2.1.2, section 2.

4.1.2.5 Priority Queue Unit

This unit (shown in Figure 31) has been defined to manage several priority queues and does not address any physical architecture. The PQU includes pointers to the priority queues. Each priority queue is a `MessageBuffer` joint with a `priority`.

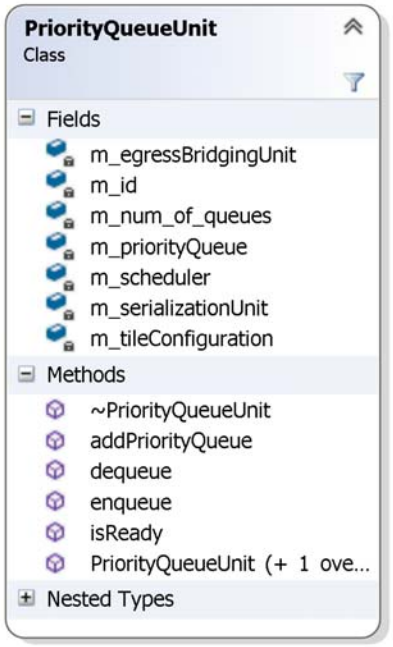

Figure 31 PriorityQueueUnit class diagram

4.1.2.6 Serialization Unit

Looking at the LRS, one observes that at two stages merging of messages take place. First merge happens at the EBU. As illustrated in Figure 18 the EBU merges messages of the same priority into a same priority queue, taking into account the timing constraints defined in the configuration parameters (e.g., periods, phases, MINTS). This merging is needed, since the SU is not aware of these temporal configurations. The second merging takes place at the SU, since the NoC interface is priority agnostic. This is the aim of the SU to inject the messages of highest priority each time there is free bandwidth at NoC.

In order to make sure that the TT messages are subject to minimal latency, timely block can be employed. In this case, a time-triggered blocking window blocks the transmission of any sporadic or aperiodic message in order to make sure the ongoing transmission of those message would not affect the upcoming transmission of TT the message. Alternatively, timely block can be avoided to achieve more efficient network bandwidth usage by the cost of possible latency of TT messages. This delay in worst case will be the delay of transmission of one message.

SU (described in Figure 32) is defined as `Consumer` so that once a new message is enqueued into the priority queues, an automatic wakeup triggers the SU to analyze the enqueued message. This unit inherits from the `ClockedObject` for the same purpose explained for the EBU. There are some cases that individual dequeuing events must be scheduled for the next coming ticks. For this purpose, the `EventWrapper` fulfils this need.

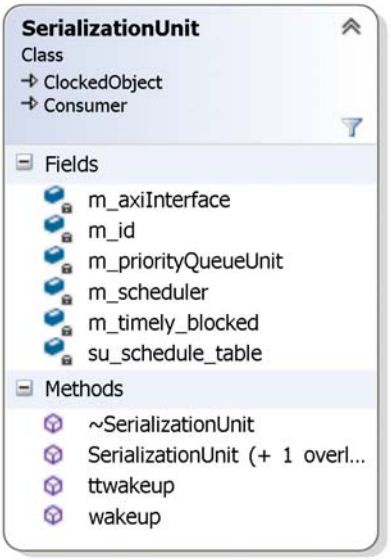

Figure 32 SerializationUnit class diagram

4.1.2.7 Ingress Bridging Unit

The IBU will be waked up once a message coming from the NoC arrives at the LRS. This unit dispatches the message into the destination port based on the physical address defined in the header of the message.

IBU (shown in Figure 33) is defined as a `Consumer` in order to benefit the inherited wakeup method from this class. This means, any new message enqueued by the NoC interface sends a wakeup to the IBU and the IBU will dequeue the message from the buffer between the IBU and the NoC Interface (which is shown as AXI Interface).

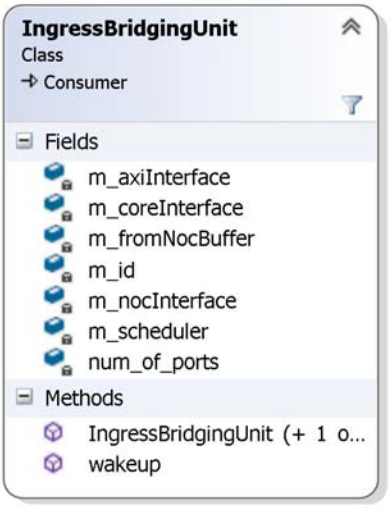

Figure 33 IgressBridgingUnit class diagram

4.1.3 Configuration and data gathering

A typical on-chip end-to-end communication channel has been shown in Figure 34. We can categorize the configuration parameters of the on-chip communication services in four categories, i.e., core, time-triggered (TT) extension layer, the NoC and the global system-wide configurations. In this text, the general (system-wide) configuration as well as configuration parameters for the LRS (the TT extension layer) will be covered.

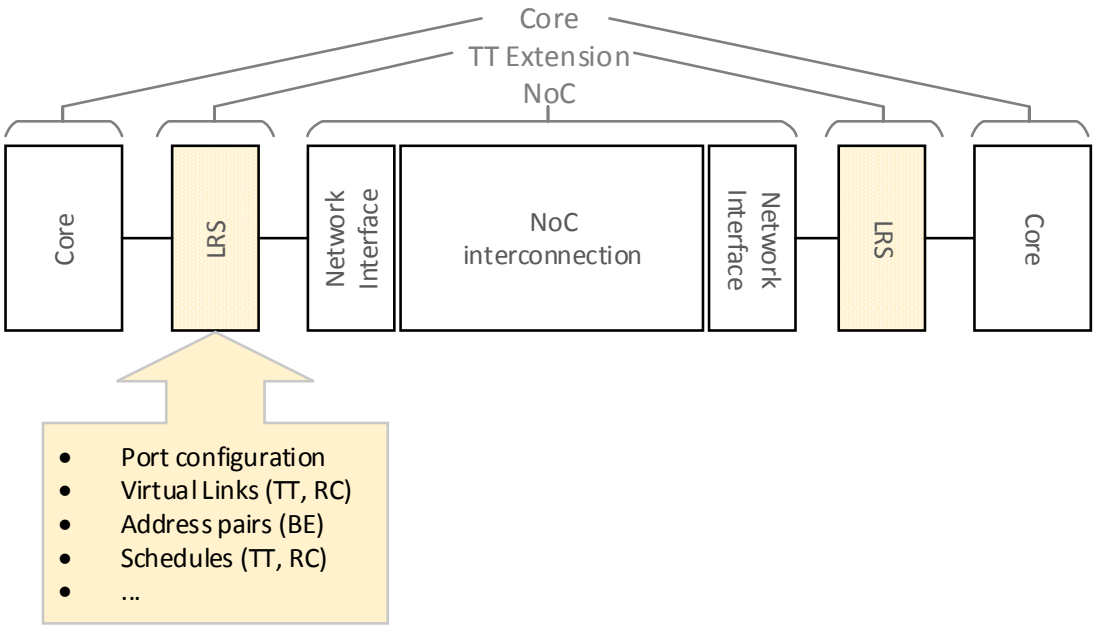

Figure 34 Sample on-chip end-to-end communication channel

4.1.3.1 System-wide configuration parameters

As several tiles of possibly different chips operate in the system, there are some parameters which are necessary for the harmonized operation of network interfaces. These parameters depict the general characteristics of the on-chip communication, mainly the virtual links and the address pairs. Some of these parameters, e.g., the route for virtual links, might need to be joined with those of off-chip networks in order to close the path for system-wide messages. However, in this document only the part concerning the on-chip network will be covered.

4.1.3.1.1 Structure of the messages

In order to be able to communicate, a common defined structure for the messages is necessary. This is characterized by the following parameters:

- Number of flits: this parameter is a static number which can vary for different message types.
- Structure of the message
  - Preamble
  - Message size (in bytes)
  - Source (PhyName)
  - Virtual link ID/Destination
  - Path

4.1.3.1.2 Virtual links

Each virtual link is identified by the following parameters:

4.1.3.1.2.1 VLID

Identifies the VL by a number as ID.

4.1.3.1.2.2 Source

This parameter represents the mapped port with the VL as well as the mapped logical address of the port.

PhyName: the physical name as defined in architectural style (D1.2.1) which is composed of $Cluster.Node.Tile.Port$

LogName the physical name as defined in architectural style (D1.2.1) which is composed of $Criticality.Subsystem.Component.Message$

4.1.3.1.2.3 Destinations

As already mentioned, VLs are end-to-end multicast channels. Therefore, for each VL several destination ports can be defined, each of them characterized by physical and logical name as follow:

* PhyName_i: $Cluster.Node.Tile.Port$ * LogName_i: $Cluster.Node.Tile.Port$

4.1.3.1.2.4 Path

The path is a sequence of physical links which represents the path from the source port to the destination port. Each element of this vector is the id of corresponding router.

4.1.3.1.2.5 Priority (RC)

As the platform is capable of supporting different priorities for rate-constraint messages, ports dedicated for RC messages may have a parameter defining the priority.

4.1.3.1.3 Address pairs

Since the best-effort messages do not benefit the concept of virtual links, each network interface needs to map the logical name addressed by the application to the physical name, which is needed for the NoC. These address pairs will be available as a table and are part of system-wide configuration.

PhyName: $Cluster.Node.Tile.Port$ <-> LogName: $Cluster.Node.Tile.Port$

4.1.3.2 Configuration at the LRS

As described in the architectural style (D1.2.1, Part II, section 1.1) the LRS at the NI is meant to implement the temporal characteristics as well as ensuring the priorities. Configuration parameters at the LRS can be categorized in the following groups:

- Port configuration: configuration of ports at the tile interface
- Egress bridging unit (implementation parameters and the schedule)
- Priority queues (implementation parameters)
- Serialization unit (implementation parameters and the configuration)
- Ingress bridging unit (implementation parameters)

However the egress bridging unit and the serialization unit need to be synchronized and time-triggered in order to control other units. Therefore, the schedule will proposed only for these modules.

4.1.3.2.1 Port configuration

Since the tile interface serves the cores and the bridging units, no schedule for this unit is needed. Each port at the “Core Interface” can be characterized by the following configuration parameters:

4.1.3.2.1.1 Physical address of the port

According to the defined name space in the architectural style (chapter 1.1, namespace), each port can be addressed by a unique system wide name as follows:

- Cluster ID
- Node ID
- Tile ID
- Port ID

4.1.3.2.1.2 Type

Each port can be used for sending or receiving of either of following traffic types:

- TT: for periodic messages
- RC: for sporadic messages
- BE: for aperiodic messages

4.1.3.2.1.3 Partition ID

This parameter helps the hypervisor to map the port with the corresponding software component.

4.1.3.2.1.4 Virtual Link ID (VLID)

By this parameter the defined VLs will be mapped to the port.

4.1.3.2.1.5 Direction

Defines if the port will be for outgoing message (OUT) or for the incoming ones (IN).

4.1.3.2.1.6 msgSize /maxMsgSize

Defines the size of the messages are going to be sent through the port in bytes.

4.1.3.2.1.7 QueueSize

This parameter defines the length of the queue dedicated to the port. The unit is the number of messages

4.1.3.2.2 Egress bridging unit

The egress bridging unit reads (dequeue) the ports at the right time (according to the schedule) and feeds the messages into the corresponding priority queue. In this part, the parameters of this unit followed by the schedule is covered.

4.1.3.2.2.1 Configuration parameters of the egress bridging unit

The current status of each port will be stored at the port status. This status will be updated by the tile interface while the egress bridging unit has only a read-only access to them. The egress bridging unit will be characterized by following parameters.

- List of ports (PhyName, memory address)
- List of priority queues (ID, priority)

4.1.3.2.2.2 Schedule for the egress bridging unit

The schedule for this unit is defined by a periodic set of (dequeue and enqueue) operations defined by following parameters:

- Period
- Phase
- Port (for dequeueing)
- PriorityQueue (for enqueueing)

4.1.3.2.3 Priority Queues

Priority queues serve the egress bridging unit and the serialization unit. Each priority queue is allocated to the messages of the same priority and the same type. This unit can be characterized by following parameters and as explained earlier, there is no need for any schedule.

- QueueID
- Type
- Priority (RC)
- QueueSize (number of messages),

4.1.3.2.4 Serialization Unit

Serialization unit reads the priority queues based on the schedule and feeds them into the NoC. It can be characterized by the following parameter:

- List of priority queues (ID, priority, type)

4.1.3.2.4.1 Configuration of the serialization unit

- Open guarding window
  - Period
  - Opening phase
- Closing guarding window
  - Period
  - Closing phase

4.1.3.2.5 Ingress bridging unit

The ingress bridging unit is meant to dispatch the incoming messages to the corresponding port at the right time (according to the schedule). The configuration parameters are as follow:

- List of ports (PhyName, VLID, Memory BaseAddress)

4.1.3.3 Input configuration files

Following CSV files have been used to define the configuration parameters for the simulation models:

4.1.3.3.1 HWConfig.csv

This file provides the defined topology, number of tiles and number of cores in each tile.

4.1.3.3.2 PortConfiguration.csv

This file represents the configuration all on-chip ports. Each row is dedicated to a single port and contains the following information:

ID	Core-ID	Partition-ID	Phy-Address	Log-Address	Type	VLID	Direction	Message Size	Queue Size

Table 1 The structure of PortConfiguration.csv

4.1.3.3.3 VLConfiguration.csv

This file provides the configuration of virtual links. As explained before, virtual links could have multiple destinations. In order to ease the representation of the file, we consider each destination as the termination point of each branch and define several branches with the same VLID. Each line in the file is dedicated to a single branch of each virtual link and contains the following parameters:

VLID	Type	BranchID	PhyName Source	LogName Source	PhyName Destination	LogName Destination

Table 2 The structure of VLConfiguration.csv

4.1.3.3.4 ExampleCoreSchedule.csv

This file have defined to provide the simulation model with an example scenario. This file contains all details needed by the core for transmission of messages. Table 3 lists these parameters.

Number of Ports				
ID	Tick	Message-ID	Port-ID	Destination-ID

Table 3 Configuration parameters at each core

4.1.3.3.5 TTSchedule_EBU.csv

As described in D2.1.2, sec. 2, the EBU needs a table which contains the portIds and the instants, at which the TT port must be dequeued. This information will be provided by this file.

Number of Ports			
Number of PriorityQueues			
ID	Phase (tick)	Port	PriorityQueue

Table 4 Configuration parameters at EBU

As shown in Table 4, at the beginning of the file, the number of ports and priority queues will be provided. Afterwards, a list of dequeue actions for a single period will be followed. This actions will be repeated in the next periods.

4.1.3.3.6 Configuration_SU.csv

This file includes the parameter which represents whether TimelyBlock will be employed or not. In case of timelyBlock, the opening and closing phases must be provided. Table 5 represents the parameters of this file.

Timely-Block				
ID	PriorityQueue	Period	Opening Phase	Closing Phase

Table 5 Configuration paramteres of the SU

4.1.3.4 Structure of output files

The output of the simulation model is a text file which is generated by Gem5 simulator. This file is defined based on the chosen debug flag.

4.1.3.4.1 Different debug flags

In order to be able to customize the debugging message based on the application, multiple debug flags have been defined as follow:

- `USiegenHWCfg`: this flag enables the debugging messages which shows the current hardware configuration of the LRS
- `USiegenTT`: this flag enables the messages which show the statistical and warnings related to the TT messages.
- `USiegenRC`: this flag enables the messages which show the statistical and warnings related to the RC messages.
- `USiegenMsgTrace`: this flag enables us to trace each message by capturing the message at each stage.
- `USiegenFlitTrace`: this flag enables us to trace each message by capturing the flits at each stage.
- `USiegenTimeStamp`: this flag prints the statistics of each message, once the message arrives at the destination.

4.1.4 Example scenario

In order to be able to demonstrate an example for each item, we present a sample scenario as follows. As mentioned earlier, this document covers only the on-chip communication and therefore, the considered scenario is also pure on-chip.

This scenario represents a chip hosting four tiles (1..4), each of which have multiple ports of different traffic types. Blue boxes represent messages and the number within the box represent the `messageId`. The message ID will be incremented each time the message is sent.

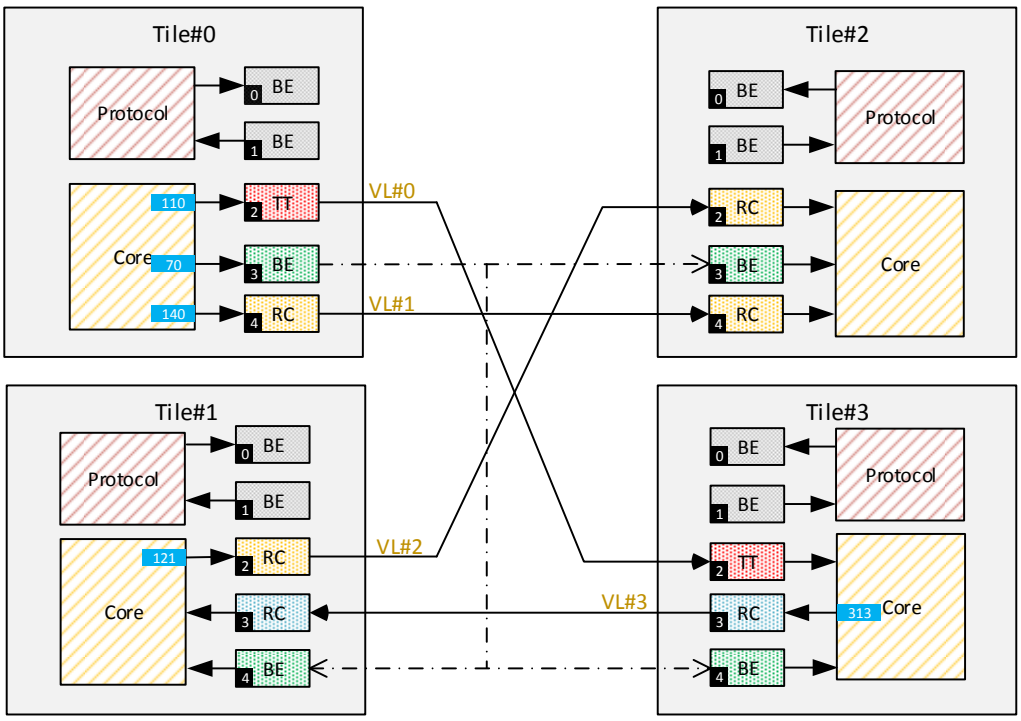

Figure 35 Example scenario representing four tiles interconnected by four virtual links

4.1.4.1 Global configuration

In this example, each message is composed of 9 flits. The global period is 500 ticks and the simulation will run for one period. The topology is Spidergon with four cores and four directories. In order to examine the operation of LRSs, messages are destined only to the cores and not to the directories.

