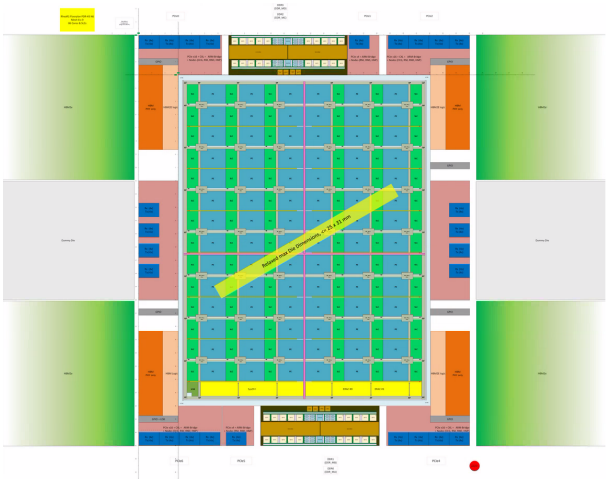
Correlation of Rhea EPI processor simulation models for functional verification

PFE Report

Bastien HUBERT



ENSTA Paris, May - Sept 2023

NDA and confidentiality

The EPI is a project that has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 800928. Please see `http://www.european-processor-initiative.eu/` for more information. The content of this document is the result of extensive discussions within the EPI © Consortium as a whole.

Table of Contents

Terms and Abbreviations

Abbreviations and definitions are relative to this document and may vary from other literature

3DS-RDIMM	3D-Stacked RDIMM	
ACE	AXI Coherency Extensions	an ARM IP
AMBA	Advanced Microcontroller Bus Architecture	an ARM IP
AP	Application Processor	an element of a PE
APB	Advanced Peripheral Bus protocol	an ARM IP
ATB	Advanced Trace Bus protocol	an ARM IP
AXI	Advanced eXtensible Interface	an ARM IP
CC	Core Cluster (or Processing Cluster)	one of Rhea's subsystems
CHI	Coherent Hub Interface	an ARM IP
CI	Continuous Integration	a distributed development paradigm
CMN	Coherence Mesh Network architecture	an ARM IP
C-NoC	Control NoC	one of Rhea's buses
CSR	Control and Status Register	an element of a processor
CXL	Compute eXpress Link	an ARM PCIe configuration
CXS	Credited eXtensible Stream protocol	an ARM IP
DDR	Double Data Range SDRAM	one of Rhea's subsystems
DIMM	Dual Inline Memory Module	
DPI	Direct Programming Interface	a communication paradigm for hardware design
D-NoC	Data NoC	one of Rhea's buses
DR	hardware Design Requirement	a Sipearl specification term
DSU	DynamIQ Shared Unit	an element of a PE, an ARM IP
DUT	Device Under Test	a verification paradigm
DVM	Distributed Virtual Memory protocol	an ARM IP
EP	End Point	an ARM PCIe configuration
EPI	European Processor Initiative	
ERAC	EPI Rhea ACcelerator	one of Rhea's subsystems
FLOPS	FLoating point Operations Per Second	a performance metric for processors
FSM	Finite State Machine	a mathematical model
GIC	Generic Interrupt Controller	an ARM IP
GPP	General Purpose Processor	
HAS	High-level Architecture Specification	a Sipearl specification document
HBM	High Bandwidth Memory	one of Rhea's subsystems
HDA	Hierarchy Dependency Analyser	a Sipearl compilation tool
HDL	Hardware Description Language	a type of programming language for hardware design
HLS	High-Level Synthesis	
HN-F	Fully-coherent Home Node	an ARM IP

HN-I	Non-Coherent Home Node	an ARM IP
HPC	High Performance Computing	
IoU	Intersection over Union	a measure of the similarity between multiple sets
IP	Intellectual Property	
JTAG	Joint Test Action Group	a hardware verification standard
LRDIMM	Load-Reduction DIMM	
MAS	hardware Micro-Architecture Specification	a Sipearl specification document
MT	Mesh Tile	an element of the CC
NI	Non-coherent Interconnect architecture	an ARM IP
NoC	Network-on-Chip	a hardware architecture paradigm
NTX	Neural network Tensor Accelerator	an element of a STX, an EPI IP
PCIe	Peripheral Component Interconnect express	one of Rhea's subsystems
PE	Processing Element	an element of a MT
Phy	Physical layer between controller and pins	
PLL	Phase-Locked Loop	an element of a MT, an ARM IP
PPU	Power Policy Unit	an element of a MT, an ARM IP
RDIMM	Registered DIMM	
RISC	Reduced Instruction Set Computer	a hardware architecture paradigm
RN-D	DVM Request Node	an ARM IP
RP	Root Port	an ARM PCIe configuration
RTL	Register Transfer Level	a type of processor model
SAD	Software Architecture Document	a Sipearl specification document
SDRAM	Synchronous Dynamic Random-Access Memory	
SEC	Sequential Equivalence Checking	a formal verification paradigm
SF	Snoop Filter	an element of a MT, an ARM IP
SLC	Single-Level Cell	an element of a MT, an ARM IP
SLM	System-Level Model	a type of processor model
SPU	Stencil Processing Unit	an element of a STX, an EPI IP
SR	System Requirement	a Sipearl specification term
STX	Stencil/Tensor Accelerator	an element of ERAC, an EPI IP
SWR	Software Requirement	a Sipearl specification term
SYSCTRL	System Control	one of Rhea's subsystem
TLM	Transaction-Level Modelling	a communication paradigm for hardware design
TMB	Tile Management Block	an element of a MT, an Atos IP
UVMC	Universal Verification Methodology Connect	a communication paradigm for hardware design
VP	Virtual Prototype	a type of processor model
VR	Verification Requirement	a Sipearl specification term
VRP	VaRiable Precision accelerator	an element of ERAC, an EPI IP

List of Figures

1 Introduction

As processor complexity escalates and design margins narrow, it becomes essential to ensure the correctness and efficiency of complex architectures before they are fabricated, saving valuable resources and time. The verification process aims to establish a clear understanding of how the hardware behaves under various workloads, ensuring its adherence to design specifications and identifying potential functional or performance bottlenecks. This research focuses specifically on the simulation models, a fundamental component of the verification process, which allows the investigation of the processor's behaviour in a virtual environment.

This internship aims to describe and improve the verification paradigm used on a concrete example from the industry. Understanding the correlation between architectural models empowers hardware architects to make informed decisions during the design process, leading to more a reliable and efficient design. Additionally, the insights gained from this study can influence future simulation methodologies, refining the accuracy of models and enhancing the overall verification process.

The subsequent sections of this document will delve into the theoretical underpinnings of the Rhea EPI processor architecture, the intricacies of simulation models, the verification methodologies employed, and the systematic approach to analysing and interpreting the results. By the end of this study, we envision a more comprehensive understanding of the correlation between hardware simulation models, paving the way for a new era of high performance, energy-efficient, and reliable processor designs.

Keywords: hardware verification, equivalence checking, SystemVerilog, SystemC, cosimulation, interlanguage communication, DUT

2 Context

2.1 The European Processor Initiative

The **European Processor Initiative** [1] (EPI) is a European project that targets EU independence in high-performance computing (HPC) microprocessor technologies and infrastructure. Launched in 2019, its objective is to design and implement a roadmap for a new family of low-power European microprocessors dedicated to exascale computing (the ability to calculate at least 10^{18} FLOPS), high-performance big-data, and a range of emerging applications within a 20MW power budget. The EPI is supported by a consortium of 30 partners originating from 10 European countries who ensure that the key competence of high-end microprocessor design remains in Europe. The consortium includes a right balance between academia and industry, and covers the whole value chain among which research institutes, supercomputing centres, computing, electronics and automotive industrials. To this day, it consists of:

- BMW Group (Germany)
- Barcelona Supercomputing Center (Spain)
- CEA (France)
- Chalmers University of Technology (Sweden)
- CINECA (Italy)
- E4 Computer Engineering (Italy)
- Elektrobit (Germany)
- ETH Zürich (Switzerland)
- Eviden (France)
- Extoll (Germany)
- Faculty of Electrical Engineering and Computing, University of Zagreb (Croatia)
- Forschungszentrum Jülich (Germany)
- Foundation for Research & Technology - Hellas (Greece)
- Fraunhofer (Germany)
- GENCI (France)
- Infineon (Germany)
- Kalray (France)
- Kernkonzept (Germany)
- Karlsruhe Institute of Technologies (Germany)
- Leonardo (Italy)

- Menta (France)
- Prove & Run (France)
- Semidynamics (Spain)
- Sipearl (France)
- STMicroelectronics Italy (Italy)
- SURF (Netherlands)
- Técnico Lisboa (Portugal)
- University of Bologna (Italy)
- University of Pisa (Italy)
- ZeroPoint Technologies (Sweden)

The EPI is a part of the European High Performance Computing Joint Undertaking (EuroHPC JU) and is currently in its second implementation phase during which it will continue the initial developments of Phase 1 on European microprocessors and accelerators to support European technological autonomy and sovereignty in this critical area. Based on a solid, long-term economic approach, the EPI will deliver central components of future European supercomputers to tackle societal challenges and boost innovation and the digital transformation of the European economy and science. The specific focus of the second phase is to finalise the development of the first generation of low-power microprocessor units and accelerators, enhancing existing technologies to target the incoming European Exascale machines and paving the way for industrialisation and commercialisation of these technologies.

2.2 Sipearl

Created in June 2019, **Sipearl** [2] is the private company in charge of bringing to life the EPI project by designing a European high-performance and low-power microprocessor, which will be used to achieve European Exascale computing power. It works in collaboration with its 29 partners from the EPI consortium from both scientific communities and industrial sectors. Composed of more than 130 employees dispatched across 6 locations in France, Germany and Spain, Sipearl is currently developing its first microprocessor : Rhea.

2.3 Internship progress

Different sites each have their expertise, and the German team focuses on frontend hardware development. For this reason, this internship was conducted in Duisburg, while maintaining a close relationship with the main office in France and the verification team in France and Spain. The Duisburg team consists of 20 employees, work students and interns working under the direction of a site manager, and reports directly to the main office in France.

Because Sipearl is at the centre of a major European challenge, many precautions have been taken to prevent security breaches of any kinds. The price to pay for that security is that obtaining credentials to access or modify

data is a very slow process involving many different parties. It took more than one month be be granted access to the main source of documentation, and constant communication with other team members across all sites has been maintained all along this internship to keep information aligned. Furthermore, some information regarding the verification process is not described in Sipearl's private resource, so meetings with more experienced employees have been conducted to acquire the proper training for Sipearl-exclusive tooling.