4.1.4.2 Message injection schedule

As defined in the above tables, four virtual links represent the communication channels in the sample scenario. The injection time of the messages into the network has been illustrated in Figure 36

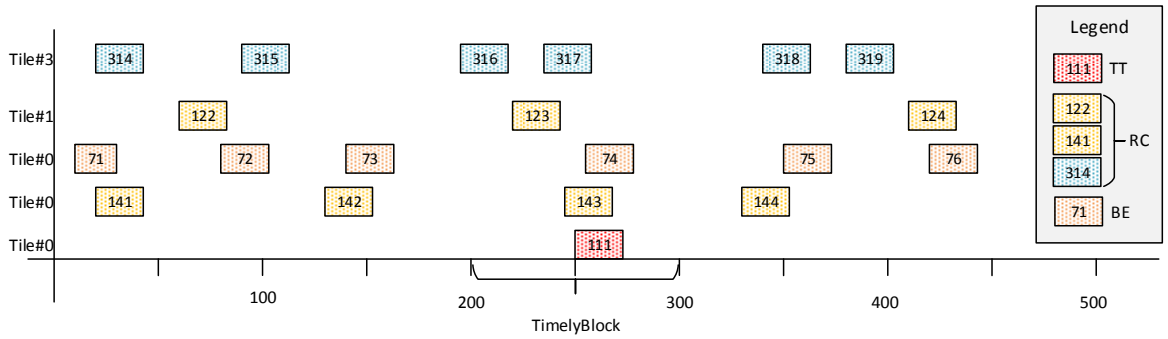

Figure 36 bandwidth allocation for TT and RC messages

The table below is another representation of the above figure. In other words, the table lists the instants at which the messages are planned to leave the core.

NI	Tick	Msg ID	Port
0	10	71	3
0	20	141	4
0	80	72	3
0	130	142	4
0	140	73	3
0	245	143	4
0	250	111	2
0	255	74	3
0	330	144	4
0	350	75	3
0	420	76	3
1	60	122	2
1	220	123	2
1	410	124	2
3	20	314	3
3	90	315	3
3	195	316	3

3	235	317	3
3	340	318	3
3	380	319	3

Table 6 Sample schedule for transmission of messages

Based on the schedule represented in Table 6, schedule of different units can be generated.

4.1.4.3 Sample VL configuration

As shown in Figure 35, following virtual channels interconnect the TT and RC ports of tiles together.

VLID	Type	Source		Destination		Path			MINT/Period	Jitter/Phase
		PhyName	LogName	PhyName	LogName	Hop#1	Hop#2	Hop#3		
0	PE	0.0.0.2	3.2.1.110	0.0.3.2	2.1.3.110	2	2	0	500	250
1	SP	0.0.0.4	2.1.3.140	0.0.2.4	2.4.2.140	1	2	0	100	100
2	SP	0.0.1.2	4.2.1.121	0.0.2.2	4.1.1.121	2	0		200	200
3	SP	0.0.3.3	4.1.1.313	0.0.1.3	2.1.3.313	1	1	0	50	50

Table 7 Sample virtual link configuration

4.1.4.4 Sample port configuration

In the following table, an example configuration of the ports defined in the above scenario (see Figure 35) has been listed. Physical addresses as well as the IDs match the information given in the figure.

Port ID	Partition ID	Physical Address	Type	VLID	Direction	msgSize /maxMsgSize	QueueSize
0	1	0.0.0.0	BE	-	OUT	10	10
1	1	0.0.0.1	BE	-	IN	10	10
2	2	0.0.0.2	PE	0	OUT	10	10
3	3	0.0.0.3	BE	-	OUT	10	10
4	3	0.0.0.4	RC	1	OUT	10	10
0	1	0.0.1.0	BE	-	OUT	10	10
1	1	0.0.1.1	BE	-	IN	10	10
2	2	0.0.1.2	RC	2	OUT	10	10
3	3	0.0.1.3	RC	3	IN	10	10

4	3	0.0.1.4	BE	-	IN	10	10
0	1	0.0.2.0	BE	-	OUT	10	10
1	2	0.0.2.1	BE	-	IN	10	10
2	3	0.0.2.2	RC	2	IN	10	10
3	4	0.0.2.3	BE	-	IN	10	10
4	4	0.0.2.4	RC	1	IN	10	10
0	1	0.0.3.0	BE	-	OUT	10	10
1	1	0.0.3.1	BE	-	IN	10	10
2	1	0.0.3.2	TT	0	IN	10	10
3	2	0.0.3.3	RC	3	OUT	10	10
4	2	0.0.3.4	BE	-	IN	10	10

Table 8 Sample port configuration

4.1.4.5 Sample egress bridging unit configuration

4.1.4.5.1 Implementation parameters

A list of port IDs and `portStatus` memory address shall be provided to the EBU.

4.1.4.5.2 Schedule

NI	Tick	Port	PriorityQueue
0	253	2	TT

Table 9 Sample schedule at egress bridging unit

4.1.4.6 Sample priority queues configuration

QueueID	Type	Priority	Size
1	TT	-	10
2	RC	1	10
3	RC	2	10
4	BE	-	10

Table 10 Sample priority queues configuration

4.1.4.7 Sample serialization unit configuration

4.1.4.7.1 Implementation parameters

A list of priority queues and memory address shall be provided to the SU

4.1.4.7.2 Schedule

Period	Opening phase
500	200

Table 11 Sample opening guarding window for TT messages

Period	Closing phase
500	300

Table 12 Sample closing guarding window for TT messages

4.1.4.8 Sample ingress bridging unit configuration

The implementation parameters of the ingress bridging unit is the same as those of egress bridging unit, i.e., a list of ports and the respective memory address will be provided by the CSV files.

4.1.5 Simulation result

The simulation results for 500 ticks have been shown. The difference between these two results is the employment of the timely block at the serialization unit.

First column in the result represents the core at which the message is arrived. The second column shows the message id. The third column represents the total delay (from the sending core until the arrival of the message at the destination core). The forth column represent the tick at which the message left the sender core and likewise, the last column represents the tick at which the message arrived at the destination core. Next two column (fifth and sixth) represent the enqueue time and dequeue time at the priority queue and likewise, the seventh and eighth column represent the time at which the message delivered to and the time at which the last flit left the NoC interface (`NetworkInterface_d`).

As shown in Figure 37, in case of timely block, the TT message will be delayed 22 ticks, while in case of the shuffling (shown in Figure 38), with the same configuration and schedule, the TT message incurs 25 ticks. This longer delay is due to the fact that TT message (msg[111] have to wait until the bandwidth is released by the previous RC message (msg[143]), whereas, in case of TimelyBlock the bandwidth is blocked for the TT message and the RC message is not able to use it.

```
                  delay core   PQU      NI_d     core

Core[2], msg[ 71]:  20|  10   11; 12   12; 21   30
Core[1], msg[314]:  20|  20   21; 22   22; 31   40
Core[2], msg[141]:  22|  20   21; 22   22; 33   42
Core[2], msg[122]:  18|  60   61; 62   62; 71   78
Core[1], msg[ 72]:  18|  80   81; 82   82; 91   98
Core[1], msg[315]:  20|  90   91; 92   92;101  110
Core[2], msg[142]:  20| 130  131;132  132;141  150
Core[3], msg[ 73]:  22| 140  141;142  142;153  162
Core[1], msg[316]:  20| 195  196;197  197;206  215
Core[3], msg[111]:  22| 250  253;254  254;263  272
Core[2], msg[123]:  97| 220  261;301  301;310  317
Core[1], msg[317]:  84| 235  246;301  301;310  319
Core[2], msg[143]:  83| 245  246;301  301;310  328
Core[2], msg[ 74]:  84| 255  256;311  311;329  339
Core[1], msg[318]:  20| 340  341;342  342;351  360
Core[2], msg[144]:  35| 330  346;347  347;356  365
Core[1], msg[ 75]:  25| 350  351;356  356;368  375
Core[1], msg[319]:  30| 380  391;392  392;401  410
Core[2], msg[ 76]:  20| 420  421;422  422;431  440
Core[2], msg[124]:  68| 410  461;462  462;471  478
```

Figure 37 Simulation results in case of TimelyBlock

```
                delay core   PQU       NI_d     core

Core[2], msg[ 71]:  20|  10   11; 12   12; 21   30
Core[1], msg[314]:  20|  20   21; 22   22; 31   40
Core[2], msg[141]:  22|  20   21; 22   22; 33   42
Core[2], msg[122]:  18|  60   61; 62   62; 71   78
Core[1], msg[ 72]:  18|  80   81; 82   82; 91   98
Core[1], msg[315]:  20|  90   91; 92   92;101  110
Core[2], msg[142]:  20| 130  131;132  132;141  150
Core[3], msg[ 73]:  22| 140  141;142  142;153  162
Core[1], msg[316]:  20| 195  196;197  197;206  215
Core[2], msg[143]:  20| 245  246;247  247;256  265
Core[1], msg[317]:  30| 235  246;247  247;256  265
Core[3], msg[111]:  25| 250  253;257  257;266  275
Core[2], msg[123]:  58| 220  261;262  262;271  278
Core[2], msg[ 74]:  34| 255  256;267  267;276  289
Core[1], msg[318]:  20| 340  341;342  342;351  360
Core[2], msg[144]:  35| 330  346;347  347;356  365
Core[1], msg[ 75]:  25| 350  351;356  356;368  375
Core[1], msg[319]:  30| 380  391;392  392;401  410
Core[2], msg[ 76]:  20| 420  421;422  422;431  440
Core[2], msg[124]:  68| 410  461;462  462;471  478
```

Figure 38 Simulation results in case of Shuffling

4.2 STNoC Gem5 Model

The Gem5 STNoC backbone model extends the 5-stage Garnet Fixed Pipeline Model (cf. Figure 39) provided by the Gem5 simulator. In fact, most of the Garnet network interface, router and link data structures have been modified, as described in detail in Deliverable 2.1.1.

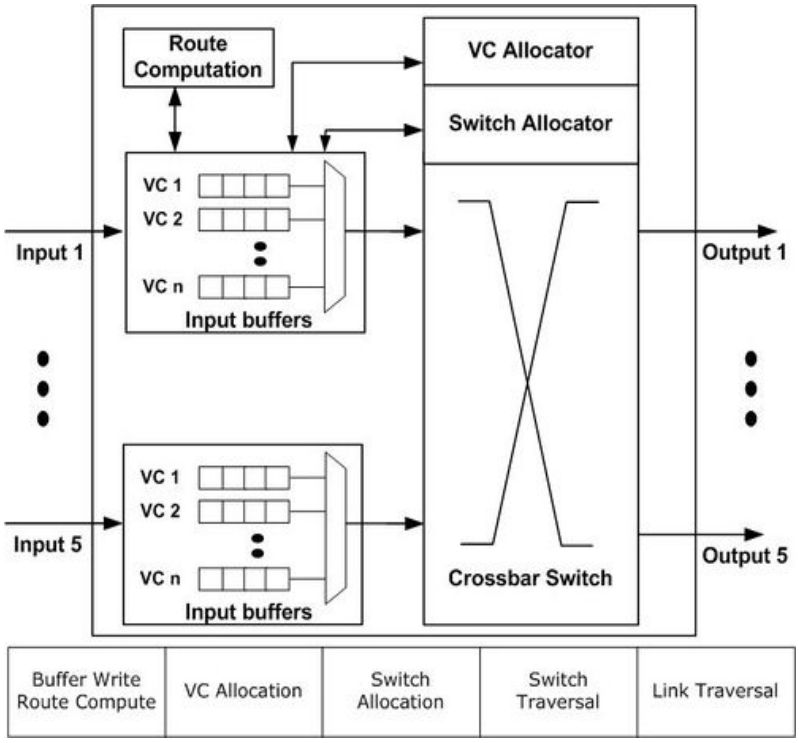

Figure 39: The initial Garnet fixed pipeline router

More specifically, in relation to the Garnet router, we have reduced pipeline depth to match an instance of the STNoC router architecture which processes a packet in only two clock cycles. One cycle is spent for input buffering and route computation stages (Buffer Write and Route Compute, called IB+RC) and one more cycle is spent for all four other stages in Garnet: virtual channel arbitration, switch allocation, switch traversal and link traversal (VA+SA+ST+LT). More specifically, we have encapsulated the Switch Allocator (stage 3 in Figure 6) within the VC Allocator (stage2). For this reason, changes have been made to the event scheduling of all Gem5 router components (as outlined in sections 2.2 and 2.3 of Deliverable 2.1.1.

The Garnet Fixed Pipeline link has been modified to incur 0 cycles delay when “carrying” data or credit flits between two routers or between router and NI (or vice-versa).

The NI is an extension of Garnet Fixed Pipeline model that connects an IP to a router. This module

- checks the protocol buffer (CPU) for packets ready to be sent to the NoC. If the downstream output VC associated with this packet has credits left, then
    - it segments (called flitisize) any such outgoing packet into a number of flits,
    - places them into an output buffer, and
    - schedules the output link for the next cycle
- checks the input buffer for incoming flits to be sent to the attached IP and upon receiving a tail flit, the incoming packet is reassembled from the flits and inserted into the protocol buffer,
- updates credits sent by the downstream router,
- inserts rate control info for implementing NoC Firewall, and

- inserts QoS flags for supporting STNoC QoS technology (FBA) and memory interleaving.

Quality of Service

In addition, we have enhanced the Garnet NI and router to support STNoC QoS policies at flit-level. These Fair Bandwidth Allocation (FBA) policies apply to flits traveling on the same VC (low or high priority one) and competing for the same output port. More specifically, we support: a) STNoC-related fields (`pk_faction_id` and `pk_priority`) for tagging header flits at the Initiator NI and b) corresponding FBA algorithms for rate control at router level. The router policy works hierarchically by

- examining the current faction (`pk_faction_id`) as first level of arbitration,
- implementing Packet Priority, or alternatively Round Robin or LRU policies (implemented at the router) as 2nd level of arbitration, and
- selecting Fixed Positional Priority, Round Robin or LRU as 3rd level of arbitration (Packet Priority is used in 2nd level arbitration, i.e. two packets competing that have the same priority).

Memory Interleaving

New generation multicore SoC platforms integrate memory controllers which implement multiple channels. Ideally many channels can be built (a channel for every DRAM cell would be the ideal solution), but practically, due to wire density and clock skewing issues channels are very difficult to add.

Channels allow parallel access to the DRAM device, increasing the theoretical amount of memory bandwidth by balancing the network load among different channels through memory interleaving techniques. Memory address interleaving can be achieved in software by carefully rewriting the system layer, in particular memory allocation. Alternatively, the STNoC interconnect offers native support for address interleaving at the initiator NI. This function complements address decoding policies.

4.2.1 Configuration Interfaces of the Gem5 STNoC Model

STNoC model configuration involves several Python files and header files referring to an instance of the STNoC backbone (NI, link, synchronous router component) implemented by modifying extending the gem5 Garnet Fixed Pipeline model. The parameters covered in this report are a subset of Gem5 STNoC model configuration, i.e. several performance (e.g. timing) characteristics referring to certain extensions of the STNoC backbone.

General Network Parameters

`src/mem/ruby/network/Network.py`

- **`use_custom_tables`** = `Param.Bool(True, …)` This parameter is set to true if using a custom Spidergon STNoC topology with a deterministic across-first routing designed by TEI. Otherwise, different topologies, e.g. mesh, can be selected during runtime with the default gem5 routing scheme. The number of initiators (CPUs) and targets (cache directories or DIRs connected to memories) is selected independently during runtime (see section on Runtime Parameters).
- **`control_msg_size`** = `Param.Int(8,…)`. This specifies the packet size (in bytes) of a `Control` packet or the size of the header in a `Data` message. Thus, it is used for read or forward requests (which include header and payload) or acknowledgments (header-only packets).

`configs/common/Options.py`

- `parser.add_option("--`**`cacheline_size`**`", type="int", default=64)` This configures the Ruby block size (in bytes) using the runtime parameter `cacheline_size` (described in runtime parameters below). The default value is 64. The total size of a `Data` message (header + payload) can be calculated by summing this value to the `control_msg_size`.

`src/cpu/testers/networktest/NetworkTest.py`

- **block_offset** = Param.Int(6, …) Block offset in bits. **WARNING**: This parameter should always be equal to **$\log_2$(cacheline_size**. If not properly set, routing will not work properly since a device address is embedded into the bits after block offset bits in the packet address.

Network Interface

src/mem/ruby/network/garnet/BaseGarnetNetwork.py

- **ni_flit_size** = Param.Int(16, …) is the flit size in bytes and must always match the bandwidth_factor defined below.

Link

src/mem/ruby/network/BasicLink.py

- **bandwidth_factor** = 16 defines the link width or phit size (in bytes). **WARNING**: This parameter should always be equal to ni_flit_size (defined above)

Router

*src/mem/ruby/network/garnet/*fixed-pipeline/GarnetNetwork_d.py

- **buffers_per_data_vc** = Param.Int(5, …) Buffer size in flits for Data VC and Control VC in VNET0 (used by STNoC). Notice that this is equivalent to the number of available credits at initialization. It is advised to define large enough buffers so that a complete message (consisting of several flits) can be buffered, in order to avoid packets being blocked by the tail of other packets.

src/mem/ruby/network/garnet/fixed-pipeline/VCallocator_d.hh

- **int m_high_prio_vc_offset** = {-1, 0, 1} Names the highest priority VC, i.e. STNoC VC0 (or VC1) for 0 (or 1). It is set to -1 to disable VC priorities. The priority field is taken into account by the router when performing VC and port scheduling. In fact, a flit traveling on a high priority VC may preempt a low priority one; in this case, according to STNoC specifications, the STNoC router behaves "without packet lock" which allows flit interleaving between different VCs.
- **bool m_enable_FBA** = {true, false} enables or disables STNoC Fair Bandwidth Allocation QoS policies referring to each virtual channel. The FBA QoS policies are implemented at the STNoC network interface and router. At least the first-level arbitration scheme is enabled; this scheme is based on the m_pk_faction_id field of the packet and can be set at the network interface via the set_faction_id(int) member function.
- **m_oq_second_arb_scheme** = {PACKET_PRIORITY_, RR_, LRU_} defines the corresponding second-level arbitration scheme when FBA QoS is enabled.
- **int m_oq_third_arb_scheme** = {Fixed Positional, RR_, LRU_} defines the corresponding third-level arbitration scheme which is used when FBA Qos is enabled and PACKET_PRIORITY_ is selected as second level of arbitration scheme, otherwise it is ignored.
- faction_change_weight affects the operation of the first-level FBA arbitration scheme by changing the faction after a given number of packets. This field can be set at the network interface via set_faction_weight (int)member functions.

4.3 Integration of the LRS with STNoC backbone

As explained in section 4.1, the LRS acts as a layer on top of the plain NoC in order to enable the NoC to support mixed-criticality. Hence, in order to examine the functionality of the simulation model, the LRS was integrated into the STNoC simulation model. As already represented in D2.1.1, section 2, the STNoC is a customized fixed-pipeline garnet network which interconnects DREAMS tiles (shaded boxes in Figure 40).

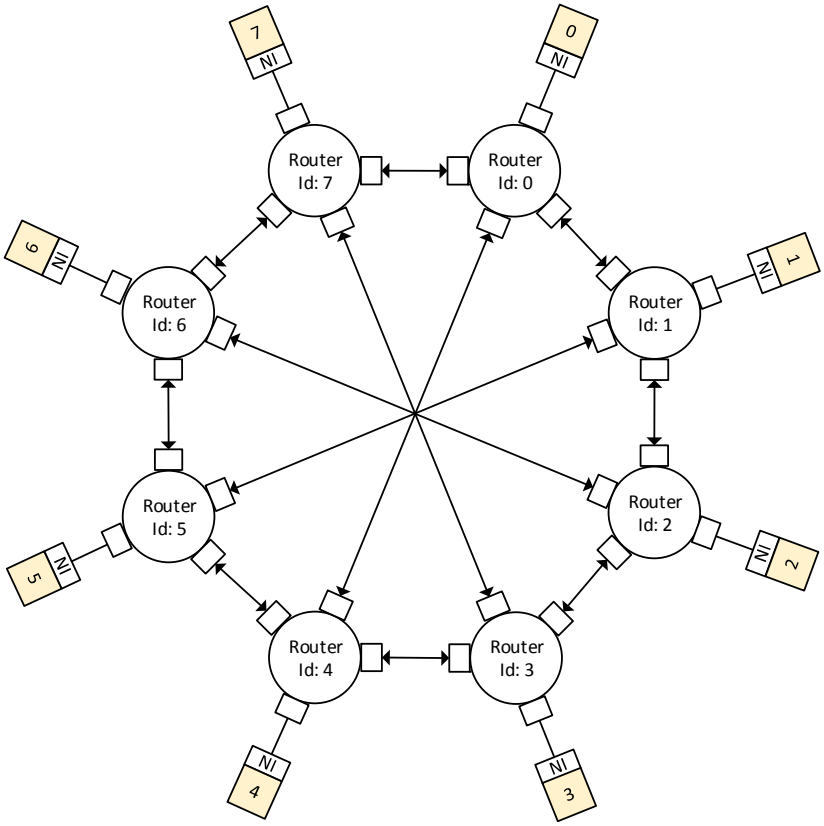

Figure 40 STNoC in Spidergon topology

Garnet however offers no support for TT transmission of message through the network. The LRS have been introduced to bridge this gap by forming an integrated interconnection which supports both TT and ET transmission of messages.

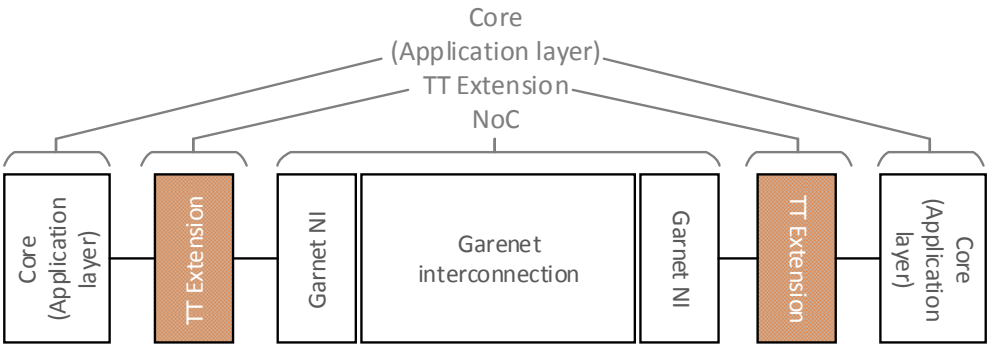

Figure 41 Integrated on-chip simulation model

4.4 Transmission of protocol messages through the LRS

The integration has been done by embedding the LRS between the cache controller and the Garnet NI (see Figure 42). This was performed technically by redirecting the messages coming from the protocol to a single BE port. In this way, all messages coming from the protocol will be enqueued into a single BE port and the port will be later dequeued by the EBU and injected to the NoC by the SU. Likewise, the coming messages from the NoC will be enqueued into another BE port in order to be dequeued by the protocol.

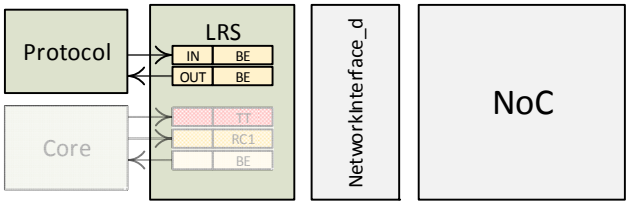

Figure 42 Integration of the LRS with Garnet

4.5 Source Based Routing

The default routing scheme employed by the Garnet is the routing algorithm which tries to choose the route with minimum number of link traversals. In the implementation of the LRS on the other hand, source-based routing scheme has been implemented. In this scheme, the source (the core or the LRS) attaches all needed routing information to the message. This information will be read at each router in order to find the output port at the router. This needs a slight change at the `NetworkInterface_d`, the `Flit_d` and routing unit of the `Router_d`.

4.5.1 Modifications at NetworkInterface_d (NoC Interface)

In order to be able to store the retrieved path from the configuration file, a `routing_table` has been added to the `NetworkInterface_d`. This table will be initialized once the simulation starts. Then for each message, the `NetworkInterface_d` will search for the destination `nodeID` (given by configuration files for each tile) and embed the given path into the message (see Code 2).

```
std::map<NodeID, vector<int>> NetworkInterface_d::routing_table;
```

Code 2 [NetworkInterface _d.hh] added routing table to store the routes retrieved from the configuration

Once the message is delivered to the `NetworkInterface_d` to be injected into the network, the shown code in Code 3 will embed the path into the head flit, in order to provide the router with needed information.

```
auto iter = NetworkInterface_d::routing_table.begin();
iter = NetworkInterface_d::routing_table.find(destID);
if (iter != NetworkInterface_d::routing_table.end())
{
      fl->setRoutingPath(NetworkInterface_d::routing_table[destID]);
}
```

Code 3 [NetworkInterface_d.cc] insertion of the path in flitisizeMessage-Method

4.5.2 Modifications at Flit_t

The following attributes have been added to `Flit_d` to convey the routing information to the routers.

```
class flit_d {
public:
      int getNextRoutingDestination();
      void setRoutingPath(std::vector<int> path);
      bool isSourceBasedRouting();
private:
      std::vector<int> m_routing_path;
      int m_routing_counter;
};
```

Code 4 applied changes in flit.hh for SB routing

The following method (shown in Code 5) returns the output port at the router.

```
int flit_d::getNextRoutingDestination() {
      if (m_routing_counter < m_routing_path.size())
      {
            return m_routing_path[m_routing_counter++];
      }
      std::cout << "No routing entry found" << std::endl;
      return -1;
}
void flit_d::setRoutingPath(std::vector<int> path) {
      m_routing_path = path;
}
```

Code 5 applied changes in flit.cc for SB routing

4.5.3 Modifications at the Routers

In order to avoid unnecessary modifications, the router checks whether the current flit belongs to a message which carries the path information or not. In case of source-based routing, the router reads the respective output port from the flit and returns the value to the switch to forward the message to the given port.

```
if (t_flit->isSourceBasedRouting()) {
      return t_flit->getNextRoutingDestination();
} else
{
      […] // Normal route computing
}
```

Code 6 [RoutingUnit_d.cc] getting the output port at the router

5 Specification of Simulation Environment for Execution Environment

The Execution Environment corresponds to an Application Tile within of the DREAMS architecture (see Figure 43). The Application Tile defines the core execution services, which are realized by a virtualization layer on top of one or more processor cores. The hypervisor establishes the partitions for the execution of components with guaranteed computational resources. Within each partition, an operating system and/or a DREAMS Abstraction Layer (DRAL) are deployed to provide software-support to application services. An application service is realized by an application component inside a partition. The application service provides its service to other components using the core communication services, where a partition with an application service is a communication end point. Core communication services are simulated on external simulation environments while the application tile simulation will be detailed in this section.

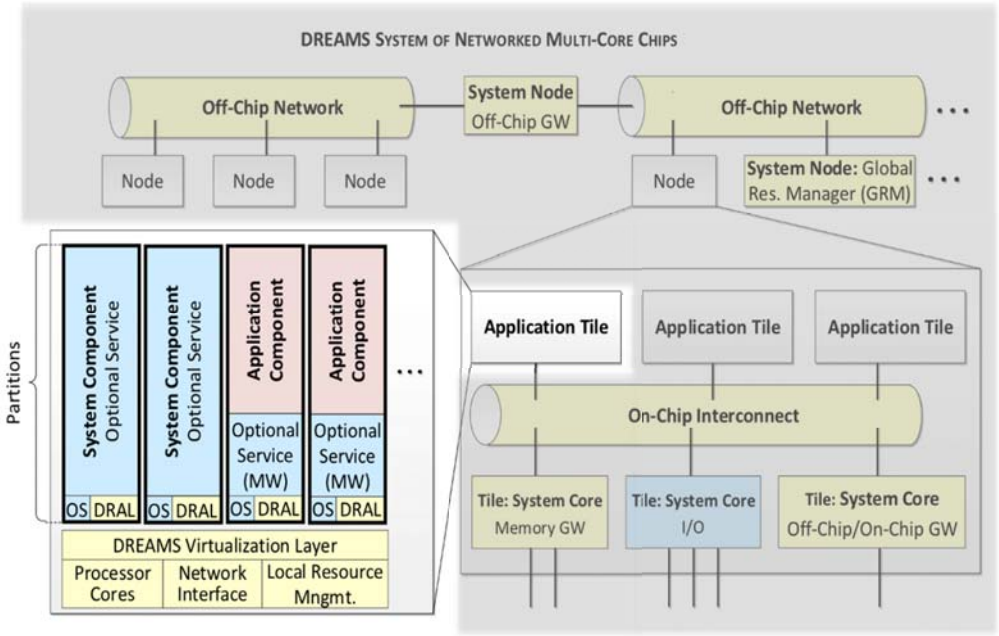

Figure 43: The execution environment corresponds to the Application Tile within the DREAMS platform (source D3.2.1).

5.1 Simulation Building Blocks

OVP has been selected as simulation environment to simulate the DREAMS Application tile or Execution Environment.

Main components to be simulated are the following (see section 2.3):

- Processor: ARM Dual-Core Cortex-A9 processor. OVPSIM or CPUManager simulator provides the ARMv7-A architecture simulation.
- Hypervisor: XtratuM hypervisor. Services and functionalities will be simulated fully.
- Partition: This component could include DRAL, DREAMS application environment, LRM, MON and LRS. These latter three applications will be included depending on the availability of them from other tasks during the deployment phase of the simulation.

Integration of the OVP simulator with other simulation tools is addressed using TCP/IP protocol. For this integration a **DREAMS Wrapper library** will be designed to receive generic simulation control commands and translate these commands to OVP commands. Simulation control commands could be generated from a global coordination component. This wrapper should also be able to receive application data from other simulations and injects it in the OVPSIM simulation and also exports data from OVPSIM simulation to the other simulations.

Execution environment is composed of software components that correspond to hypervisor and partitions. The services provided by the hypervisor will be available on the simulation environment in the same way as for a physical environment. Therefore, the partitions on top of the hypervisor will be those as the implemented on a real machine. This also meaning that DREAMS services such as LRM, MON and LRS will be simulated based on the availability of them in the phase of implementation of simulation framework, i.e., if those services are integrated and instrumented in real software partitions during the prototype implementation, those services could be included as subcomponents of the execution environment.

5.2 Simulation Tools

OVP is used to simulate part of the hardware platform, concretely the processor, memory and its bus interaction. OVP is a simulation environment composed by 3 components:

- **OVPmodels**. These are true Open Source Software that model processors peripherals and platforms.
- **OVP APIs.** The APIs enable users to model processors, peripherals behavioral models and platforms to create extremely fast software virtual platforms. This component consists of C/C++ header files and associated documentation for each API.
- **OVPsim**. This is released as a binary only implementation as a dynamically linked. OVPsim is provided with wrappers for C, C++, SystemC, and OSCI SystemC TLM (TLM2.0) environments.

With the free, open source, OVP models, OVPsim produces Instruction Accurate (IA) simulations of platforms running unmodified software binaries at typical speeds up to 500 MIPS.

The different components have different licenses:

- OVP models created by Imperas and donated to OVP are licensed under a modified Apache 2.0 open source license.
- OVP APIs are licensed with the simulator license - which is a click-through and gives users free usage of the APIs.
- OVPsim is licensed under a click-through license at point of installation. This licenses allows usage by the downloader for non-commercial use and prohibits reverse engineering etc.

5.2.1 Imperas and OVPWorld.

Imperas Software, Ltd. is a team of leading simulation and tool technologists combined with embedded software applications professionals. OVPWorld was formed in 2008 by Imperas Software, who donated a range of capabilities and model libraries to simplify the creation and use of virtual platforms.

5.2.2 CpuManager or OVPsim.

CpuManager is the commercial product available from Imperas. OVPsim is the freely-available version of this product. It is free to use for non-commercial usage and it is hoped that users will contribute back with suggestions for enhancements and will add more models to the open source model library.

5.2.3 ICM, Innovative CPU Manager

The ICM is a C API used to create the platform net list of your design/system for use with OVPsim. ICM allows instantiation of multiple processors, buses, memories and peripherals. Using buses, memories and processors can be interconnected in arbitrary topologies. It enables arbitrary multiprocessor shared memory configurations, and heterogeneous multiprocessor platforms. There are also ICM calls to load application programs into simulated memories and to simulate the platform.

5.3 OVPSim Configuration Interfaces

The ICM API provides a set of functions that cover the platform creation, application load and execution, and information logging and management. In the following subsections the main of these are described.

5.3.1 Platform Creation and Initialization.

The OVPSim simulation mode is based

- ***icmInit***: This function initializes the simulation environment prior to a simulation run. It should be the first ICM routine called in any application. It names the platform, specifies attributes to control some aspects of the simulation to be performed, and also specifies how a debugger should be connected to the application if required.
- ***icmNewBus:*** In order to use an explicit address space mapping, it is first necessary to create a bus to which all address-mapped components will be connected. A bus is defined using the function icmNewBus, which takes a bus name and bit width as arguments.
- ***icmNewProcessor:*** This function is used to create a new processor instance. Processor is initialised using this unique function, despite that the 12 input parameters provide a huge amount of configuration possibilities. Among these stand out the processor architecture variant and endianness, trace and exception emulation.
  - ***icmConnectProcessorBusses:*** Once the processor is created, usually it must be connected to any bus. This function, which takes a processor and two busses, the instruction bus and the data bus, as arguments (the simulator permits processors to have distinct data and instruction busses), connects processor and bus. Most processors use the same address space for both data and instruction accesses, so often the bus arguments have the same value.
- ***icmNewMemory:*** Any number of memory objects can then be defined and connected to the bus. A memory is defined using icmNewMemory, which takes a memory name, access privileges and high address bound as arguments.
  icmConnectMemoryToBus: Once a memory has been created, it can be connected to a bus using this function, which takes a bus object, a memory port name, a memory object and a bus address as arguments.
- ***icmLoadProcessorMemory:*** Once a processor has been instantiated by icmNewProcessor, this routine is used to load an object file into the processor memory. Currently accepted formats are ELF and TI-COFF.
- ***icmTerminate:*** This function must be called at the end of simulation to:
  - Free memory.
  - Return any licenses to the license server.
  - Ensure all tools have finished writing their results files.
  - Print the simulation statistics (if required).