Due to the inherent flexibility that comes with its startup status, Sipearl is regularly subject to restructuring, even more so with its current intensive recruiting policy. In fact, the company has undergone an important reorganisation of its branches and management teams within the first month of this internship. This impacted every site, as well as the course of this internship, and forced two realignment meetings 4 and 6 weeks into the internship.

3 Verification Paradigm

3.1 Processor Models

A processor **model** M is a **Finite State Machine** [3] (FSM) with no ending states:

$$M \stackrel{def}{=} \{\Sigma,\ S,\ s_{init},\ \delta\} \tag{1}$$

where Σ is a finite non-empty set called the **alphabet**, S is a finite non-empty set of **states**, $s_{init} \in S$ is the **initial state**, and $\delta : S \times \Sigma \to S$ is the **transition function** of the model.

From each state $s \in S$ the model can accept an input $\sigma \in \Sigma$ to reach a new state $s' = \delta(s, \sigma) \in S$. Doing so recursively allows us to create a n**-path** ($n \in \mathbb{N}$) $\hat{\sigma} \in \Sigma^n$ to connect a starting state $\hat{s}_0$ with an ending state $\hat{s}_1$:

$$\hat{s}_0 \underset{M,\hat{\sigma}}{\rightsquigarrow} \hat{s}_1 \stackrel{def}{=} \exists\,(s_0,\ s_1,\ \cdots,\ s_n) \in S^n \mid \begin{cases} s_0 = \hat{s}_0 \\ s_n = \hat{s}_1 \\ \forall k \in [\![1, n]\!],\quad s_k = \delta(s_{k-1},\ \hat{\sigma}_{k-1}) \end{cases} \tag{2}$$

Since δ is a function, it can be recursively proven that $\forall \hat{s}_0 \in S$, $\forall n \in \mathbb{N}$, $\forall \hat{\sigma} \in \Sigma^n$, $\exists!\, \hat{s}_1 \in S \mid \hat{s}_0 \underset{M,\hat{\sigma}}{\rightsquigarrow} \hat{s}_1$. Thus, we can define the **macro-transition function** of M:

$$\hat{\delta} : \begin{cases} S \times \bigcup_{n\in\mathbb{N}} \Sigma^n \to S \\ (\hat{s}_0,\ \hat{\sigma}) \mapsto \hat{s}_1 \mid \hat{s}_0 \underset{M,\hat{\sigma}}{\rightsquigarrow} \hat{s}_1 \end{cases} \tag{3}$$

n.b. to simplify reading, we will use the convention $E^{(\mathbb{N})} \stackrel{def}{=} \bigcup_{n\in\mathbb{N}} E^n$.

With this new function, we can define a binary equivalence relation between two models M_1 and M_2 as [4]:

$$M_1 \sim M_2 \stackrel{def}{=} \begin{cases} S_1 = S_2 \\ \Sigma_1 = \Sigma_2 \\ \hat{\delta}_1 = \hat{\delta}_2 \end{cases} \tag{4}$$

However, since $\Sigma \subseteq \Sigma^{(\mathbb{N})}$, we can see that using only 1-paths in $\hat{\delta}$ proves that:

$$M_1 \sim M_2 \implies \delta_1 = \delta_2 \tag{5}$$

Two equivalent models can thus only differ in their initial state. While this definition is too restrictive, it is the cornerstone to define other, more useful, relations. For instance, a model M_1 is a **submodel** of a model M_2 if:

$$M_1 \preceq M_2 \stackrel{def}{=} \begin{cases} S_1 \subseteq S_2 \\ \Sigma_1 \subseteq \Sigma_2 \\ \hat{\delta}_1 = \hat{\delta}_{2\ \mid S_1 \times \Sigma_1^{(\mathbb{N})}} \end{cases} \tag{6}$$

3.2 Hierarchy of Requirements

When designing the Rhea processor, a series of **requirements** must be respected to ensure the processor functions correctly and accordingly to specific and clearly-defined rules. The first Rhea model, called **System-Level Model** (SLM) we will study in this document is directly extracted from those requirements and can be thought of as a reference model. As such, every other model will try to be equivalent to the SLM.

It is most important that the SLM be clearly defined as early as possible in the development of Rhea so that subsequent models can be efficiently aligned with it. To achieve this, Sipearl has opted for a recursive approach: from global **System Requirements** (SR), we can derive more precise hardware **Design Requirements** (DR) and **Software Requirements** (SWR), which can in turn be broken down into **Verification Requirements** (VR).

These requirements are found in three distinct documents:

- The **High-level Architecture Specification** (HAS) is the document describing the SRs. It is divided into two main sub-documents: the System HAS that specifies SRs from an system engineering point of view, and the Toplevel HAS, specifying SRs from the chip architecture point of view. Both views are complementary and the total forms a coherent set of SRs from which the DRs and the SWRs can be defined.
- The hardware **Micro-Architecture Specification** (MAS) is one of the two documents derived from the HAS, and focuses on DRs. It contains hardware specifications on Rhea and each of its subsystems, and provides documentation on security, booting and system configuration. It is the main source of information for hardware engineers, and helps describing the hardware VRs.
- Likewise, the **Software Architecture Document** (SAD) is the second document derived from the HAS, and is the SWR counterpart of the MAS. It describes software specification for Rhea and its subsystems, and is used by firmware and software engineers. In view of the subject of this document, the use of such a resource was not necessary, and will therefore not be described any further.

VRs can be found in the Verification tabs of the MAS and the SAD, and describe a precise verification item related to a DR or a SWR. Figure 1 shows an example of VR extracted from the MAS.

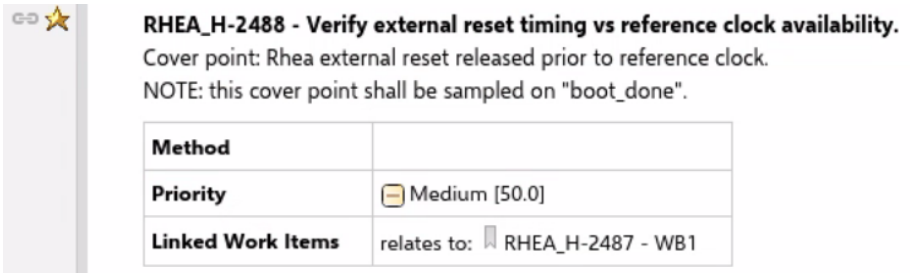


Figure 1: An example of Verification Requirement

In order efficiently manage the hierarchy of requirements for Rhea, Sipearl has chosen to use **Polarion** [5], an application lifecycle management system. It is the central database for specification documentation, and is able to manage technical requirements on different levels. These requirements can be dynamically linked based on

their relationship and individually reviewed and approved. Figure 2 presents the Sipearl specification flow used for Rhea. Throughout this document, every figure about Rhea's architecture has been extracted from Sipearl's Polarion database.

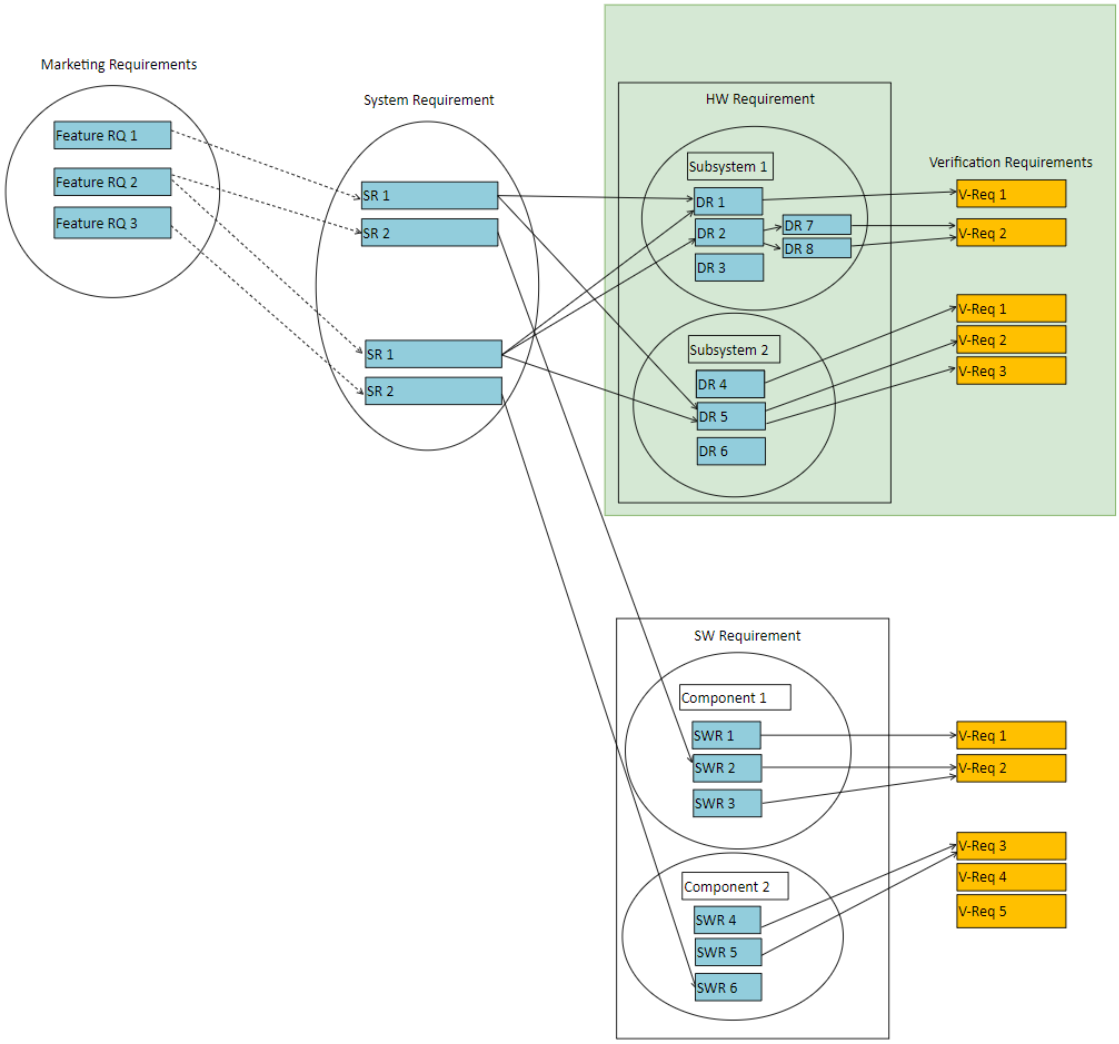


Figure 2: Specification flow for the Rhea project

Polarion also allows to perform test planning, automated test execution and issue tracking. It implements a versioning tracking feature, allowing for release and iteration planning, impact analysis and resource and time management for each version. Figure 3 shows an instance of the top-level view of the Rhea project.