5.3.2 Simulation Execution.

The application load in the memory of the processor could be run using one of these provided functions:

- ***icmSimulatePlatform:*** This function is used to run simulation of the processor and program, for a specified duration. Note that time is not exactly simulated respect to the hardware platform.
- ***icmSimulate,*** can be used to simulate a specific processor for a precise number of instructions and then returns. The precise reason why simulation stopped is indicated by the return code.

The simulation could be interrupted by using breakpoints and watchpoints. When a breakpoint has been set for a specific address, any attempt by the processor to execute at that address will cause icmSimulatePlatform or icmSimulate to return with the processor's stopReason.

- ***icmSetAddressBreakpoint:*** Allow breakpoints to be set for a specific processor
- ***icmClearAddressBreakpoint.***
- Two routines allow breakpoints to be set and cleared for a specific processor and address:
  - ***icmSetAddressBreakpoint:*** to set the breakpoint.
  - ***icmClearAddressBreakpoint:*** to clear the breakpoint.
- Four routines allow watchpoints to be set and cleared for a specific processor, address and access modes:
  - ***icmSetProcessorReadWatchPoint:*** to set the watchpoint in Read Mode.
  - ***icmSetProcessorWriteWatchPoint:*** to set the watchpoint in Write Mode.
  - ***icmSetProcessorAccessWatchPoint:*** to set the watchpoint in Any Access Mode.
  - ***icmClearProcessorWatchPoint:*** to clear the watchpoint in Read Mode.
- Adding callbacks across memory regions allows memory watchpoints, amongst other features, to be implemented. A callback is executed whenever there is either a read or a write access to a specified range of memory addresses.
  - ***icmAddReadCallback:*** to set the Callback in Read Mode.
  - ***icmAddWriteCallback:*** to set the Callback in Write Mode.

5.3.3 Information Access.

Exists a set of functions to access the registers and memory areas during the simulation to trace or debug the execution.

Reading and Writing Processor Registers:

- ***icmGetPC:*** The return value from icmGetPC is of type Addr, which is a 64-bit unsigned integer. For processors with address widths less than 64 bits, this value should be cast to an appropriate sized value if it is to be used subsequently in an arithmetic expression.
- ***icmReadReg:*** The current value of any processor register can be found using icmReadReg, which fills a byref argument buffer with the current value of a named register.
- ***icmWriteReg:*** To write a processor register, there is a similar function icmWriteReg.

The memory space can also be read and written directly using two functions:

- ***icmReadProcessorMemory:*** This function transfer N bytes of data between the simulated memory space and the local buffer using the simulated memory address.
- ***icmWriteProcessorMemory:*** This function transfer N bytes of data between a local buffer and the simulated memory space using the simulated memory address.

5.4 Hypervisor configuration file

Specification of the execution environment is based on specification of the XtratuM hypervisor, i.e., the configuration of the hypervisor will determine the configuration of the platform to be simulated: processors and memory available, numbers of partitions, resources allocation to partitions, scheduling

plan, channels of communication, among others. The DREAMS meta-model maps the majority of information of the XtratuM configuration file (XM_CF). Specific information needed by the XM_CF but not included in the DREAMS meta-model will be pre-configured at default values valid for the simulation.

This section describes briefly the more relevant aspects of the specification for the XtratuM configuration file. However, a complete description of the attributes and features of each element in the XM_CF is beyond the scope of this document. Complete information about this issue can be found in the document "*D2.3.1 XtratuM support of enhanced hypervisor layer services: description and interfaces"* and in the User Manual "*XtratuM Hypervisor for ARM Cortex A9*" [6].

Some of the parts of the configuration file are shown as example below:

- The configuration file contains information about the platform cores:

```
<ProcessorTable>
    <Processor id="0" frequency="400Mhz">
...
    </Processor>
    <Processor id="1" frequency="400Mhz">
...
    </Processor>
<ProcessorTable>
```

- The partitions are described with some attributes regarding their permissions:

```
<PartitionTable>
    <Partition id="0" name="Partition0" flags="system boot" console="Uart">
...
    </Partition>
    <Partition id="1" name="Partition1" flags="boot fp" console="Uart">
...
    </Partition>
    <Partition id="2" name="Partition2" flags="boot" console="Uart">
...
    </Partition>
</PartitionTable>
```

  In each definition of partition, the possible values for "flags" are: "system", "fp", "boot", icache_disabled", "dcache_disabled".
  - "*system*": define the rights to manage the system. System partitions are allowed to manage and monitor the state of the system and other partitions.
  - "*fp*" : enable the use of the floating point coprocessor
  - "*boot*" : the partition starts their execution once their first execution slot is scheduled. If it is not defined, the partition will start their execution when other partition performs a reset partition.
  - "*icache_disabled* and *dcache_disabled*": disable respectively the use of instruction and data caches.
- The configuration file contains information about the scheduling plan. Partition 0 and 1 are attached to processor 0 and the partition 2 is attached to processor 1. Each processor has independent scheduling plans to schedule the execution of the partitions. In the example, each processor defines two plans available:

```
<Processor id="0" frequency="400Mhz">
  <CyclicPlanTable>
    <Plan id="0" majorFrame="3s">
```

```
        <Slot id="0" start="0s" duration="500ms" partitionId="0" />
        <Slot id="1" start="1s" duration="500ms" partitionId="1" />
      </Plan>
      <Plan id="1" majorFrame="1200ms">
        <Slot id="0" start="800ms" duration="400ms" partitionId="1" />
      </Plan>
    </CyclicPlanTable>
</Processor>
```

- Memory regions available in the hardware or memory areas expected to be used in the system are describe:

```
<MemoryLayout>
  <Region type="rom" start="0x0" size="1MB" />
  <Region type="sdram" start="0x00100000" size="1024MB" />
</MemoryLayout>
```

- Description of the hypervisor: device to be used as console, definition of actions for health monitoring events, size of memory reserved for the hypervisor image plus configuration file of the system.

```
<XMHypervisor console="Uart">
<PhysicalMemoryArea size="512KB" />
</XMHypervisor>
```

- For each partition is defined: physical memory areas allocated, inter-partition communications ports, hardware resources and definition of actions for health-monitoring events:

```
<Partition id="0" name="Partition0" flags="system boot" console="Uart">
  <PhysicalMemoryAreas>
    <Area start="0x10000000" size="1MB" />
  </PhysicalMemoryAreas>
  <PortTable>
    <Port type="queuing" direction="source" name="portQ"/>
    <Port type="sampling" direction="source" name="portS"/>
  </PortTable>
  <HwResources>
    <APBDev device="ZEDBOARD_DEV_TTC ZEDBOARD_DEV_UART2" />
  </HwResources>
  <HealthMonitor>
        <Event name="XM_HM_EV_ARM_PREFETCH_ABORT" action=
        "XM_HM_AC_PROPAGATE" log="no" />
        <Event name="XM_HM_EV_ARM_UNDEF_INSTR" action=
        "XM_HM_AC_PARTITION_COLD_RESET" log="yes" />
  </HealthMonitor>
</Partition>
```

- Finally, the inter-partition communication channels among partitions are defined. These channels link the communication ports defined in each partition.

```
<Channels>
  <QueuingChannel maxNoMessages="16" maxMessageLength="128B">
    <Source partitionId="0" portName="portQ"/>
    <Destination partitionId="1" portName="portQ"/>
  </QueuingChannel>
  <SamplingChannel maxMessageLength="128B">
    <Source partitionId="0" portName="portS"/>
    <Destination partitionId="1" portName="portS"/>
    <Destination partitionId="2" portName="portS"/>
  </SamplingChannel>
</Channels>
```

6 Multi Simulation Architecture

Concepts and specifications of the multi simulation architecture are identified in this section. The idea of the interleaving execution of multi simulation architecture is presented in section 6.1. In later sections the major simulation control executions are shown. The control of simulation framework at the chip level and cluster level is described at section 6.2. The simulation control of the execution environment at chip level will be illustrated in section 6.3. The coordination of the simulation components is analyzed in section 6.4. Finally, in section 6.5, dataflow and control-flow of the multi simulation architecture are discussed.

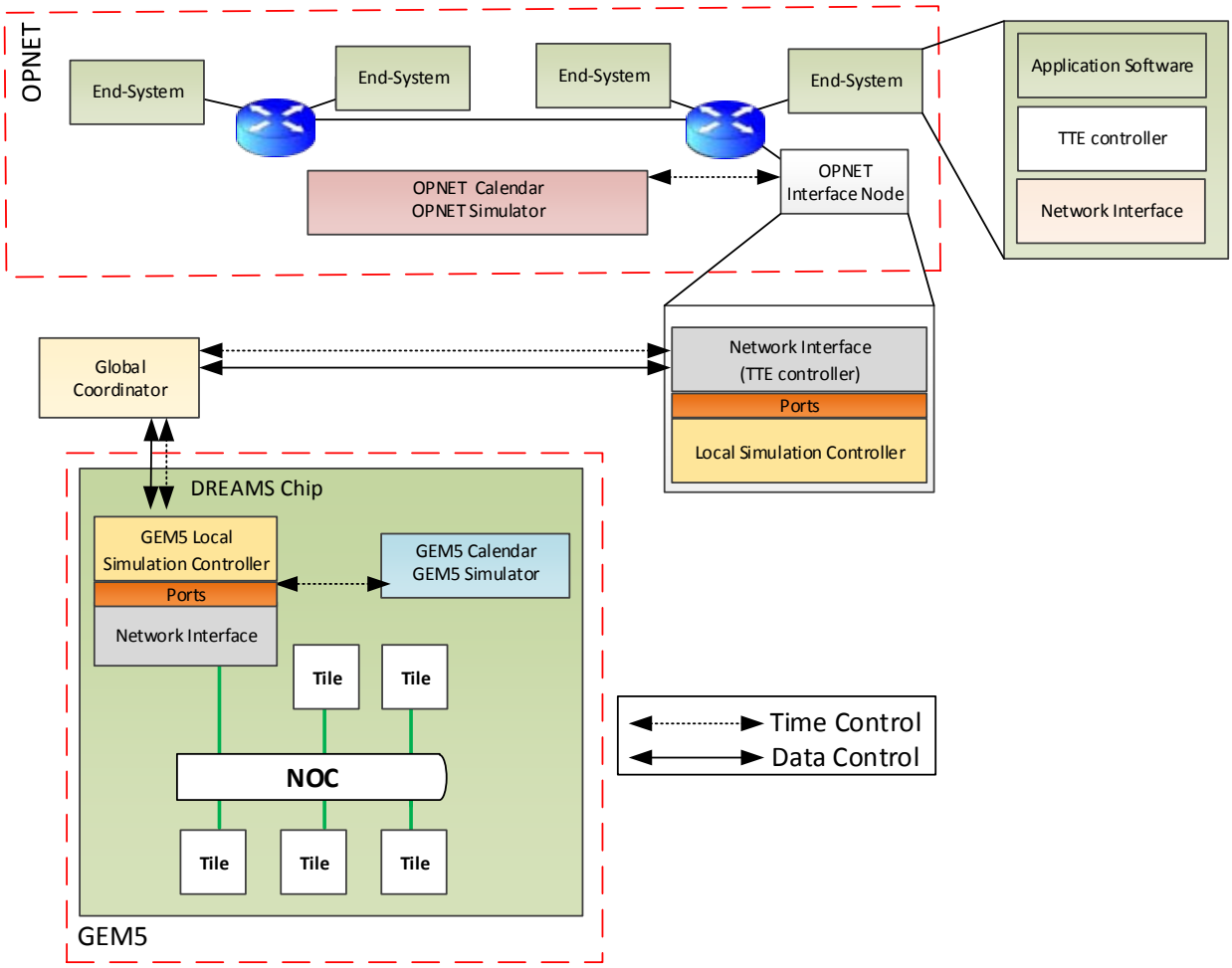

Figure 44: Gem5/OPNET multi simulation architecture

6.1 Discrete Event Simulations

In discrete event simulations, each event occurs at a particular point of time and then it should propagate changes to the simulation controller. During discrete events execution, the simulation controller jumps from one event to the next one, and no events or changes of the simulation system can occur in between two consecutive events.

For the purpose of the DEARMS project, the virtual platform of the project shall support multi simulation framework to combine different discrete event simulators in order to provide on-chip/off-chip simulation framework of the DEARMS platform. This simulation framework will be provided to the demonstrators in WP6, 7 and 8.

The proposed multi simulation architecture will support two major simulation setups. The first one is for the on-chip/off-chip simulation, in which OPNET and Gem5 will be used. For this setup an interleaving of discrete event simulation approach is presented in 6.1.1. Secondly, a tick-based discrete event simulation for Gem5 and OVPSim will be used to introduce an execution environment for on-chip simulation in 6.1.2.

In the DREAMS virtual platform, Gem5 is a cycle accurate simulation tool that is used to simulate the DREAMS chips (see section 3.2.2), in combination with OPNET (see section 3.2.1) that is used to simulate the off-chip network including end systems and switches. These two cycle accurate simulation tools have their own simulation calendar in which discrete events are scheduled on time base. OVPSim on the other hand is not cycle accurate simulator and it requires different simulation control handling. The parallelization of the simulation tools would create overlapping and inconsistency in the executions especially for events that depends on results of the other simulation tool. The coordination of these simulation tools requires the definition of simulation control approaches that avoids such cases. For the scope of the DREAMS virtual platform, we are going to use two simulation approaches, interleaving and tick-based simulations. In order to implement these approaches we introduce a set of coordination controllers that communicate using TCP/IP socket, these controllers will be integrated in the DREAMS multi simulation architecture in order to manage the executions of the simulation tools and to coordinate their execution (see, Figure 44).

6.1.1 Interleaving discrete event simulation

Interleaving simulation executions means that simulators will run once a time. In other words, one simulation tool will run for a specific time and the other simulation tool shall suspend its executions until it is time to run again, based on the collected time events of the upcoming events of both simulation tools.

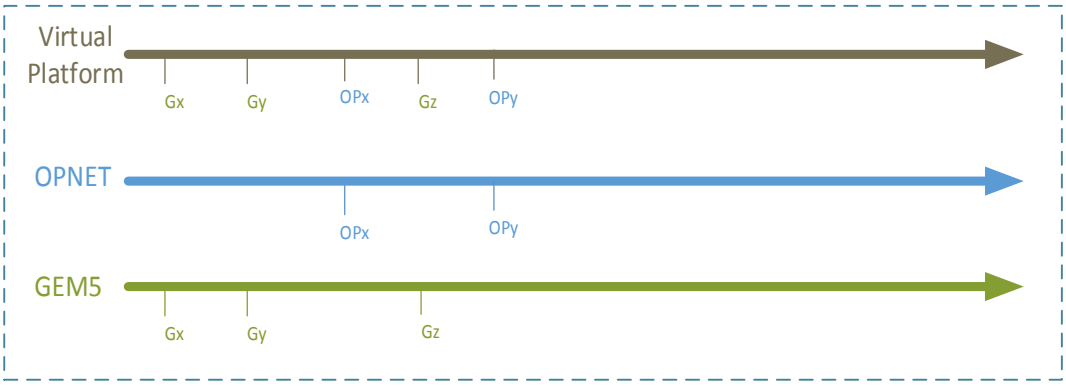

Figure 45: Gem5/OPNET simulation control execution time lines

The major idea in this approach is that the simulation tool with the smallest next time event will be allowed to run until the next maximum time event of the other simulators. In the meanwhile, the simulators with the maximum next time event will suspend until the other simulator finishes its current execution or it sends an early update event. In both cases a new calculation of the next time event will be performed before the simulation tools proceed for the next event in their simulation calendar. In case

that the next events of the two simulation tools are at exactly the same moment, both simulation tools run until this next event.

As shown in Figure 45, the time lines of the Gem5, OPNET show the upcoming events of each simulation based on their own current calendar. The coordination controllers of the multi simulation architecture guarantee that each simulation tool is sending its own execution time of the upcoming event, in order to decide with simulator is going to start and for how long shall it run. In this example, OPNET has the following events {OP_x, OP_y} and Gem5 has {G_x,G_y,G_z}.

Both simulation tools will send their upcoming events {OP_x, G_x}, this would mean that Gem5 would start and can run until the execution time of the first next event of OPNET which means that the first execution sequence is {G_x,G_y,OP_x}. In which GEM5 has run until {OP_x}, then OPNET will run until its planned event {OP_x}. Afterward, the coordination controllers would calculate the next execution sequence based on the remaining events in both simulators and this will result in the following overall execution sequence of the virtual platform coordination { G_x,G_y,OP_x,G_z,OP_y}.

6.1.2 Tick-based discrete event simulation

Simulating the execution environment at the chip level using the hypervisor simulator using OVPSim introduces further challenges. The major difference with respect to the simulation tools considered so far (i.e. OPNET, Gem5) is that we are asked to handle a non-cycle accurate simulation tool. Simulations in OVPSim are based on the Millions of instructions per second (MIPS), this is usually defined during simulation configuration phase, i.e, the temporal duration of each instruction will depend on the MIPS defined in the simulation environment and it is assumed that all instructions have the same WCET. So the temporal granularity in the execution or minimum time step will depends on the duration of an instruction. This would mean that we could not control the execution as we did in using the interleaving approach; consequently, we are introducing the so-called tick-based coordination control for the simulation tools in this case.

The fundamental solution presented in this approach is that simulation tools shall run for one tick and stop for controlling exchanged data or new event requests. The simulation tick is defined based on extensive experiments that need to be done for the OVPSim simulation to determine the optimum MIPS for running the simulation in combination with Gem5.