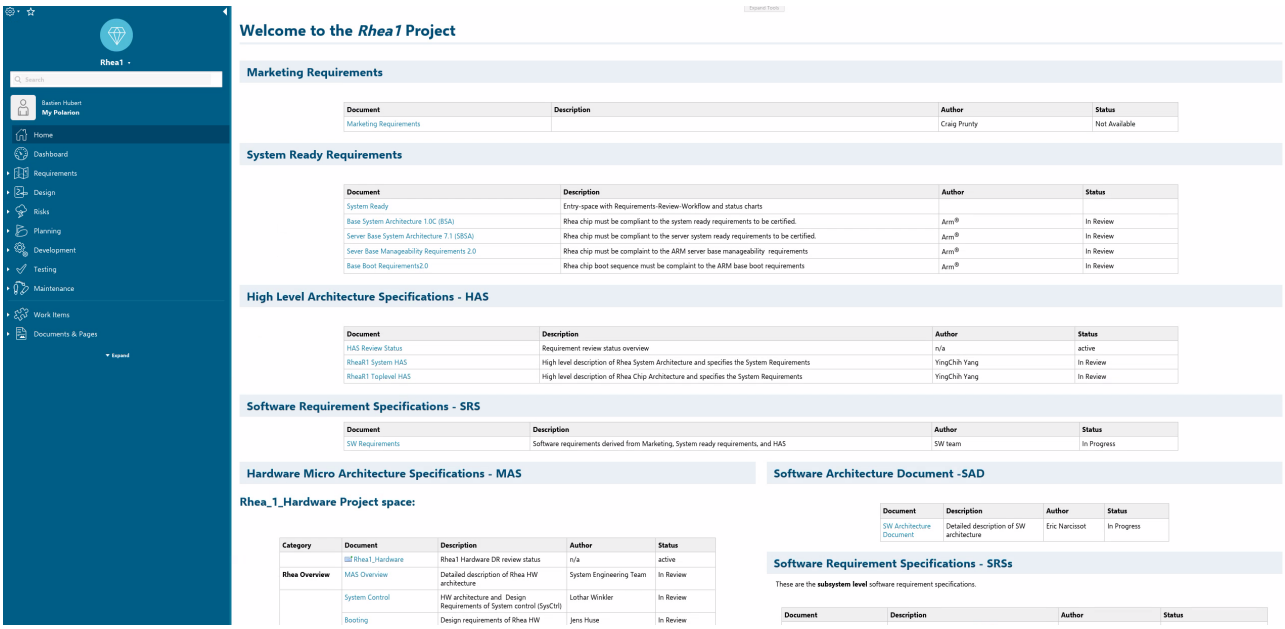


Figure 3: Polarion interface for the Rhea project

3.3 Coverage Items

A **coverage item** is an atomic verification item of a VR, and is used as an indicator of the coverage and verification rates of a VR by a verification protocol. Figure 4 presents a list of icons used in Polarion for verification purposes, including the different types of coverage items used in the Rhea project.

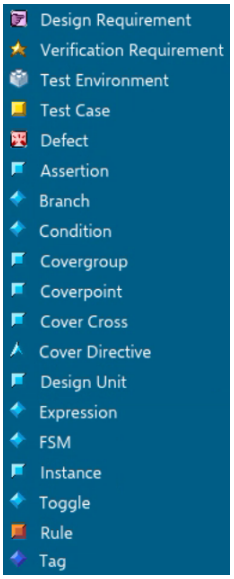


Figure 4: Verification icons legend

Mathematically, a coverage item is defined as a triplet $(\hat{s}_0, \hat{\sigma}, \hat{s}_1)$ such that, for an ideal, target model M^*:

$$\hat{s}_0 \underset{M^*,\hat{\sigma}}{\rightsquigarrow} \hat{s}_1 \tag{7}$$

The SLM is thus defined as such:

$$SLM \stackrel{def}{=} \{\Sigma_{SLM},\ S_{SLM},\ s_{init},\ \delta_{SLM}\} \tag{8}$$

where Σ_{SLM} is the set of all $\hat{\sigma}$ from each coverage items, S_{SLM} is the set of all $\hat{s}_0$ and $\hat{s}_1$ from each coverage items, s_{init} is an arbitrary initial state (usually the powered-down state), and δ_{SLM} is the function defined as:

$$\delta_{SLM} : \begin{cases} S_{SLM} \times \Sigma_{SLM} & \rightarrow & S_{SLM} \\ (\hat{s}_0,\ \hat{\sigma}) & \mapsto & \hat{s}_1 \mid (\hat{s}_0,\ \hat{\sigma}, \hat{s}_1) \text{ is a coverage item} \end{cases} \tag{9}$$

Since a coverage item is a goal to reach, we can define the following relation for every model M:

$$M \text{ verifies } (\hat{s}_0,\ \hat{\sigma}, \hat{s}_1) \stackrel{def}{=} \hat{s}_0 \underset{M,\hat{\sigma}}{\rightsquigarrow} \hat{s}_1 \tag{10}$$

By definition, the SLM verifies every coverage item.

It is important to understand that, while δ_{SLM} is the transition function of the SLM, it acts as a *macro*-transition function according to 3 and 10. This means that the verification goal is not to make models equivalent to the SLM, but rather to adapt them so that the SLM is a submodel of every model we want to verify.

3.4 Simulation-based Sequential Equivalence Checking

Traditionally, there are three verification paradigms that use coverage items to ensure a hardware design is aligned with the SLM: **formal verification**, **simulation** and **hybrid**. While formal verification methodologies are well know techniques in research and in the industry [6] [7], they are complex methods that require important computation power for large designs due to the explosion of the state-space, so they cannot be used as such for Rhea. On the other hand, pure simulation-based verification methods such as testbenches [8] or blackbox and whitebox testing [9] [10] also require tremendous computation capabilities to simulate large-scale designs, and their verification power is highly dependent on the coverage power of human-written testcases.

Hybrid paradigms [11] consist of using the best of the two paradigms and adapt them based on the situation. For Rhea, Sipearl has opted for an hybrid verification paradigm called **simulation-based Sequential Equivalence Checking** (SEC) [12]. Namely, it uses testcases to evaluate coverage items as if they were sequential compare points from the formal sequential equivalence checking [13] [14] paradigm.

SEC is a formal equivalence checking method used to compare models that have different internal mechanisms but are supposed to behave the same way to the same inputs [15]. It is particularly useful to assert the equivalence of models of different abstraction levels, such as the ones later presented in section 5.1. To assert the equivalence of two FSMs, SEC creates a **product machine** by connecting both FSMs' inputs together, and their outputs to a not-exclusive-or (XNOR) gate. Figure 5 [13] shows two FSMs (a) and their associated product machine (b). States in the product machine whose output is 0, meaning that the original FSMs give different values for the same output, are shaded and should never be entered.

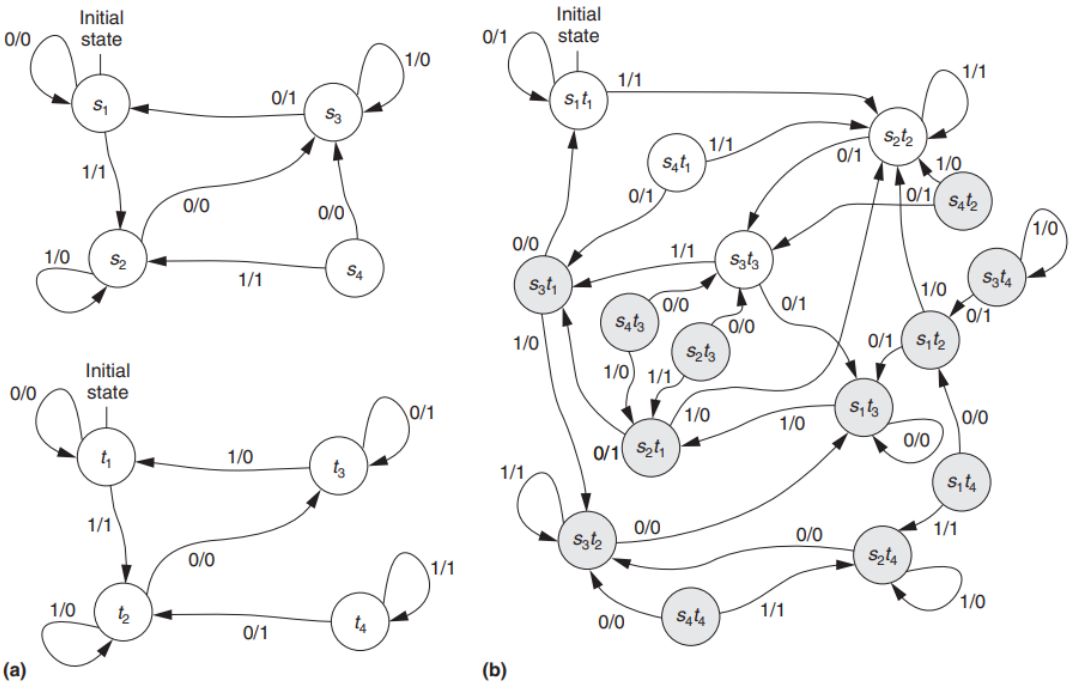


Figure 5: Two FSMs and their associated product machine

"Equivalence" between the two FSMs is established if the output of every reachable state of the product machine is 1. Formally, it means that the restrictions of each FSM to their reachable states and transitions, which are submodels of their respective original model, are equivalent.

Two issues emerge from this technique:

- the two FSMs need to have the inputs and outputs (but not necessarily the same states or transitions), or the product machine cannot be built
- the number of states in the product machine is the product of the number of states in the two original FSMs: $\#M_{prod} = \#M_1 \times \#M_2$

The first issue has already been mentioned in section 3.2 and will be further discussed in section 5.2. It is of crucial importance in the early stages of developing models based on the SLM, because it can introduce misalignments between the different models we want to verify, and greatly complicates the verification process. It is one of Rhea's key challenges on which Sipearl is currently focusing.

While the first issue can be solved by a careful development of each model, the second is more inherent to the SEC paradigm. However, it can be mitigated by two distinct techniques:

- partitioning the state-space of the SLM and asserting the equivalence between the restricted models separately
- finding meaningful **sequential compare points** (*i.e.* moments in time where the two models behave the same way relatively to a transition) and only asserting the equivalence on the models restricted to the compare points in order to avoid taking other states into account

The first technique will be discussed in section 5.2, while the second is precisely the object of simulation-based SEC: by using simulations instead of formal verification techniques, it is possible to bypass the verification of some "minor" states that do not impact the system-level behaviour of the model. For Rhea, Sipearl has heavily leaned into this technique by always comparing a model to the SLM, which, by definition, only describes "meaningful" states. This means that the sequential compare points used in simulation-based SEC are the coverage items from section 3.3.

3.5 Testplans and Testcases

A **testplan** describes a set of configurations and inputs for a simulation with the aim of verifying one or more VRs. Mathematically, it is a finite sequence of couples $T \overset{def}{=} \left(\hat{\sigma}^{(k)}, \hat{s}_k\right)_{k\in[\![1,N]\!]}$, such that:

$$\begin{cases} \forall k \in [\![1, N]\!], \hat{s}_{k-1} \underset{SLM,\hat{\sigma}^{(k-1)}}{\rightsquigarrow} \hat{s}_k \\ \text{with the convention } \hat{s}_0 = s_{init} \end{cases} \tag{11}$$

Figure 6 gives an example of testplan used to verify an architectural aspect of Rhea.

MT_TOP-1.1-9.1, In Progress - [PE_TMB] Verify that local PE timers' count values are programmable

Check that local PE timers' count values are programmable.

ID	RHEA_H-5477
Goal %	100.0
Coverage %	100.0
Linked Work Items	relates to: RHEA_H-5472 - [PE_TMB] Verify local PE timers' count values , verifies: RHEA_H 1122 - The local PE timers have a programmable count value. When the value expires it g... , links: RHEA_H 7425 - *hello_zeus__hello_world*

Figure 6: An example testplan

Unfortunately, we cannot guarantee in advance that a model will follow the same sequence of states as the testplan. We need to define **testcases** to represent instances of testplans that are allowed to diverge from the SLM while still respecting the sequencing imposed by the macro-transition function of a processor model. For a given model M, a testcase is a finite sequence of couples $T_M \overset{def}{=} \left(\hat{\sigma}^{(k)}, \tilde{s}_k\right)_{k\in[\![1,N]\!]}$, such that:

$$\begin{cases} \forall k \in [\![1, N]\!], \tilde{s}_{k-1} \underset{M,\hat{\sigma}^{(k-1)}}{\rightsquigarrow} \tilde{s}_k \\ \text{with the convention } \tilde{s}_0 = s_{init} \end{cases} \tag{12}$$

Since testplans and testcases only differ by the macro-transition function used to define them, it can be easily proven that, for every model M, the following function is bijective:

$$R_M : \begin{cases} \{\text{testplans}\} & \to & \{M\text{-testcases}\} \\ \left(\hat{\sigma}^{(k)}, \hat{s}_k\right)_{k\in[\![1,N]\!]} & \mapsto & \left(\hat{\sigma}^{(k)}, \tilde{s}_k\right)_{k\in[\![1,N]\!]} \end{cases} \tag{13}$$

This function is called the **realisation** function of M.

Testplans and testcases are designed to verify VRs by evaluating all of their coverage items. Specifically, for every model M and coverage item c:

$$\text{A testplan } T \text{ contains } c \overset{def}{=\!=} \exists\, k \in [\![1, N]\!] \mid c = \left(\hat{s}_{k-1}, \hat{\sigma}^{(k)}, \hat{s}_k\right) \tag{14}$$

$$\text{A testcase } T_M \text{ evaluates } c \overset{def}{=\!=} R_M^{-1}(T_M) \text{ contains } c \tag{15}$$

$$T_M \text{ verifies } c \overset{def}{=\!=} \exists\, k \in [\![1, N]\!] \mid c = \left(\tilde{s}_{k-1}, \hat{\sigma}^{(k)}, \tilde{s}_k\right) \tag{16}$$

$$T_M \text{ verifies } c \implies M \text{ verifies } c \text{ (with respect to 10)} \tag{17}$$

This last implication proves that simulation-based sequential equivalence checking can mathematically prove the equivalence between a model and the SLM, provided testcases proves every coverage item. Since 4 describes a binary equivalence relation, proving that two models M_1 and M_2 are equivalent to the SLM automatically proves their restrictions to the SLM are equivalent:

$$\begin{gathered} \forall c \text{ a coverage item}, \forall M \text{ a model}, M \checkmark c \overset{def}{=\!=} \exists\, T_M \text{ a testcase} \mid T_M \text{ verifies } c \\ (\forall c \text{ a coverage item}, M_1 \checkmark c \wedge M_2 \checkmark c) \implies \left(M_{1\,|\,\Sigma_{SLM} \times S_{SLM}} \sim M_{2\,|\,\Sigma_{SLM} \times S_{SLM}}\right) \end{gathered} \tag{18}$$

Moreover, any model verifying every coverage item is *a* largest submodel of the SLM, so:

$$\begin{cases} M_1 \preceq SLM \wedge (\nexists\, M \text{ an other model} \mid M_1 \preceq M \preceq SLM) \\ M_2 \preceq SLM \wedge (\nexists\, M \text{ an other model} \mid M_2 \preceq M \preceq SLM) \end{cases} \tag{19}$$

While strict equivalence between M_1 and M_2 cannot be proven that way, results 18 and 19 are the closest to an equivalence we can achieve, since states in $S_1 \backslash S_{SLM}$ and $S_2 \backslash S_{SLM}$ are never compared.

3.6 Verification Metrics

To assert the quality of a testcase, two very important metrics must be defined: the **coverage rate** and **success rate**. Mathematically, the **counting measure** [16] of a set E can de defined as such:

$$\mu(E) \overset{def}{=\!=} \begin{cases} \#E & \text{if } E \text{ is finite} \\ +\infty & \text{otherwise} \end{cases} \tag{20}$$

The coverage rate of a testcase is thus:

$$\mu_c(T_M) \overset{def}{=\!=} \frac{\mu(\{c \text{ evaluated by } T_M\})}{\mu(\{c \text{ in SLM}\})} \in [0, 1] \tag{21}$$

Similarly, the success rate of a testcase is:

$$\mu_s(T_M) \overset{def}{=\!=} \frac{\mu(\{c \text{ verified by } T_M\})}{\mu(\{c \text{ evaluated by } T_M\})} \in [0, 1] \tag{22}$$

While the distribution of testcases covering coverage items is theoretically irrelevant, it has a practical importance in terms of performance. The amount of information required to effectively run a simulation can greatly vary depending on the type and range of the simulation (some testcases only need to simulate a portion of Rhea), the complexity of the VRs to be tested, and the tools used to perform and monitor the testcase. A balance in the number and complexity of VRs per testcase is to be found based on the amount of resources available, as both strategies have their advantages and disadvantages:

fewer more complex testcases	**many smaller testcases**
can each verify more VRs	can verify VRs independently
means fewer simulations so less setup time overall	means smaller simulations so better parallelism
take longer per simulation	require more space to run every simulation
need to cover exponentially more branches overall	can miss coverage branches if simulations are too simple

This compromise can be mathematically expressed by the **redundancy rate** between two testcases:

$$\mu_r(T_{M_1}, T_{M_2}) \overset{def}{=\!=} \frac{\mu(\{c \text{ evaluated by } T_{M_1}\} \cap \{c \text{ evaluated by } T_{M_2}\})}{\mu(\{c \text{ in SLM}\})} \in [0,1] \tag{23}$$

Since the number of coverage items in the SLM is hard to evaluate, and can vary based on the portion of Rhea we want to verify, the redundancy rate alone is not a sufficient enough measure to evaluate the redundancy between two testcases. The **Jaccard index** [17], also known as Intersection over Union (IoU) [18], of two sets A and B can help solve this issue:

$$J(A,B) = \frac{\mu(A \cap B)}{\mu(A \cup B)} \tag{24}$$

By generalising the coverage rate to multiple testcases as such:

$$\mu_c(\{T_{M_i}\}_{i\in I}) \overset{def}{=\!=} \frac{\mu(\bigcup_{i\in I}\{c \text{ verified by } T_{M_i}\})}{\mu(\{c \text{ in SLM}\})} \in [0,1] \tag{25}$$

the Jaccard index of two testcases (provided at least one of them evaluates a coverage item) can be defined as:

$$\begin{aligned} J(T_{M_1}, T_{M_2}) &= \frac{\mu(\{c \text{ evaluated by } T_{M_1}\} \cap \{c \text{ evaluated by } T_{M_2}\})}{\mu(\{c \text{ evaluated by } T_{M_1}\} \cup \{c \text{ evaluated by } T_{M_2}\})} \\ &= \frac{\frac{\mu(\{c \text{ evaluated by } T_{M_1}\} \cap \{c \text{ evaluated by } T_{M_2}\})}{\mu(\{c \text{ in SLM}\})}}{\frac{\mu(\{c \text{ evaluated by } T_{M_1}\} \cup \{c \text{ evaluated by } T_{M_2}\})}{\mu(\{c \text{ in SLM}\})}} \\ &= \frac{\mu_r(T_{M_1}, T_{M_2})}{\mu_c(T_{M_1}, T_{M_2})} \in [0,1] \end{aligned} \tag{26}$$

It is interesting to note that:

$$\begin{aligned} \forall\, T_M,\ \mu_r(T_M, T_M) &= \frac{\mu(\{c \text{ evaluated by } T_M\} \cap \{c \text{ evaluated by } T_M\})}{\mu(\{c \text{ in SLM}\})} \\ &= \frac{\mu(\{c \text{ evaluated by } T_M\}}{\mu(\{c \text{ in SLM}\})} \\ &= \mu_c(T_M) \end{aligned} \tag{27}$$

so:

$$\begin{aligned} J(T_M, T_M) &= \frac{\mu_r(T_M, T_M)}{\mu_c(T_M, T_M)} \\ &= \frac{\mu_c(T_M)}{\mu_c(T_M, T_M)} \\ &= 1 \end{aligned} \tag{28}$$

Ideally, the success rate of each testcase and the global coverage rate would be 1, and the Jaccard index of each pair of testcases would be their Kronecker delta (*i.e.* 1 if they are the same and 0 otherwise), meaning that each coverage item from the VRs were evaluated and verified by exactly one testcase.