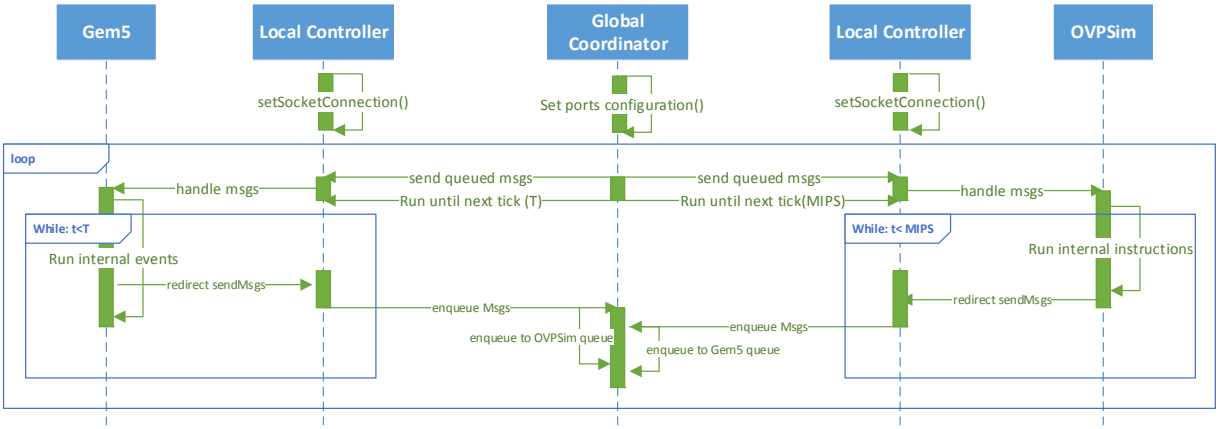

Figure 46 : Sequence diagram of the simulation control for Gem5/OVPSim

As shown in the sequence diagram in Figure 46, the coordination control starts by setting the socket IPs and ports. Then both simulation tools will run for a predefined time (T), where T = [MIPS-Δ, MIPS+Δ]. Where (Δ) is a time variation of the OVPSim due to the non-cycle accurate simulation tool, this Δ is supposed to be very small. Since the MIPS executions might be slightly different each time in comparison with Gem5.

The coordination control will request from both simulation tools to execute until T and stop after that to receive data/control messages from the other simulation tool (if there are any). Then the simulation would start the execution of their event/instructions as long as the simulation time is less than T. In the meantime simulation tools will redirect their messages to the global coordinator (GC), who is responsible of queuing the messages to the correspondent queues of each simulator. Before the simulation tools start for the next tick, the GC should send them their queued messages of the previous tick.

6.2 Multi Simulation Architecture Control

The components forming the multi simulation architecture and different simulation levels are presented in this section. As shown in Figure 44, it is divided into three major parts; the on-chip, off-chip and the global coordinator.

6.2.1 Global Coordinator

Different simulation tools might have execution limitations like operating systems, communication data formats and configuration parameters. Therefore, combining different simulators requires the presence of proper communication interfaces. Furthermore, the coordination of the virtual platform's multi simulation architectures requires continues time control of the event based executions of different simulators for the whole simulation time.

The Global coordinator is responsible for establishing the connection between the different simulation tools. It acts as the server of a client-server TCP/IP based setup. This global coordinator shall be able to create blocking data and time control flow, though the socket based communication should be able to receive and send at any point of time without any message loss. At the beginning of the simulation setup, the global coordinator initiates the socket and creates the connections based on the configurations of the multi architecture simulation. In our case, we have two clients OPNET or OVPSim and GEM5 . In the case of interleaving of discrete event simulation the global coordinator acts as communication bus between the two simulators, without any control for the execution sequence. This control has been moved to the clients (so-called, GEM5/OPNET local controllers.)

In the case of tick-based discrete event simulation, the simulation tools halt after each execution tick to receive instructions and events from the other simulation tool, therefore, the global coordinator consists of the control decision unit based on time, as well as the incoming/outgoing queues that are responsible for buffering the instructions and events that have been sent while executing during the previous tick.

6.2.2 Simulation control at chip level

OVPSim realizes the tile core execution, communication and application services on the top of one or more processor cores at chip level. In order to simulate the multi tiles network-on-chip, the combination of the GEM5 multicore simulation tool with the OVPSim is required. This arrangement will be realized as shown in Figure 47.

The local controllers of the chip level are responsible for managing, redirecting and controlling the simulator execution at chip level. For the purpose of the simulation control at the local controllers, they are structured as follow. A local controller consists of two major parts; first is the part (front-end)

responsible for providing the TCP/IP communication based on the simulation configuration setup. The second part (or the back-end) is the part responsible for interfacing the controller to the on-chip network interface of the network-on-chip. It is also responsible for messages mapping and parsing in both directions from and to the simulator.

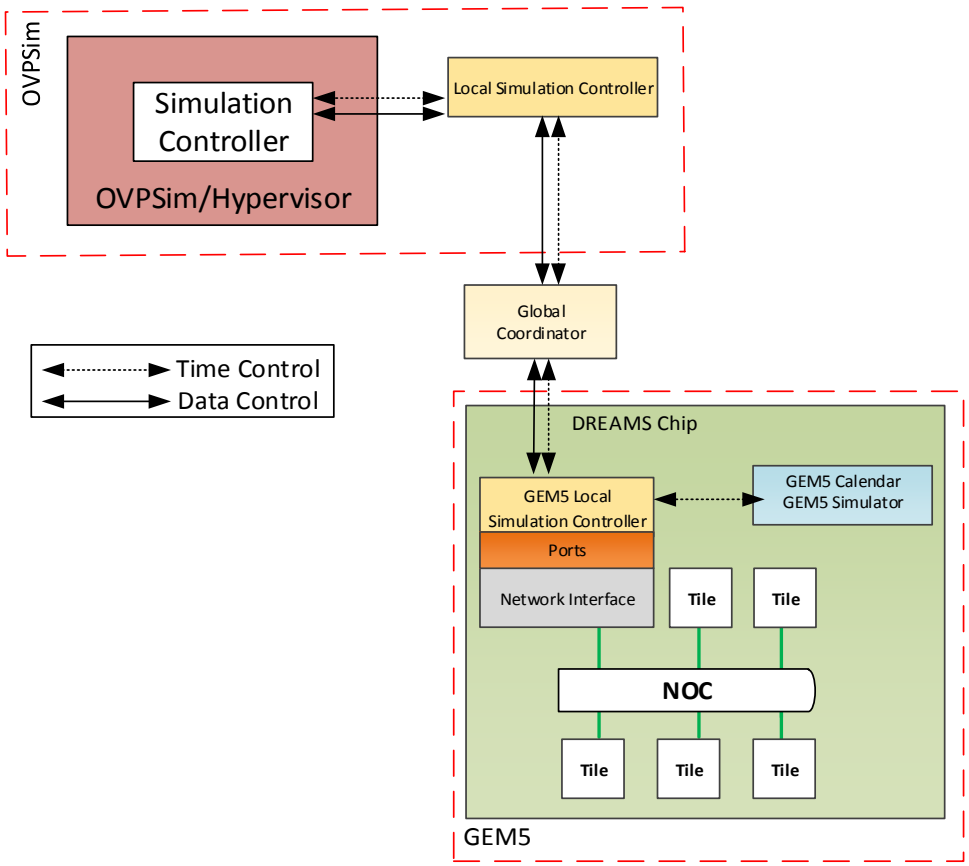

Figure 47: Gem5/OVPSim multi simulation architecture

In regard to the chip level simulation, the global coordinator defines the control sequence of the execution time tick (T) and it will allow the local controllers of the GEM5 and OVPSim to run until this time T, by invoking the control events into the simulation calendars of the simulators (see Figure 48).

Form OVPSim point of view, the (T) time represents the minimum set of instructions that have to be executed within this time limit. In the meanwhile the local controller is responsible for buffering the OVPSim instructions of the current execution the global coordinator, while it is executing the instructions of the previous tick. In this way there will be no interaction between the two simulators while running. As shown in Figure 48, after each tick the global coordinator will redirect the instructions/events to the correspondent local controllers that will modify the simulation calendars accordingly. Similarly, the GEM5 local controller is able to redirect data messages, alter and insert control events to the GEM5 simulation calendar. Further details of the GEM5 local controller will be discussed in the next subsection.

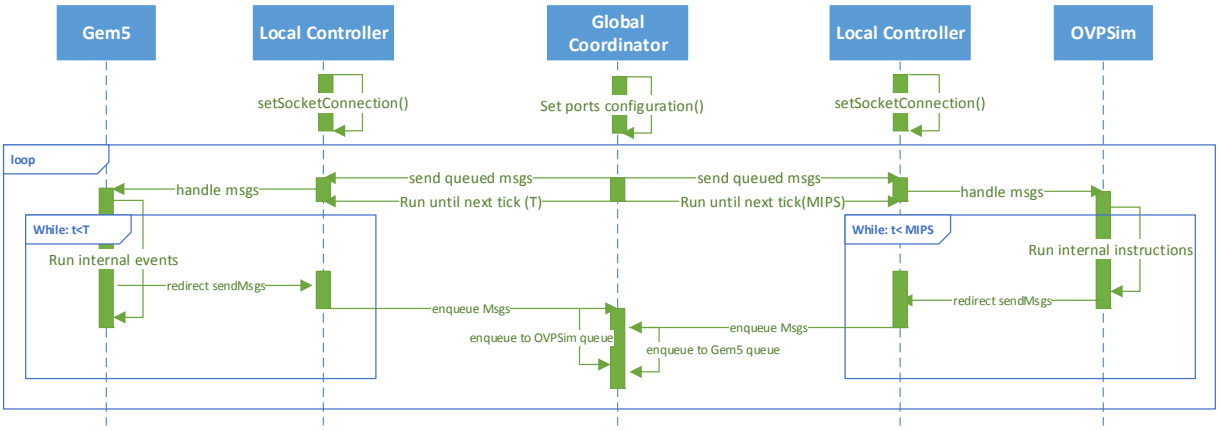

Figure 48: Sequence diagram of the OVPSim local controller simulation control

Before the simulation of the OVPSim simulator starts, the virtual board or platform must be set up. During this process the platform creates and initializes the processors and peripherals and interconnects them using a bus.

Memory areas must be created and initialized using the binary image of the Execution Environment, and must be connected to the bus. Finally the platform is connected to the wrapper and waits for control commands coming from the global coordinator.

As explained in section 6.1.2, the simulation is tick-based. The reason is that for OVPSim simulation time is advanced at the end of each simulation quantum (which is global across the simulator), so that by choosing the length of time simulated in each quantum and the number of instructions executed by each processor in a quantum (by the MIPS rating), each processor is given a number of instructions to execute. In this case the quantum is set up in a value of 1 instruction, the minimum value.

With these constraints, once the platform finishes the initialization process remains at the Execution Environment code entry point waiting for commands. According the tick-base policy the commands could be:

- runUntilNextTick(): When the OVPSim simulator receives this command, it runs the next instructions and the simulation time is updated until T=T+(1/MIPS). Is not possible under any circumstances to stops the simulation until this command is finished.
- updateRegistersState(): These command is used when the Gem5 of the coordinator needs to update the memory map values.

If during the execution of an instruction the processor modifies or accesses the memory region where the STNoc is mapped, this access is communicated to the coordinator using the message queue.

In the same way, Simulation Time could be accessed or modified at any time using this mechanism.

6.2.3 Simulation control at cluster level

The multi simulation architecture at the cluster level covers on-chip/off-chip simulation interests of the DREAMS demonstrators based on the virtual platform requirements defined in D1.1.1. As described at the beginning of this section, in the DREAMS virtual platform we combine OPNET for the simulation of the off-

chip routers, gateways and connections with the GEM5 multicore Garnet/STNoC model for on-chip simulation by using the multi architecture interleaving simulation approach. As described in subsection 6.2.1, the global coordinator is responsible of redirecting the virtual platform control and data messages without intervening in the simulation itself.

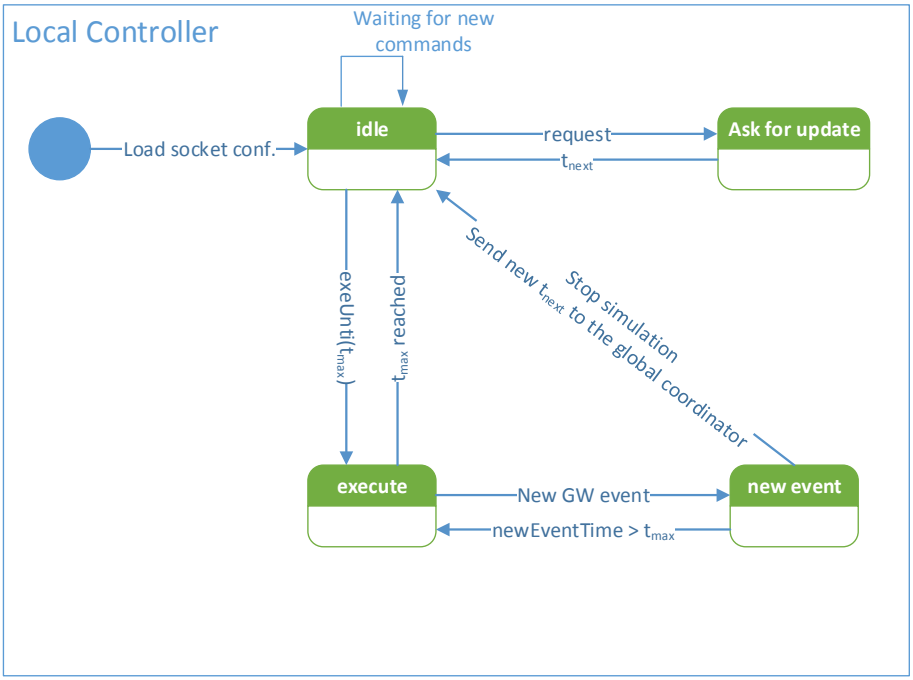

Figure 49: Local controller state machine

The simulation control is in total responsibility of the local controllers of the simulation. The execution time of the simulators is defined dynamically based on the simulators local schedule. The state machine (Figure 49) illustrates the execution flow of the local controllers. The local controller requests a time update to find the time of the next planned execution event of the other simulator schedule, this is happening in both local controllers in order to find and calculate locally the execution order and duration as discussed earlier in section 6.1.1. The outcome of calculated max-next is always the same in both controllers in each calculation request.

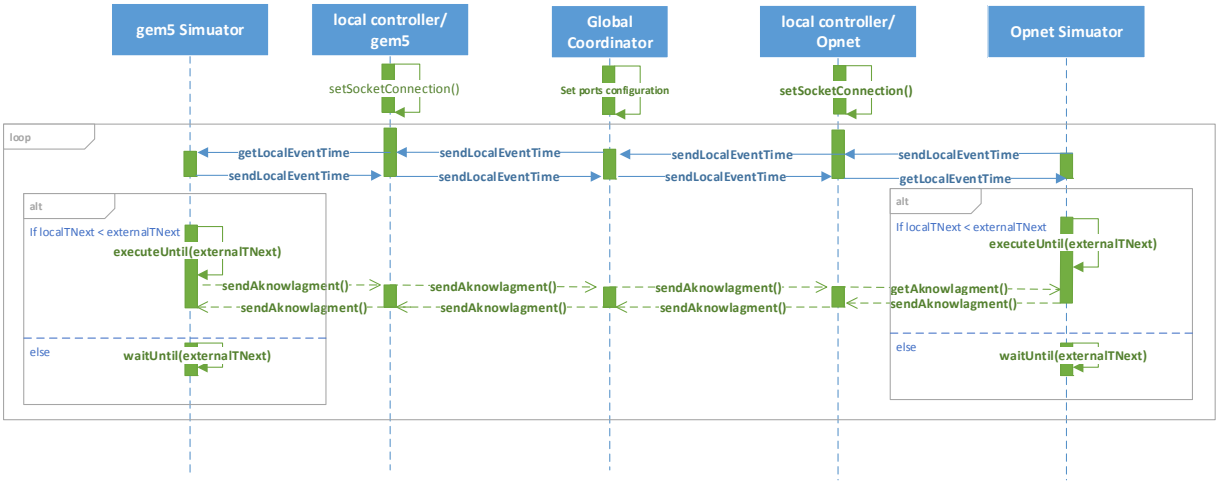

Figure 50: Sequence diagram of the simulation control, regular execution case

The simulation starts with the initiation of socket communication between the global coordinator and the local controllers. The coordination of the two simulators can be separated into two cases. The regular execution case, as illustrated in Figure 50 the running simulator completes its execution until the end of the calculated maximum next-time. In this case, one simulator is running while the other is waiting for a "start" instruction. Then the simulation tool that was on hold has to execute until the calculated max-next (it will be his planned next time event), then a new calculation request is started. The two simulation tools will run alternately until a new event that has a dependency with the other simulation tool is sent.