Whichever compromise has been used, a **coverage report** can be obtained at the end of a testcase simulation to review the coverage and success rates of the testcase and examine which coverage items failed (verification issue) or were not tested (coverage issue). The redundancy rate of two testcases can also be obtained by comparing the coverage items listed on their respective coverage report. Figure 7 shows an example of such a report. In this case, less than 20% of the coverage items were evaluated, but the ones that were all passed.

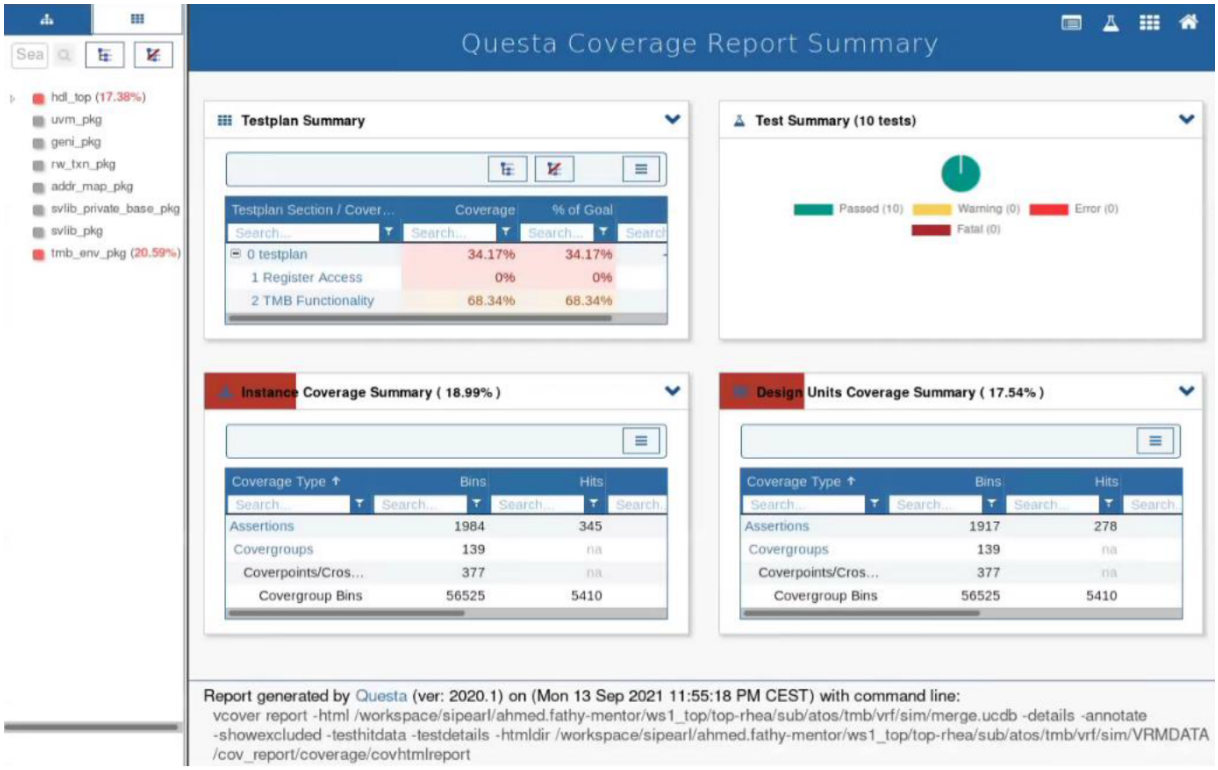


Figure 7: An example coverage report for a testcase

4 Processor Architecture

4.1 Overview

This section will try to provide a simple overview of Rhea's architecture as specified by the MAS. While not diving into details it aims to give the reader a clear understanding of Rhea's different **subsystems** and how they interact together.

The chip design, **Rhea** is Sipearl's first-generation General Purpose Processor (GPP). It is a HPC dedicated multicore chip with high energy efficiency, great computational performance and with high reliability. Figure 8 shows the top-level view of the Rhea chip, consisting of the main Rhea die, four High Bandwidth Memories (HBM), four Double Data Rate (DDR) SDRAM interfaces for external DDR5 memories and eight Peripheral Component Interconnects express (PCIe). Figure 8 gives an overview of the top-level structure of the Rhea chip.

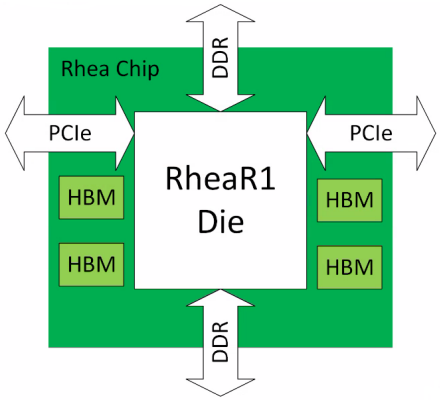


Figure 8: Overview of the Rhea chip

The Rhea chip contains multiple cores and caches connected by networks-on-chip (NoC) which connect the memory and the inputs and outputs of its subsystems together. In total, there are 3 NoCs:

- Coherent Data-NoC (D-NoC): based on the ARM Coherence Mesh Network (CMN) 700 architecture, this coherent 2D mesh network targets high-rate data-transfers while maintaining coherency of all components. It provides the following connectivity:
  - Advanced eXtensible Interface (AXI) interfaces for high-rate data transfers to I/O components and to memory devices
  - Coherent Hub Interface (CHI), AXI Coherency Extensions (ACE) lite and Credited eXtensible Stream (CXS) interfaces for high rate data transfers supporting full coherency functionality
  - AXI streaming interfaces for distribution of interrupts based on the ARM Generic Interrupt Controller (GIC) architecture
- Control-NoC (C-NoC): based on the ARM Non-coherent Interconnect (NI) 700 architecture, this NoC provides non-coherent connectivity for configuration by the System Control (SYSCTRL) based Control Processors. It provides access to control, status register spaces and debug memory spaces on the chip module via Advanced Peripheral Bus (APB) interfaces. AXI interfaces are also supported where needed.

- Trace-BUS: Dedicated NoC for tracing based on ARM STM-500 Intellectual Property (IP) blocks and the Advanced Trace Bus (ATB) protocol

Figure 9 presents the toplevel architecture of the main Rhea die with its tiles and its interconnects.

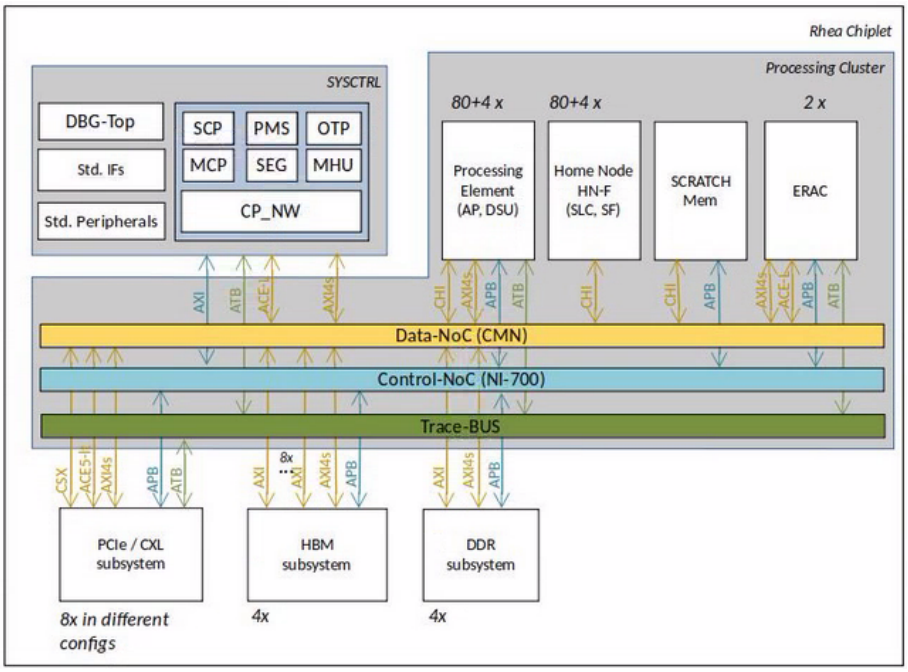


Figure 9: Toplevel architecture of the Rhea die

4.2 Core Cluster

The Processing Cluster, or **Core Cluster** (CC), is the main Application Processor (AP) processing mesh, consisting of the D-NoC, AP-cores, and other processing tiles. It consists of four quadrants, each quadrant consisting of a 2D grid of Mesh Tiles (MT). Figure 10 presents the Core Cluster's layout with focus on the q0 quadrant and a MT.

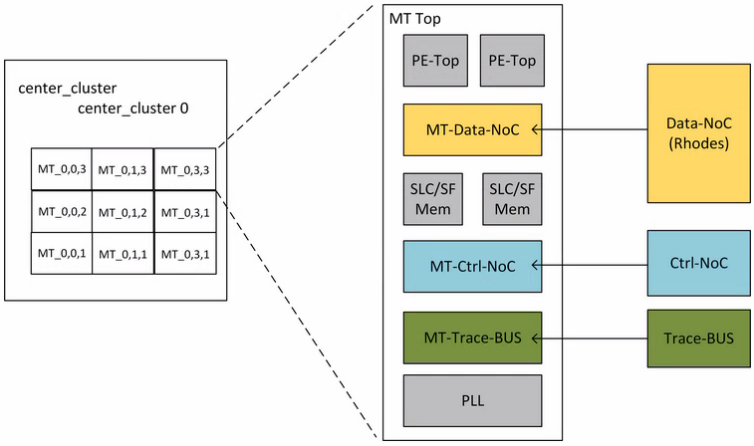


Figure 10: Structure of a Core Cluster's quadrant

A MT is a hierarchical block that can be considered as hard macro for the CC layout, and is identified by its quadrant's number and its x and y position inside the quadrant. Each MT comprises:

- two instances of Processing Elements (PE) in the form of Zeus AP cores and their respective level 3 cache control logic and interface using an ARM DynamIQ Shared Unit (DSU)
- two instances (one per PE) of Atos Tile Management Blocks (TMB) and Power Policy Units (PPU)
- two instances of Single-Level Cell (SLC) memories with their Snoop Filters (SF), using an ARM Fully-coherent Home Node (HN-F) to manage cache according to ARM's standards
- an instance of Phase-Locked Loop (PLL) for clock management inside the MT
- the required local instances of the D-NoC, C-NoC and Trace-BUS

Figure 11 presents a block diagram of a MT and its components.

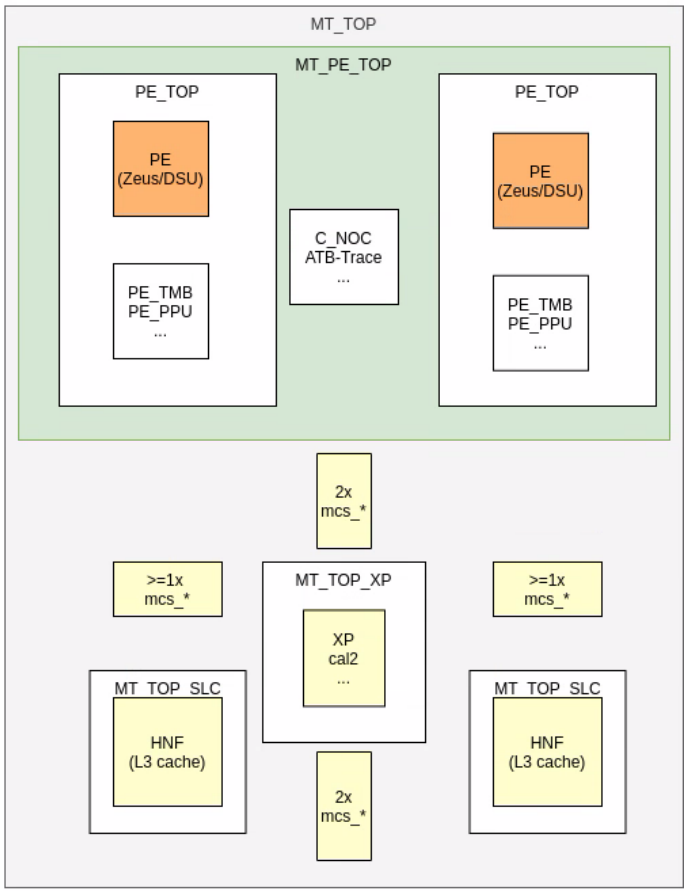


Figure 11: Structure of a Mesh Tile

4.3 System Control

The **SYSCTRL** tile is the block containing the system control logic. It is the starting point of the C-NoC, which enables configuration, reset, clock distribution and debug access to every AP's PEs and chip-wide distributed subsystems such as DDR, HBM, PCIe, ...

It comes with a wrapper shell around the C-NoC which enables interrupt collection from the PEs and the distributed subsystems, and transport the control processors inside SYSCTRL. It has a close relationship with the SCRATCHMEM memory, from which it extracts booting configuration data and transfers the code to be run by the PEs to the corresponding APs. It is complex subsystem that we won't describe in further detail here, as it is not the main purpose of this document, but extensive documentation about how it works and interacts with the rest of the Rhea chip can be found in Polarion. Figure 12 describes SYSCTRL's architecture.

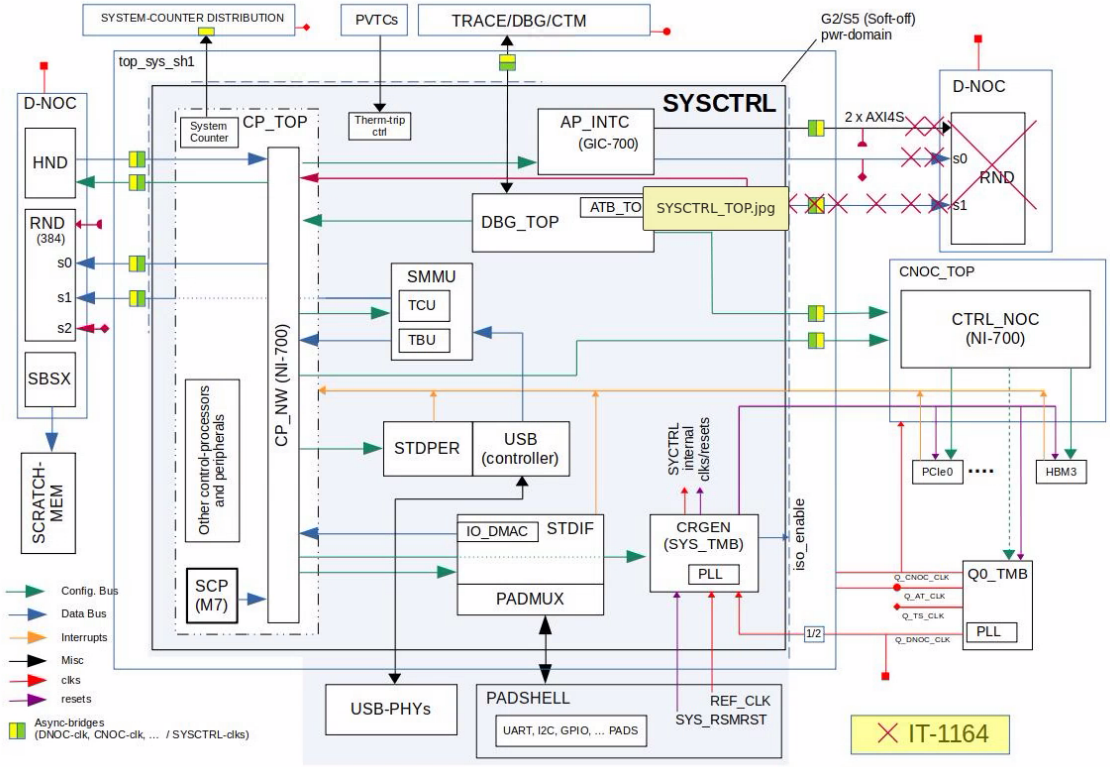


Figure 12: Detailed architecture of the SYSCTRL tile

4.4 DDR

The **DDR** subsytem consists of an interface to the external DDR memories, and supports high-capacity high-bandwidth DDR5 memories. Each DDR tile supports dual channel to connect to DDR5 Registered DIMMs (RDIMM), Load-Reduction DIMMs (LRDIMM) and 3D-Stacked RDIMMs (3DS-RDIMM), with each channel using a 40-bit data lane (32-bit data and 8-bit error correction code). Four physical ranks are supported and can be mapped to up to two DIMM slots (two physical ranks for 3DS-RDIMM). The DDR tile supports DDR5 RDIMMs at up to 5200MT/s frequency, LRDIMM at up to 4800MT/s, and 3DS-RDIMM up to 8-high.

It is connected to the D-NoC through two AXI 256-bit interfaces running at 650MHz for the two DDR5 channels. The AXI interfaces are connected through the datapath to the inputs of the controller. The controller through two interfaces to the Phy (the physical layer between controller and pins) which communicates with the two DDR5 lanes to the DIMMs at up to 5200MT/s.

The DDR subsystem is initiated and configured through a 32-bit APB interface connected to the Control-NoC.

The central block of the control path is the TMB in charge of handling clocks, resets, and more. It contains five APB devices (the TMB, the performance monitor, the DDR controller and the DDR Phy) which are all connected to and controlled by the APB interface. Figure 13 gives a an overview of the DDR interface's architecture.

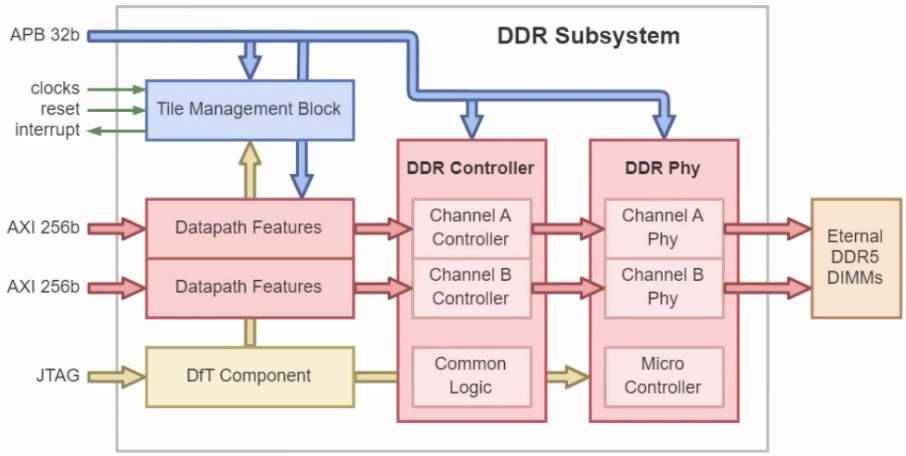


Figure 13: Architecture of a DDR tile

4.5 HBM

The **HBM** subsystem is the high-speed memory counterpart of the DDR subsystem, located on the Rhea chip and outside of the Rhea die. A HBM is a 3D-stacked SDRAM capable of handle large amount of data transfer (reading or writing) at the same time. This can be achieved by using a much wider memory bus than its DDR5 counterpart.

To achieve this high-bandwidth, the HBM subsystem is connected to the D-NoC through 16 AXI 256-bit interfaces corresponding to the 16 pseudo-channels of the HBM stacks. Those AXI interfaces run at 800MHz and are fed to the inputs of the controller. Two pseudo-channels in the data-path correspond to one channel. The controller runs at 800MHz and connects through 8 interfaces to the Phy which communicates with the HBM stacks at 3.2GT/s.

The HBM subsystem is also initiated and configured through a 32-bit APB interface connected to the Control-NoC. The central blocks of the control path are the two TMBs which handle clocks, resets, and more. The other IPs of the tile including the TMB, the HBM controller and the HBM Phy are also connected to and controlled by an APB interface. Figure 14 gives a an overview of the DDR interface's architecture.