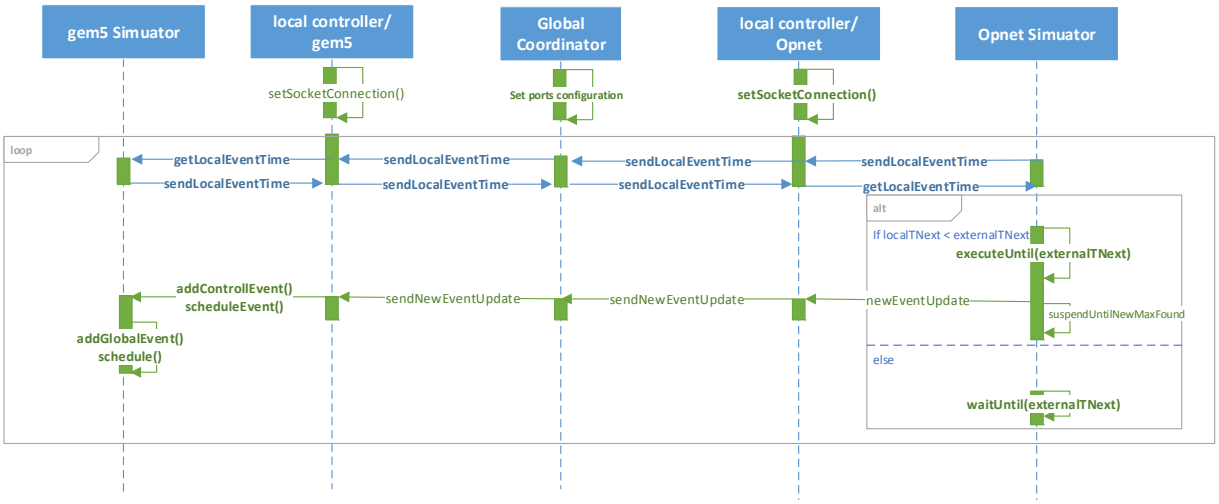

Figure 51: Sequence diagram of the simulation control, new event update case

New events that are addressed to the other simulation tool might occur while executing before the end of the max-next time. Those events (see, Figure 51 "newEventUpdate") induce the second case of coordination, in which, the running simulation tool will send an update event then its local controller will set the simulation tool on hold and redirect the new event to the other simulator which will start by adding this new control event to its local simulation schedule and the a new round for calculating the next-max time in both local controller is started.

6.2.4 Dataflow and Control flow between Simulations

The local controllers consist of a front- and a back-ends. The front-end is responsible of the socket based connection and it is similar in all local controllers for the different simulation tools. The back-end of a local controller is responsible of the message mapping and parsing. The back-ends differ from one simulation tool to another from implementation point of view while they have the same functionalities in regard to the data and control flows. The messages exchanged between the simulators are classified into the control events responsible for guaranteeing the correct execution order in the simulation tools, and data messages which include regular communication messages between the different on-chip/off-chip tiles.

Event Types
Data Message
Run Until <time T>
Event Update <time T>
Acknowledgment Event
Start Simulation Event
Stop Simulation Event

Data Message Format
Message Size
Creation Time
Message Sequence Number
Message Deadline
PortID
Message Type
VLID
BE source Logical Address
BE Destination Physical Address
Criticality Level

Figure 52: Multi Simulation Architectures Event Types

6.2.4.1 Event Types

The DREAMS virtual platform data message structure contains the following fields:

- Message Size (integer): this parameter contains the messages payload size
- Creation Time (double): the message creation time, used for statistical and fault injection purposes.
- Message Sequence Number (integer): the message identification number, used for message routing and for statistical and fault injection purposes.
- Message Deadline (double): The message time limit, used for statistical and fault injection purposes.
- PortID (integer): The message port identification number, used for routing.
- Message Type: The message type, it can be (Time Triggered, Best Effort or Rate Constraint message)
- VLID (integer): Virtual link ID of the message, used for routing
- BE source Logical Address: In case of best effort message the logical address of the source of the message shall be included as defined in the architecture style deliverable D1.2.1.
- BE Destination Physical Address: In case of best effort message the destination physical address of the message shall be included as defined in the architecture style deliverable D1.2.1.

- Criticality Level: To support different message criticality levels, the message criticality shall be included.

Moreover, the control flow of the simulation requires the following control event messages:

- Run Until <time T>: This control message inform the local controller to run until a specific time (T), this time calculated based on the max-next time as described earlier.
- Event Update <time T>: This control message is sent in case of schedule changes in a simulator while running, it stops the current running simulator and update the other simulator with the new event and its execution time.
- Acknowledgment Event: this event is sent when a simulator has completed his execution of a previous “Run Until <time T>” event in order to synchronize the simulation controllers and to start a new max-next time calculation.
- Start Simulation Event: This event starts the simulator through it is local controller to begin the simulation and set the socket based connections.
- Stop Simulation Event: used to announce the end of the simulation for simulation controllers and end the socket based connections to the global coordinator.

7 Model-Driven Configuration of Simulation Environment

The model of an instance of the DREAMS architecture to be simulated is concretely available as a set of EMF model files, since the relevant DREAMS sub-meta-models (application, platform, platform specific) are defined as EMF meta-models (see D1.4.1 and D1.6.1).

In order to realize the tool supported configuration of the simulation blocks out of the DREAMS model of an instance of the DREAMS architecture, two approaches have been considered:

1. Simulation block specific configuration code that uses the EMF API to extract the needed configuration data from the EMF model files, in order to configure “on the fly” the simulation block during the initialization phase of the simulation.
2. The model-to-text transformation framework of Autofocus (see T4.2) is used for generating simulation block specific configuration files with dedicated CSV format off-line before the simulator is started. During the initialization phase of the simulation, these configuration files are read in order to set up the simulation building blocks. The model-to-text transformation is based on templates which consist of static text and dedicated instructions that extract data from specified system model during the execution of the model-to-text transformation and insert them at specified positions in the template in order to produces the configuration file.

After some investigations, it has been concluded that the generation of configurations files with the help of model-to-text transformation is the more convenient approach. The reason is twofold: on the one hand, the EMF API for extracting information from a DREAMS model is by default only available in Java, whereas the simulation blocks are implemented in C/C++. A generator of a C++ version of the EMF API has been identified but found to have blocking limitations such as the handling of multiple model files. On the other hand, the usage of simulation block specific configuration files allows to configure the simulation building blocks even if the model-to-text transformation is not (yet) available or the simulation building blocks are used in an other than the DREAMS context.

Thus, the generation file generator will be implemented as shown in Figure 53. If the DREMAS model does not cover certain parameters needed in the configuration files of a simulation block, then these could be added statically or dynamically through the generation templates.

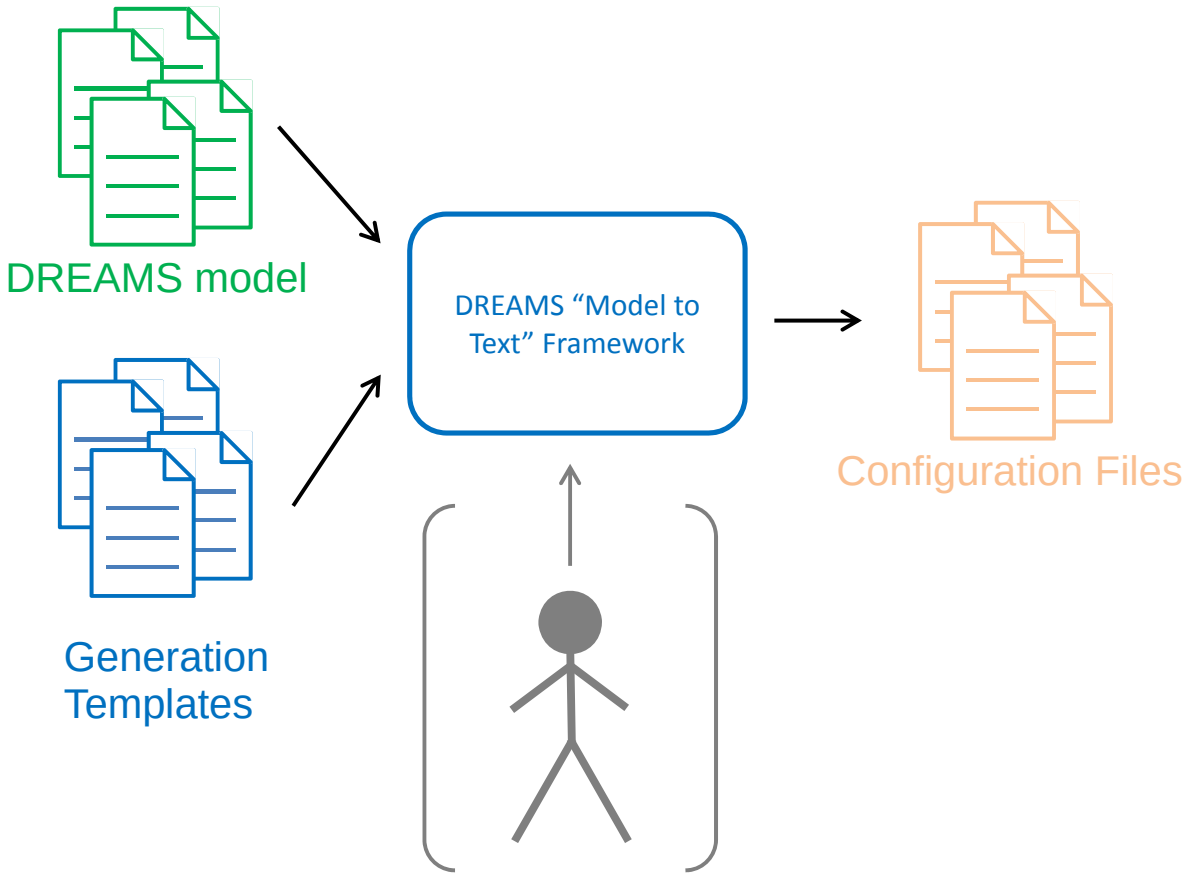

Figure 53: Inputs and outputs of the model to text configuration file generator

7.1 Information from Platform-Specific Model for Simulation Environment

Most of the information needed for the generation of the configuration files of the simulation block are provided by the "platform specific model". But since the detailed specification of the "platform specific model" will only be available in M32 we can give here only an overview of the kind of information it will provide.

The central purpose of the "platform specific model" is to define the mapping of the applications to the hardware platform and how the provided execution and communication resources are shared.

Applications are mapped to partitions that are regularly allowed by the LRS to execute, on one or several processor cores, according to the current resource configuration. Inside a partition, tasks of the application are scheduled according to some scheduling policy, such as cyclic and priority based scheduling. The description of these aspects in the "platform specific model" provides the information required for configuring the "Execution Environment" simulation block.

Depending on the mapping of the applications, some messages need to be sent over the on-chip or even over the off-chip network. How these communication resources are shared for the transmission of the messages is defined by the on-chip and off-chip network configurations. The description of these aspects

in the “platform specific model” provides the information required for configuring the “Network Interfaces”, the TTEthernt, the EtherCat and the Noc simulation blocks.

7.2 Specification of Configuration File Generators

The configuration file generators will be integrated into the DREAMS model editor and implemented with the help of the DREAMS model-to-text transformation framework or direct transformations (e.g., for XML files that conform to a schema). From context menus, it will be possible to execute the configuration file generation separately for each of the simulation blocks. Depending on the specificities of the simulation block, the user may be asked to set values for parameters that are not covered by the DREAMS meta-model (e.g., by providing an external configuration file template that is either created manually or using existing tools for the corresponding building block). As pointed out in D1.3.1, Sec. 3.6), this approach has been selected for configuration generation in order to separate implementation specific (i.e., configuration parameters that are specific to concrete implementations of the DREAMS architectural style), and use-case specific (e.g., parameters that are specific to the configuration of the simulation) from parameters stemming from the DREAMS architectural style.

8 Analysis of Simulation Traces

In this section, we describe the higher level properties that will be synthesized from the traces (Section 8.4) and the events that must therefore be logged in the simulation traces (Section 8.1). Furthermore, we explain the overall structure of the traces files (8.2) as well as the detailed specification of each kind of trace file (Section 8.3).

8.1 Events that need to be logged in the simulation traces

Let us recall that in discrete event simulators, the evolution of the system model is driven by the occurrence of events that induce state changes. To be able to analyze the simulated behavior after the simulation and synthesize higher level properties, a certain subset of these events needs to be written to trace files. These subsets of events depend on the properties that should be analyzed and on the objectives of the analysis. In the following section we describe which events are needed for end-to-end delays.

8.1.1 End-to-end Delays

In the DREAMS meta-model, end-to-end latency constraints are described with the help of timing chains. By definition, a timing chain consists of a stimulus event and a response event that are related by a cause and effect relation. If the response is the writing of a command to an actuator, for example, then the stimulus could be the reading of a sensor value that has been used in the elaboration of the command for the actuator. Depending on how many tasks are involved in the elaboration of the command and where they are allocated on the platform, the timing chain can (and must) be decomposed in to chain segments that detail how the stimulus leads to the response.

The relation between the sensed value and the applied command results generally from an alternating chain of tasks and messages, where a message is produced by the preceding task and read by the following task:

Figure 54: Example of timing chain at task level

Figure 54 shows the example of a timing chain at task level, where the following temporarily ordered events occur during the transformation of a senor data value into an actuator data value:

1. Task A reads instance *n* of the sensor data value
2. Task A produces instance *m* of Messag1, elaborated from instance *n* of the sensor data
3. Task B reads instance *m* of Message1
4. Task B produces instance *p* of Messag2, elaborated from instance *m* of Message1
5. Task B reads instance *p* of Message2
6. Task B produces instance *q* of the actuator data value, elaborated from instance *p* of Message2

In the above list, we explicitly mention instances of messages and data values, because tasks execute in some recurrent manner and read and write at each execution (potentially) different instances of the

messages and data values. The indices are all the same in the special case where all involved task have the same period and are scheduled in a time triggered manner. The cause and effect relation between a stimulus and a response event exists in a timing chain only when a following task reads the same instance of message than that produced by the previous task in the chain**.** Regarding the relation between the inputs and outputs of the execution of a task, it seems to us reasonable to suppose that the read instance of the input data is always used in the elaboration of the written instance of the output data.

The delay between the writing of a message instance by a previous task and the reading of the message instance by a following task depends on the used (hierarchical) scheduling mechanism and on whether the messages are stored in some local memory or transmitted over on-chip and/or off-chip networks. The later depends on the allocation of the tasks to the Tiles of physical platform. In the following Figure 55 is shown timing chains where all kinds of communications are involved:

- intra-partition task communication (not covered by DREAMS services)
- intra-tile, inter-partition communication (supported by DRAL)
- inter-tile communication over the NOC
- inter-node communication over an off-chip network
- inter-node, inter-cluster communication through an inter-cluster gateway

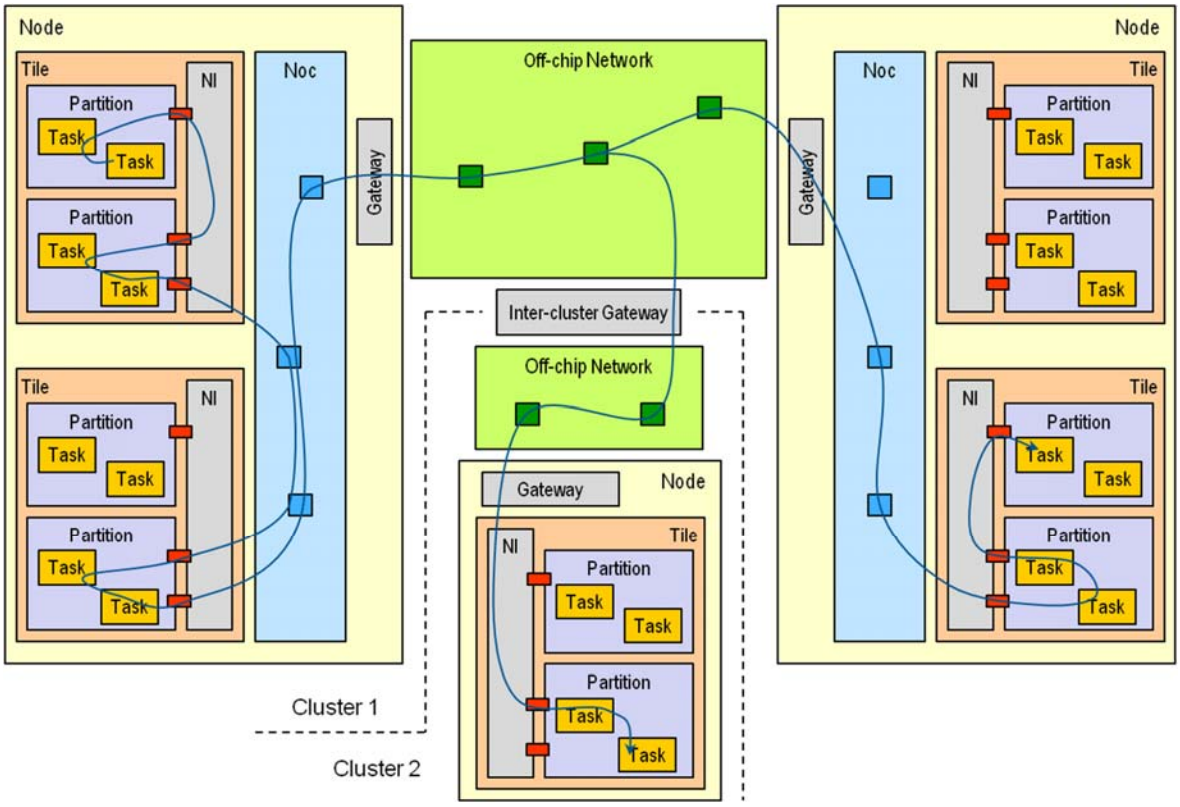

Figure 55 – Illustration of timing chains after deployment

As shown in the figure, the communication may also span an inter-cluster gateway.

Notice that different periods or recurrence pattern of consecutive tasks in the timing chain may also have an influence on these delays.

In general, in order to be able to establish the cause and effect relation between a stimulus and a response, the traces must contain several intermediate events from the timing chain that allow deducing the cause and effect relation. In the initial example we have mentioned already to kinds of such events:

a) A message instance, produced by a task, is ready for being read
b) A message instance has been read by a task

In case of inter-partition communication through the DRAL, a produced message is stored in an output port in the NI and then copied to the input ports of the local destination; thereafter it is ready for being read. In case of intra-partition communication (not covered by DREAMS services), the messages are stored in some memory location, but the event a) must still have the semantic of "ready for being read".