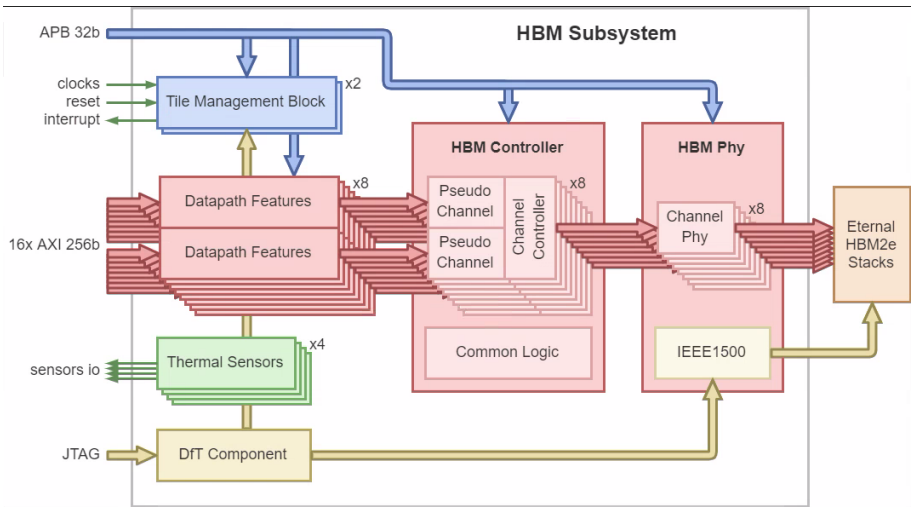


Figure 14: Architecture of a HBM tile

4.6 PCIe

The Rhea chip embeds six **PCIe** x16 lane and two PCIe x4-lane instances, each allowing to establish communication with an external peripheral using the appropriate communication standard . The x16 and x4 subsystems can operate as either as Root Port (RP) or End Point (EP), and the x16 subsystems can also operate as Compute eXpress Link (CXL). Each of the PCIe subsystem can be configured to one of the operational modes at the boot time. The use cases of the operational modes are as follows:

- RP: provides access to external PCIe-based devices.
- EP: allows external PCIe-host access to Rhea resources.
- x16 CXL: allows to connect to other Rhea or future EPI processors and forming a cache coherent compute cluster.

Figure 15 presents the PCIe spatial distribution on the Rhea chip.

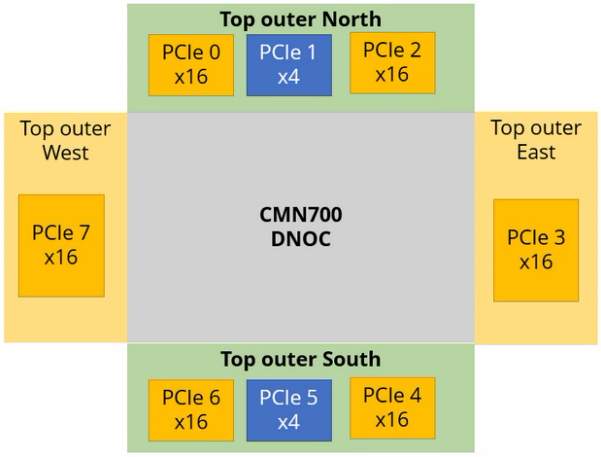


Figure 15: PCIe distribution for Rhea

4.7 ERAC

The EPI Rhea ACcelerator (**ERAC**) is a Rhea-specific EPI accelerator [19] designed by the CEA. The ERAC tile is connected to the GPP D-NoC through a Non-Coherent Home Node (HN-I) port and a DVM Request Node (RN-D) port. The HN-I provides a 128 bits AXI slave port while the RN-D provides two 512 bits ACE5-Lite master ports. It is also connected to the C-NoC through an APB slave port. Finally, ERAC is connected to the top Joint Test Action Group (JTAG) chain for RISC-V cores debug functionality. ERAC tile embeds one STX (the integration of a Stencil Processing Unit (SPU) and a Neural network Tensor Accelerator (NTX)) and four VRP (VaRiable Precision accelerator) cores based on the CVA6/ARIANE RISC-V processor [20], interconnected through an AXI network to the GPP bridge which:

- translates memory addresses to the GPP physical memory space
- generates interrupts to the GPP ARM cores.

At top level, the tile includes a TMB to generate and control clocks, provide configuration registers, control resets and provide a power management interface. It also implements an AXI4 configuration network accessible through the HN-I port. Figure 16 gives a an overview of the ERAC's architecture.

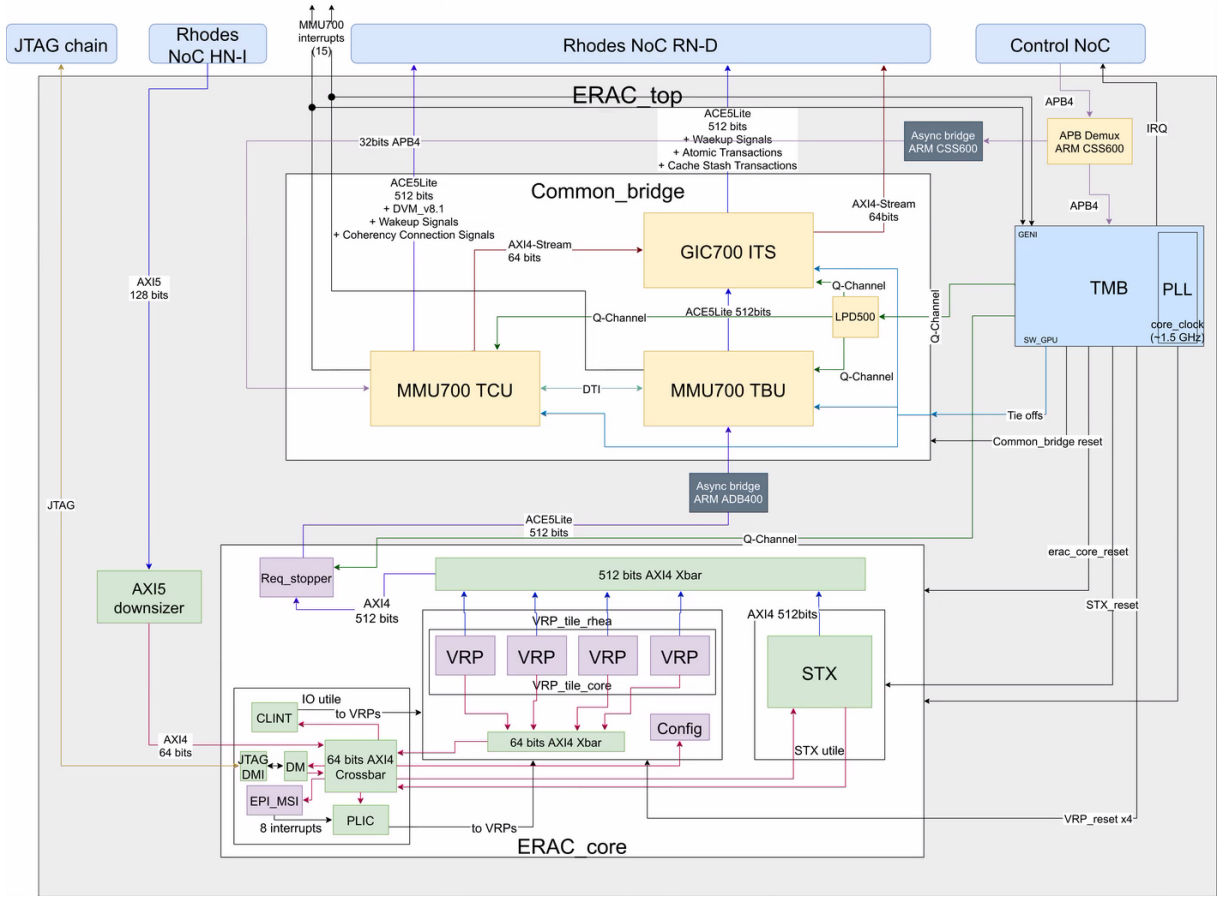


Figure 16: Detailed architecture of the ERAC tile

5 Simulation

5.1 RTL and Virtual Prototype

The specifications and the verification flow have been defined, and Rhea's macro and micro-architectures have been extracted from the requirements, but it still needs to be effectively implemented. To achieve this, two new models have been used : the **Register Transfer Level** (RTL) and the **Virtual Prototype** (VP).

The RTL view of Rhea is the design abstraction level that models the processor in terms of the flow of the digital signals being exchanged between hardware registers by physical wires, and the logical operations performed on those signals. The RTL model is described using the SystemVerilog Hardware Description Language (HDL), an object-oriented extension of the Verilog HDL. This view serves mostly two purposes:

- from the frontend perspective, it is used to perform simulations to verify the chip design or run performance tests.
- from the backend perspective, it is the entry point to synthesise, place and route the chip to obtain the floorplan and send it to the manufacturer for tapeout. This document will not focus on backend aspects, but it should be noted that, ultimately, it is the RTL version that will define the chip's behaviour.

On the other hand, the VP view is a more abstract model of the processor, in which the physical registers and wires are replaced by an abstract black box whose behaviour is defined using a high-level programming language. Some registers, the Control and Status Registers (CSR), are retained in the VP, but they are modelled in a much higher view point than they were in the RTL. Since the CSRs are explicit in all three models of Rhea, they will serve as state outputs for verification purposes. In Rhea's case, the SystemC language, an event-driven set of C++ classes and macros, has been chosen as base language for the VP. As they provide an ecosystem of efficient computing and communication IPs, ARM Fast Models [21] have been extensively used in this model, especially in the CC. This model can be used in a wide variety of situations:

- for debugging purposes, its high-level view makes it easier to identify and correct behavioural issues in simulations, as well as monitor performance.
- for architectural exploration, its high flexibility allows the model to be reshaped at ease to compare different architectural configurations or create new ones.
- for software development, as it is a lighter and more portable model than its RTL equivalent, and can be used to enable early software development while the RTL is still in progress.
- for hardware frontend and backend, it can be used to perform High-Level Synthesis (HLS) to obtain a RTL translation of the SystemC code.

Both models are complementary and have their own advantages and disadvantages:

the RTL view	**the VP view**
is a more precise model of the chip	is faster and smaller to simulate
defines the actual behaviour of the chip	needs to be aligned with the RTL
can be monolithic and challenging to modify	can easily be altered into a new configuration

5.2 Cosimulations

With infinite resources, computational power and time, simulating the full RTL chip would be the simplest solution to run testcases and evaluate their associated VRs. In this scenario, verifying the RTL model would technically be enough to ensure the correct behaviour of the chip, even though its development would be slower than implementing both the RTL and the VP from debugging and exploration points of view.

However, as mentioned in section 3.4, the size of the RTL model for Rhea is much greater than the maximum allowed by the simulators currently used to run the testcases, and even simulating smaller parts of the chip can take an unreasonable amount of time. A solution to address this issue is to perform a **cosimulation**, which consists of running a simulation that combines the two models. A cosimulation is thus able to benefit from the best of each model by allowing to precisely simulate the relevant subsystems while accelerating the simulation speed and reducing the size of the other ones.

This, however, raises two issues:

- the VP models used in the cosimulation need to be equivalent to their respective RTL models to prevent differences in behaviour from skewing the assessment of the VRs.
- since the RTL and the VP are written using different programming languages, they cannot be interfaced together as such. An interlanguage communication protocol is required to establish the communication.