Depending on the granularity of the tasks simulation models, the behaviors of the tasks might be black boxes so that it is not possible to known when input messages are exactly read or output message are exactly written. In that case, an approximation may consist in considering the execution start and end (of recurring tasks) to be the time where the messages are read or written.

If the communications goes across on-chip and/or off-chip networks then more events are required to be able to follow the progress of the transmission and match the instances of the messages on sender and receiver side:

c) A message instance, located in an Network interface port and coming from a tile, is elected for transmission over the NOC by the NI (Bridging of Outgoing Messages)
d) A message instance, arriving from the NOC, is available in the ingress queue of the on-chip/off-chip NI
e) A message instance is elected by the on-chip/off-chip NI (Bridging Service) for transmission over the off-chip network
f) A message instance, arriving from the off-chip network, is available in the ingress queue of the on-chip/off-chip NI
g) A message instance is elected by the on-chip/off-chip NI (Bridging Service) for transmission over the NOC
h) A message instance, arriving from the NOC, has been written by the NI to the corresponding port

If no NOC is present, then additional variant of these events exist:

i) A message instance, located in an NI port and coming from a tile, is elected by the on-chip/off-chip NI (Bridging Service) for transmission over the off-chip network
j) A message instance, arriving from then off-chip network, has been written by the NI to the corresponding NI port

Notice the following important point regarding the way the events are defined: if the event concerns the arrival of a message or a frame, than the event occurs when the message or a frame is available for the next step in the chain. This implies that if in a chain an arrival event occurs at some time t_0 and the following event (selection for sending or reading) occurs at some time $t_1 > t_0$, then it is the same instance of the message that is concerned – unless it is overwritten by the arrival of the next instance. The analysis of the simulation trances is based on this supposed property.

If a time triggered scheduling scheme is used for task execution and message communication, then the occurrence time of the response events and that of the corresponding stimulus event can be predicted. In that case, one could wonder if it is useful to derive the response event for a stimulus event from the intermediate event from the traces. But the goal of the simulation is to detect non-conformances, due to what-ever cause, including errors in the design algorithms/methods that are supposed to produce design solutions that are correct by construction.

8.1.2 Frames discarded by routers because of insufficient memory

It would be interesting to deduce the loss of a frame due to insufficient memory in a router from the missing re-transmission of a frame instance that has arrived at the router, since this approach would potentially allow detect other problems in the design or configuration. It would however be rather difficult to find a reliable criterion for identifying a frame loss, since the departure of a following frame instance can easily be misinterpreted as the transmission of the actually lost frame instance. Furthermore, it would be impossible to distinguish limited memory from other may be not foreseen causes. Therefore the frame transmission events in the traces should:

- contain frame instance identifiers
- contain explicit frame discarding events, due to insufficient memory

8.1.3 Task Response Times

By definition, the “response time of a task” is the delay between the times when the tasks is eligible for execution until the time when its execution ends. In between it might be waiting at some time for getting (again) access to the execution resource. This definition is adapted to the context where task execute periodically or in some other recurrent manner and where an execution start and end can be identified.

In order to be able to analyze task response times, the task execution related traces should

- Contain the following events:
  - tasks instance activation time
  - tasks instance execution end time
- contain task instance identifiers

Notice that in case of time triggered tasks scheduling, tasks are usually only activated when no other task is competing for the execution resource and thus, the “activation time” is equal to the “execution start time”.

Depending on the level of details of simulation model, the scheduling of application task might not be modeled. In that, task response time related traces are not generated and the dnd-to-end statistics are based on the partition execution related events.

8.2 Overall Structure of Trace Files

8.2.1 Layout of a trace file

An efficient and often used structure of trace file is the following:

- every line contains the description of an event
- the line separator is the new line character “\n” (C/C++ or Java)
- the order of appearance of the lines reflect the order of occurrence of the events
- The description of an event consists in
  1. time stamp that corresponds to the date of occurrence
  2. keyword that identifies the type of event
  3. event specific details
- character uses to separate items in a line is the space character

Thus, the structure of a line would be the following:

<TIME> <EVENT_TYPE> … details …

8.2.2 Scope of a trace file

I would be possible to use just one file for all events of the simulation. However,

- the simulator consists of several simulation blocks, which might be implemented as distinct processes. The resulting need of “simultaneous” write access to a same file would either be impossible to realize or lead to a performance bottle neck.
- if the scope of a file is for example only one network, i.e. the file only contains events related to only one network, then it would not be necessary to repeat the identifier of the network in every line, which would reduce the size of the file.

The scope of a trace file should therefore be not larger than one simulation block. The scope of the trace file could be coded into the name of the file.

8.2.3 Identifiers

The trace analysis tool must be able to relate the entities concerned by the events logged in the trace files to those described in the meta-model. In [5] is suggested a name based hierarchical identification scheme that would be independent of the context where the model data is handled (XML file, in memory, etc). This scheme should also be used in the trace files, but if hierarchical names are long then this may lead to large trace files. For the trace files, the hierarchical names should be based on numerical identifiers that are also stored in the DREAMS model file.

8.3 Trace Files and events

8.3.1 Off-chip network related events

For each off-chip network, the relevant events are written to a separate file. The events concern the emission and reception of frames between on-chip / off-chip gateways and inter-cluster gateways. The scope is illustrated in Figure 56:

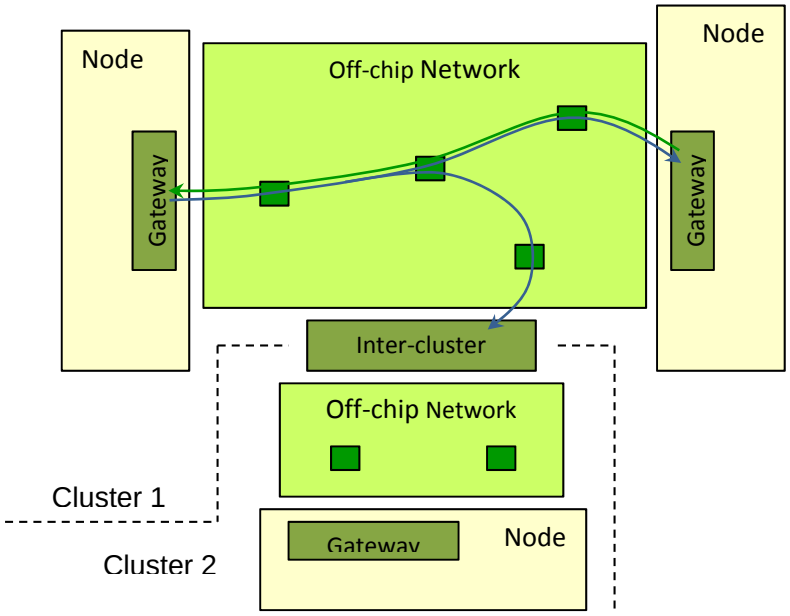

Figure 56: Illustration of off-chip network related events to be traced

Two different kinds of events can be identified. Their trace file syntax is as follows:

<TIME> Frame**Tx** Gateway <GATEWAY_ID> <FRAME_ID> <FRAME_INSTANCE_ID>
<TIME> Frame**Rx** Gateway <GATEWAY_ID> <FRAME_ID> <FRAME_INSTANCE_ID>

With respect to the loss of frames due to insufficient memory in the routers, the following event is defined:

<TIME> FrameDiscarted <ROUTER_ID> <FRAME_ID> <FRAME_INSTANCE_ID>

The semantics are defined in the following table:

FrameTx Gateway	A frame instance is queued for transmission over the off-chip network, as soon as allowed by the protocol. Depending on the traffic class and priority, the frame is either immediately transmitted (periodic) or may wait in a queue (rate-constraint or best effort).
FrameRx Gateway	Frame instance has arrived and the contained messages are ready for being forwarded inside the connected node or over the off-chip network of the connected cluster.
<GATEWAY_ID>	Identifier of the concerned gateway.
<ROUTER_ID>	Identifier of the concerned router.
<FRAME_ID>	Identifier of the concerned frame.
<FRAME_INSTANCE_ID>	Identifier of the frame instance. This identifier is incremented with each “FrameTx Gateway” event.

8.3.2 On-chip network related events

For each on-chip network, the relevant events are written to a separate file. These events concern the emission and reception of messages between NI ports and the on-chip / off-chip gateways. The scope is illustrated in Figure 57:

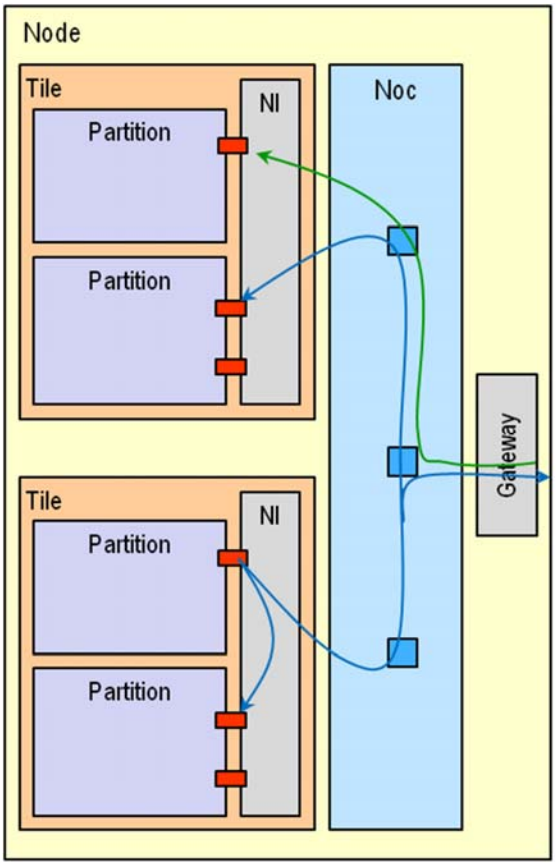

Figure 57: Illustration of on-chip network related events to be traced

Four different kinds of events can be identified. Their trace file syntax is as follows:

<TIME> MessageTx OutPort <PORT_ID> <MESSAGE_ID> <MESSAGE_INSTANCE_ID>
<TIME> MessageRx InPort <PORT_ID> <MESSAGE_ID> <MESSAGE_INSTANCE_ID>
<TIME> MessageTx Gateway <GATEWAY_ID> <MESSAGE_ID> <MESSAGE_INSTANCE_ID>
<TIME> MessageRx Gateway <GATEWAY_ID> <MESSAGE_ID> <MESSAGE_INSTANCE_ID>

The semantics are defined in the following table:

MessageTx OutPort	A message instance, contained in the output port, is emitted. Depending on the location of the destination and the optional presence of a NOC, one or several of the following happen after the event has occured: • copying to destination ports on the same Tile • if NOC: reading by the bridging layer for further processing in order to sent the message over the NOC • if no NOC: reading by the on-chip / off-chip gateway for transmission over the off-chip network

MessageRx InPort	A message instance is ready in the input port for being read by tasks in the related partition.
MessageRx Gateway	Message instance, coming from inside the chip (over the NOC or not) is ready for being put into a frame for sending over the off-chip network.
MessageTx Gateway	Message instance, coming from the off-chip network, is forwarded inside the chip: • if NOC: reading by the bridging layer for further processing in order to sent the message over the NOC • if no NOC: copying to destination ports
<PORT_ID>	Hierarchical identifier of the concerned port, without Node and Cluster part, since the scope of the trace file is limited to a Node.
<GATEWAY_ID>	Identifier of the Gateway.
<MESSAGE_ID>	Hierarchical identifier of the concerned message.
<MESSAGE_INSTANCE_ID>	Identifier of the message instance. This identifier is incremented with each “MessageTx OutPort” event.

8.3.3 Partition execution related events

For each Tile, the relevant events are written to a separate file. These events concern the execution of the partitions according to some plan that guaranties time separation. The scope is illustratedin Figure 58:

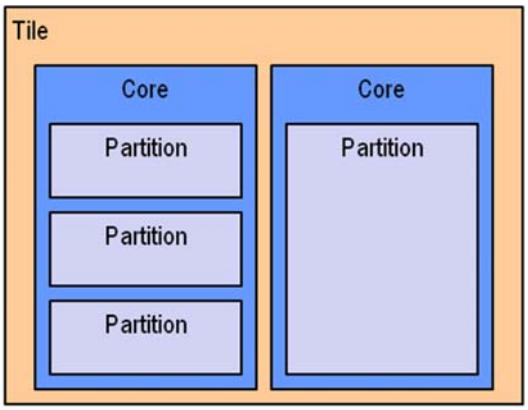

Figure 58 : Illustration of events to be traced that are related to the execution of partitions

Two different kinds of events can be identified. Their trace file syntax is as follows:

<TIME> PartitionSelectedForExecution <CORE_ID> <PARTITION_ID>

<TIME> PartitionSuspended <CORE_ID> <PARTITION_ID>

The semantics are defined in the following table:

PartitionSelectedForExecution	The partition gains (again) exclusive access to a processor core, i.e. is (again) able to execution.
PartitionSuspended	The exaction of the partition ends is suspended.
<PARTITION_ID>	Identifier of the Partition.
<CORE_ID>	Identifier of the processor core.

8.3.4 Task execution related events

For each partition, the relevant events are written to a separate file. These events concern the execution of the tasks according to some scheduling policy. The scope of the trace file is illustrated in Figure 59:

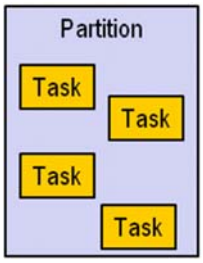

Figure 59: Illustration of events to be traced that are related to the execution of tasks

Four different kinds of events can be identified. Their trace file syntax is as follows:

<TIME> TaskInstantiated <TASK_ID> <TASK_INSTANCE_ID>
<TIME> TaskSelectedForExecution <TASK_ID> <TASK_INSTANCE_ID>
<TIME> TaskSuspended <TASK_ID> <TASK_INSTANCE_ID>
<TIME> TaskInstanceEnded <TASK_ID> <TASK_INSTANCE_ID>

The semantics are defined in the following table:

TaskInstantiated	The recurrent task has been instantiated again, i.e. is ready for a new run, as soon as the task scheduler selects the task for execution.
TaskSelectedForExecution	The task scheduler has selected (again) the task instance for execution.
TaskSuspended	The task scheduler has suspended executuion the task instance.
TaskInstanceEnded	An instance of the recurrent task has arrived at the end of its run.
<TASK_ID>	Identifier of the task.
<TASK_INSTANCE_ID>	Identifier of the task instance. This identifier is incremented with each “TaskInstantiated” event.

8.4 Properties synthesized from the Traces

Depending on the availability of the traces, the following properties can be synthesized:

Observed properties	**Deduced properties**
End-to-end delays	Statistics on the end-to-end delays, for each specified end-to-end latency constraint
Frame losses	Statistics on inter-frame loss time per router and per traffic class.
Task response times	Statistics on task response times

By “statistics” we mean the histogram of observed values, the classical min, average and max statistics and last but not least quantile, i.e. thresholds that are overstepped only with a small probability.

9 Specification of Formal Verification Framework for DREAMS

Mixed Criticality systems are systems where safety, timing, security or reliability are the most important proprieties to verify. In fact such systems have to deliver a response to an event within a certain time period. A failure in such systems may have catastrophic consequences. Such systems are getting increasingly complex, especially with the introduction of multicore platforms. Typical examples of such systems are wearable medical devices, which combine life-supporting functionality with low power wireless connectivity to support remote monitoring by the patient or doctor. Other systems that include multiple real-time functions with different levels of criticality are related to automotive. Formal verification can play an important role in verifying some of properties previously mentioned. In fact formally proving that single hardware or software components are corrects we can try in several cases by composition to prove the correctness of the overall systems.

STNOC is one of the key components of the DREAMS architecture template. As already mentioned DREAMS provides a natural mapping of the current and next generation mixed critical architectures.

In this section we present how to apply formal verification, more specifically property checking, to verify the functionality of the configurable components that are used to build STNoC networks. Although methodology and the framework could be used for all the on-chip and off-chip communication infrastructures we are going to focus to one or two components of STNoC. The components of a STNoC network are first presented and the motivations for using formal verification are discussed.

9.1 STNOC components

Figure 60 shows the main components that are used to build a STNoC network.

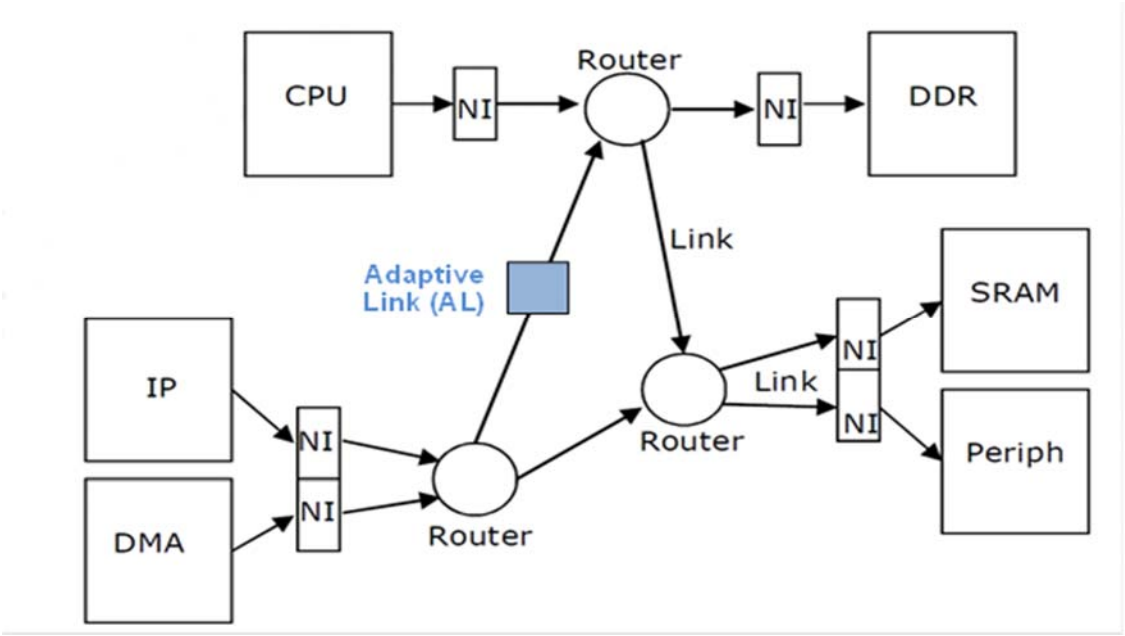

Figure 60 STNOC

Routers are responsible for routing packets to destination based on the contents of their headers and for ensuring the required Quality of Service (QoS) or Best Effort (BE) service. Adaptive Links (AL) are inserted in router-to-router links to break long wires for physical implementation, perform frequency conversion, and flit size downsizing/upsizing. Network Interfaces (NI) perform protocol conversion between ARM AMBA interconnects (AXI, ACE) and the network of routers that use the STNoC protocol.

Two virtual networks are implemented on top of two disjoint virtual channels that share the physical link bandwidth.

STNoC networks are constructed using the iNoC design platform that ST has developed for that purpose. Components are generated based on parameter settings provided by the user via a graphical interface or scripts.

9.2 Motivations for using formal verification

There are two distinct tasks involved in the verification of STNoC components that have to be performed at two different points in time. When designing a component, it must be verified that any configuration of this component that may be generated using the iNoC platform will be free of functional bugs. When using a component to create a NoC that meets the mixed criticality needs of a particular SoC, it must be verified that the configuration that has been generated is free of functional bugs, and that the underline protocol is correct in term of timing.

The second task does not present uncommon challenges, but the first one definitely does due to the extremely large number of potential configurations of the components. Finding a bug or a protocol violation in a component when generating a network for a SoC project could have an adverse impact, especially for mixed critical systems. Correcting the component design could turn out to be a significant effort and a substantial re-verification effort could be required if interactions with other components are impacted by changes. As a result, it could be impossible to avoid delaying some milestones of the SoC project. Therefore, it is critical to verify as many configurations as possible during the design phase of the components, in the amount of time available before the platform gets used in SoC projects.

Using simulation to verify a component means developing a new testbench and a new set of tests for each type of configuration. Constrained random tests have to be used, which implies that adequate means to collect coverage metrics must be developed. Tests generation followed by coverage analysis loops are then required to gain sufficient confidence. However we never reach the 100% confidence. This means that this is an issue especially for Mixed Critical Systems.

We estimated that formal verification is well suited for the mixed criticality systems for several reasons:

- The behavior of a component in its different types of configurations could be expressed with parameterized properties. The parameters used in the iNoC platform to generate a particular component configuration could also be used to configure the properties. A verification environment could then be readily available every time a new component configuration would be generated, with no setup time and effort.
- There would be no need to develop new tests to verify a new configuration. Only machine time would be required, thus saving precious verification resources.
- Exhaustive coverage of the configurations tested could be achieved, which is generally impossible with simulation given the available time and resources.

We have selected the Adaptive Link (AL) as the first component to be tested using this formal verification approach. There were two reasons for making this choice. Firstly, the AL is less complex than the router and Network Interfaces (INI) components which would make it a better vehicle to develop our formal solution and gain experience with it. Secondly, the techniques to be use are quite generic so we can extend to other components . The AL was tested indirectly by inserting it between two routers and generating traffic, which obviously limited the coverage that could be achieved. This means that we cannot use the AL in Mixed criticality system based on STNoC due to lack of full coverage. For this reason thanks to DREAMS project we are introducing a new way to verify mixed critical systems and the outcome of this task can be used by the WP8 demonstrator as wellas other internal ST mixed critical products.

9.3 The Adaptive Link

The Adaptive Link (AL) component has a downstream (DS) interface that receives packets from an upstream component (router, AL, NI) and an upstream (US) interface that sends the packets to a downstream component.

An AL can perform various functions:

- *Relay station*: used to break wires that are too long for physical design. The AL acts as retiming registers.
- *Frequency conversion*: used when the DS and US interfaces operate with two different clocks which are synchronous but have different frequencies.
- *Flit upsizing/downsizing*: used when the flit sizes on the DS and US sides are different.
- *Asynchronous clocks adaptation*: used when the DS and US interfaces operate with two different clocks which are asynchronous. The two clocks may or may not have the same frequency.
- *Store&Forward*: used to store complete packets within the AL before sending them to the downstream component.

Complex configurations can be generated that combine several of these functions. For example, the same AL may implement frequency conversion, flit upsizing and Store&Forward

Like the other STNoC components, the AL supports the two Virtual Networks.

9.4 Modeling of the STNoC protocol with SVA properties

There are essentially three parts in the STNoC protocol:

- The composition rules that specify the structure of a correct packet. The first flits of a packet convey packet routing and QoS/BE service information, and the following flits are either payload flits or ARM AMBA interconnect request/response flits.
- The sideband signals that are associated to flits. For example, a 3-bit signal indicates whether the flit that is present on the physical link is the first flit of a packet, an intermediate flit, or the last flit of a packet.
- The credit based control flow that ensures that an upstream component (router, AL, NI) only sends flits to a downstream component when it is ready to accept them, i.e. when it has space available for them in its input buffer.

A set of SVA properties will be developed to model the communication protocol between an US component and a DS component over the physical link that connects them. This methodology used here can be easily exported to other on-chip and off-chip communication infrastructures

These properties will be parameterized with the iNoC platform parameters that control the generation of STNoC components. Examples of parameters include size of the flit, length of the header, number of byte enable bits within a payload flit, presence/absence of optional sideband signals, etc. Because properties use the same parameters as the iNoC platform, a protocol checker will be readily available every time STNoC components are generated.

Then some RTL code has to be written to keep track of the number of flits that the US component sends to the DS component and of the number of credits that the DS component sends to the US component.

Properties will make use of this RTL code to check that no overflow of the input buffer of the DS component can occur and that the valid/credit round trip delay is always within range. This is very important to insure that the protocol timing used to compute the WCET is the one that we find in the real implementation.

10 Fault injection on Hardware Emulation Flow for DREAMS chip

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 underline emulation technology used we have developed a framework to extend the current hardware emulation flow toward the fault injection support. Fault injection is important to evaluating the dependability (the study of failures and errors) of a System on Chip. The dependability of a system is the study 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 occurs according 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 critical system where the timing is critical. On the other side, testing a prototype will allow us to evaluate the system without any assumption yielding towards more accurate results. In fact in prototype based fault injection methodology faults are injects using random inputs enabling a better understanding of the effectiveness of fault tolerance mechanisms as well as concrete number 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 we 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 to consider a prototype we consider a fully operational SoC therefore fault injection provides information about the real-life failures. Injecting random fault 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.

10.1 The Fault Model

In this section we describe a new methodology to validate the functionalities and the robustness against soft errors of a DREAMS Chip instances. As described in [7] 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. 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. There are several fault models however in this chapter we consider only the following models:

- SEU: Single Event Upset

- flip the value of a flip-flop (0→1, 1→0)
- value remain corrupted until next write in register

- STUCK-AT 0/1 (STK)
  - stuck the value of a flip-flop to 0/1 permanently
- MBU: Multiple Bit Upset
  - flip the value of several (1→*n*) flip-flop
  - 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
  - value in cells remains corrupted until next write in memory cell
  - simulate physical propagation of error

10.2 The fault injection DREAMS 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. Neutrons originated by cosmic rays, alpha particles originated by the natural emission of the molding composing 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. The figure below show a generic Fault injection environment which typically consists of the DUT (aka SoC) plus a fault injection library, a workload generator, workload library , controller, monitor, data collector and data analyzer. The fault injection injects faults into the emulated SoC and executes the software from the workload generator (baseline software with applications). The monitor tracks the execution while the data collector tracks data at runtime. The data analyzer performs the data analysis. The controller is a software that run on a separate computer and control the overall framework, for this reason is often referred as software controller or software host controller

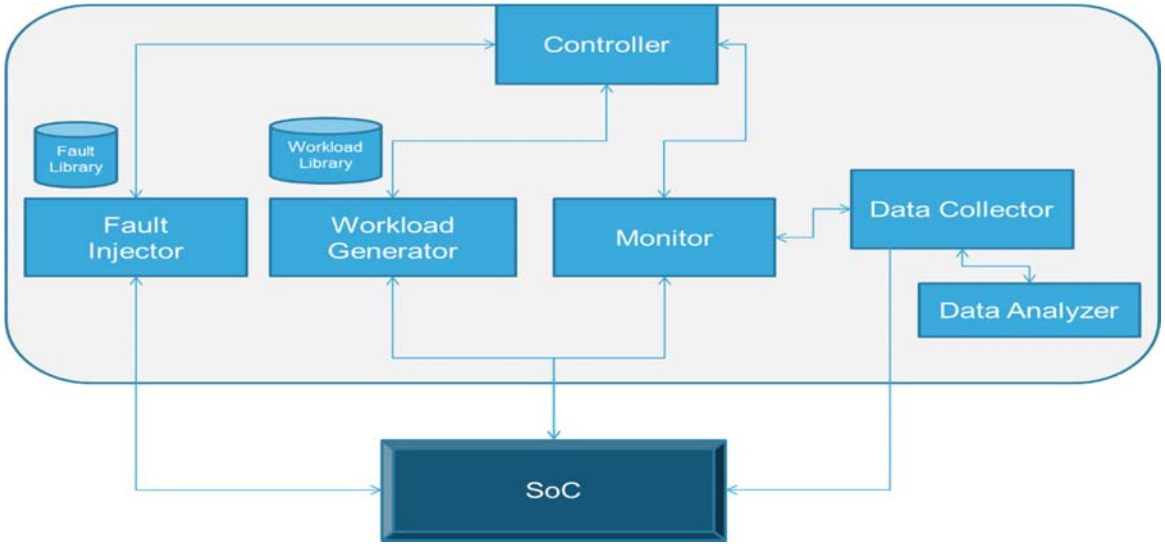

Figure 61 Fault injection Framework

The fault injection emulation framework developed in DREAMS is composed of several DREAMS hardware blocks (synthesized on the hardware emulator platform) and 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 debug 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 core 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 receive/send fault data/fault results to the software host controller through the so called SCEMI-API interface. Thanks to 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 62. 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.

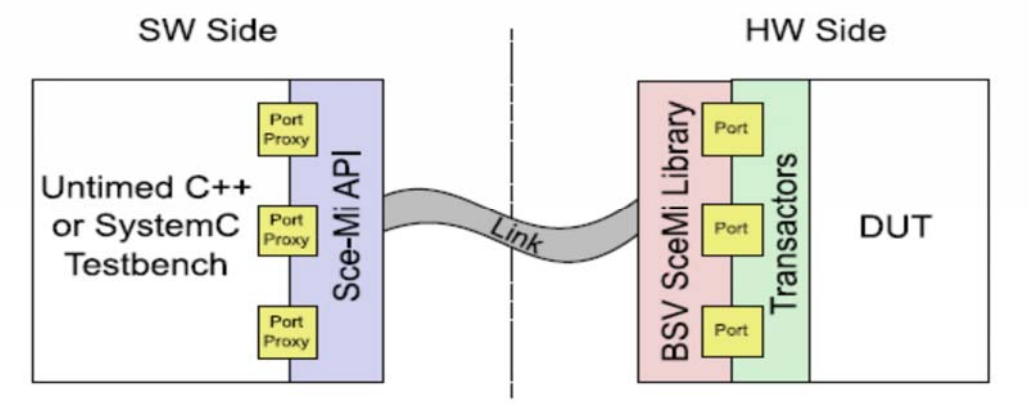

Figure 62 SCEMI API

In order to speed up the fault injections coverage an extra hardware controller is added to the DUT as shown in Figure 63. This hardware component, still controlled by the software controller enable to run in parallel different scenario of fault injections reducing the time to reach the target level of fault coverage.

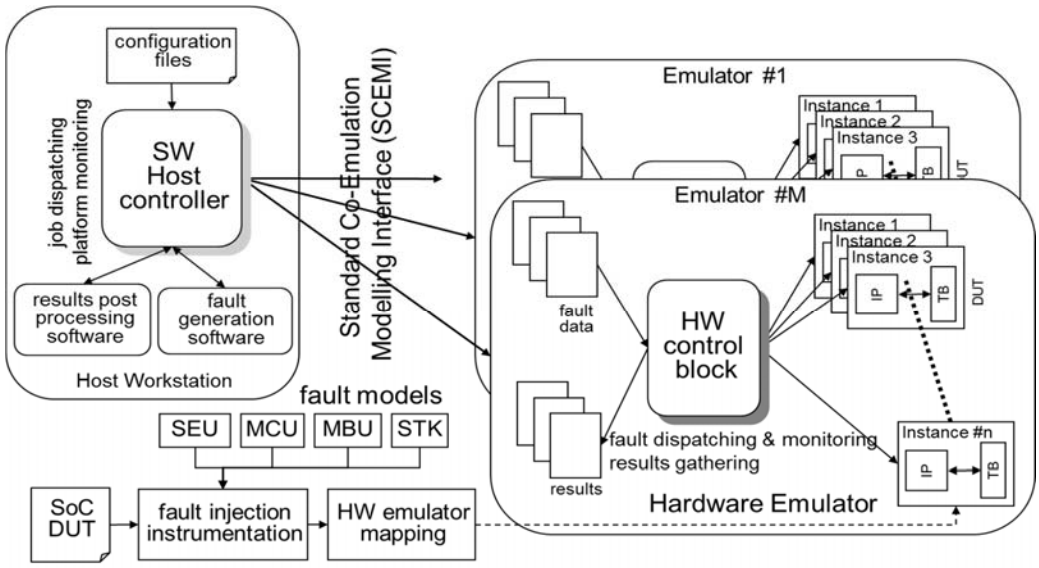

Figure 63 Emulation Platform

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 has completed. The software controller is multithreaded and is able to control several hardware emulators in parallel. An example of DUT is the architecture of STNoC defined in DREAMS for the harmonized platform. Using the Hardware controller is possible to run in parallel different fault injection instances testing different fault models. For instance, we can think to have two instance of the DUT in which we inject faults to the request network (Instance 1) and response network (instance 2)

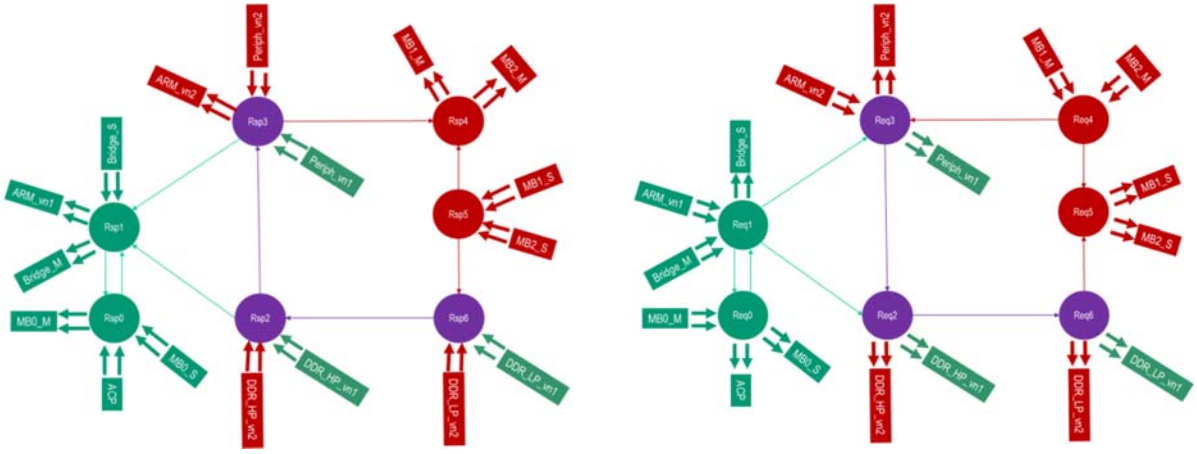

Figure 64 STNOC DREAMS instance

The emulation based fault injection framework is well suited for system such as the mixed critical ones that require high time resolution.

11 Bibliography

[1] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. -v. Peetz and S. Wolf, "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," in *In Proceedings of the 8th International Modelica Conference*, 2011.

[2] "IEEE standard for local and metropolitan area networks: Media Access Control (MAC) bridges," IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998).

[3] S. standard, "{AS}-6802 -- {Time-Triggered Ethernet}," 2011.

[4] I. E. E. COMMITTEE, "AIRCRAFT DATA NETWORK PART 7 AVIONICS FULL DUPLEX SWITCHED ETHERNET ({AFDX}) NETWORK," 2005.

[5] D. Consortium, "Architectural Style of DREAMS," 2014.

[6] Fent Innovative Software Solutions, "User Manual - XtratuM hypervisor for ARM Cortex-A9," 2015.

[7] "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Register-transfer_level.

[8] I. Standard, "IEEE Standard for Local and metropolitan area networks: {Media Access Control (MAC)} Bridges," 2004.

[9] M. Abuteir and R. Obermaisser, "Simulation Environment for Time-Triggered Ethernet," in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*, 2013.