The first issue will be discussed in this section, while the second one is the subject of the next section.

For Rhea, the verification paradigm for cosimulation is the following:

- for each subsystem, a RTL model and a VP model are implemented by the hardware team and software team respectively. This ensures a complete independence between the two models of each subsystem, and with the design of the testplans (as the verification team is another separate team).
- if an IP comes from a private company or an otherwise external organisation, it is considered verified by the IP provider. Otherwise, no unit testcase in performed. Since Sipearl mainly focuses on integration, it is case of the vast majority of IPs used in both models.
- no individual integration testcase is performed on each subsystem model individually. While this is not an issue in itself, it must be taken into account when designing global integration testplans to avoid verification loops. A verification loop can arise if two testcases each need to use a signal from a subsystem model the other is verifying, as a signal from an unverified model cannot be used to verify another model. Currently, no verification mechanism has been implemented by Sipearl to prevent such loops. A possible solution would be to create a graph-based representation of the dependencies between testcases and transform it into a cycle-less tree representation.
- global integration testcases on **cosimulation models** are run to assert the correlation between each model of each subsystem and the SLM as described in section 3.5.

In total, four distinct cosimulation models are being used to verify the VRs, one per main subsystem: CC and SYSCTRL, DDR, HBM, and PCIe. In each cosimulation model, the main subsystem is implemented as its RTL model while the rest of the chip is in its VP model. Since the RTL will ultimately define the chip's behaviour,

Sipearl has chosen to put the emphasis on verifying the RTL model and mostly assuming the correctness of the VP, which explains the imbalance between the two models in each cosimulation model. A solution to counterweight this would be to create a unique cosimulation model based on a SystemVerilog bone structure in which each subsystem model could be inserted and replaced by its other model independently of the rest of the chip.

5.3 Interlanguage Communication

The **Universal Verification Methodology Connect** (UVMC) [22] is an open-source UVM-based library that provides **Transaction-Level Modelling** (TLM) [23] connectivity and object passing between SystemC and SystemVerilog UVM components and modules. It can be thought of as a high-level wrapper around the SystemVerilog **Direct Programming Interface** (DPI) with SystemC communication protocol. Figure 17 shows how communication between a SystemVerilog producer module and a SystemC consumer component can be established using UVMC.

SystemC

```
#include "uvmc.h"
#include "consumer.h"
S
int sc_main(int argc, char *argv[]) {
  consumer cons("cons");

  uvmc.uvmc_connect(cons.in, "foo");
  sc_core.sc_start();

  return 0;
}
```

SystemVerilog

```
import uvmc_pkg::*;
`include "producer.sv"

module sv_main;
  producer prod = new("prod");
  initial begin
    uvmc_tlm #()::connect(prod.out, "foo");
    run_test();
  end

endmodule
```

Figure 17: SystemVerilog/SystemC communication using UVMC

The previous example illustrates just how simply communication can be established across the language barrier by using a global lookup string ("foo" in this case). It is worth mentioning that the same protocol can be used to connect two SystemVerilog modules or two SystemC components together, though the DPI protocol already allowed such transactions. Despite its apparent simplicity, UVMC is an extremely powerful tool because, apart from the lookup string, it doesn't require any knowledge from the producer nor the consumer of whom it is connected to or in what language it is written as it is the case for standard DPI communication.

Communication using only native DPI would also be technically possible, but much more complicated, and introduce many inter-dependencies between subsystems. Figure 18 [24] gives an example of pure DPI communication between a Verilog module and C functions. As in the previous example, the presented code is deliberately simple to focus on the communication aspects, but it should be noted that the C functions can, and usually are, wrapped as methods inside SystemC classes, and the Verilog modules can be integrated into SystemVerilog threads if needed.

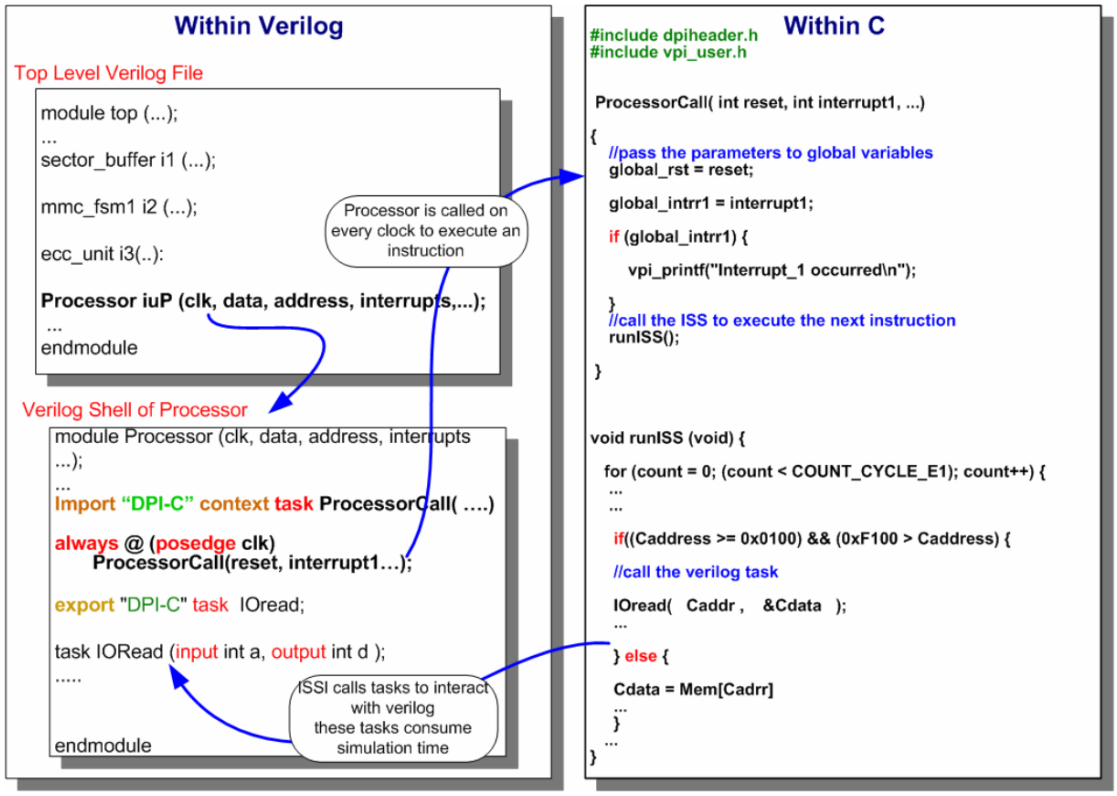


Figure 18: Verilog/C communication using native DPI

The reason why UVMC makes communication so transparent for the user is because it creates a target socket proxy and an initiator socket proxy between the producer and consumer, and uses DPI to connect them together. Figure 19 [25] shows a diagram explaining how UVMC uses this concept to allow interlanguage communication between a SystemVerilog (in blue) producer module and a SystemC (in orange) consumer component.

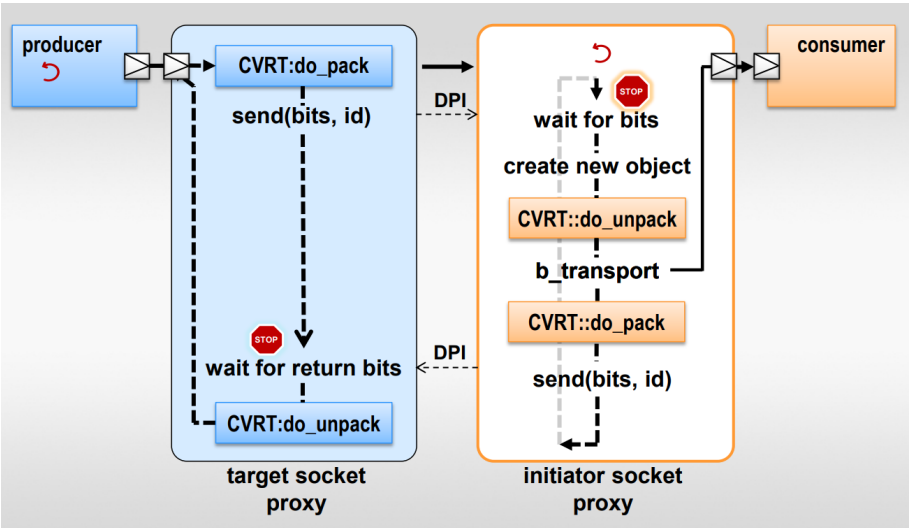


Figure 19: UVMC communication using TLM-based DPI

Since most communications between subsystems happening in Rhea are handshakes, UVMC communications are blocking transports. This is why figure 19 describes a transport in which the producer waits for the target socket proxy to receive

5.4 The DUT paradigm

Thanks to UVMC, Rhea can be simulated on Sipearl's simulators, but it still lacks an environment required to run a testcase. The **Device Under Test** (DUT) paradigm is the combination of a design to be tested, one or more agents sending stimuli to the design (active agent) or monitoring its response to those stimuli (passive agent), and a scoreboard to keep track. A classic DUT testing sequence always follows the same pattern [26]:

- the test sequencer inside an active agent communicates a series of stimuli to be sent to the design through its interface to the agent's driver.
- the design receives those stimuli, processes them, and sends its response to its interface.
- a passive agent monitors the response and sends it to the scoreboard for processing.
- in parallel, the active agent responsible for the stimuli sends to the scoreboard the system's expected behaviour as a reference model. The active agent also monitors the response and adapts its model based on the design's previous responses, so that any difference between the expected and the actual behaviours are only caused by the current stimuli and that previous behavioural differences do not cascade on the current test.
- the scoreboard receives the design's response from the passive agent and the reference model from the active agent, and compares them. If the design behaved as predicted, the test is considered as passed. Otherwise, it has failed.
- when every test has been evaluated, the scoreboard compiles the results and produces a report describing which coverage item has been evaluated and verified. This report is used to create new tests if full coverage hasn't been achieved, or correct the design if certain tests failed. It is also used to extract the verification metrics described in section 3.6.

[26] provides more information and excellent SystemVerilog tutorials on each DUT component, and gives examples on how to write and execute testcases.

For Rhea, the designs to be tested are composed of both SystemVerilog and SystemC codes connected by UVMC. The scoreboard, active and passive agents, and the interface are written in SystemVerilog.

Figure 20 presents a diagram explaining the architecture of a DUT and the interactions between its components.

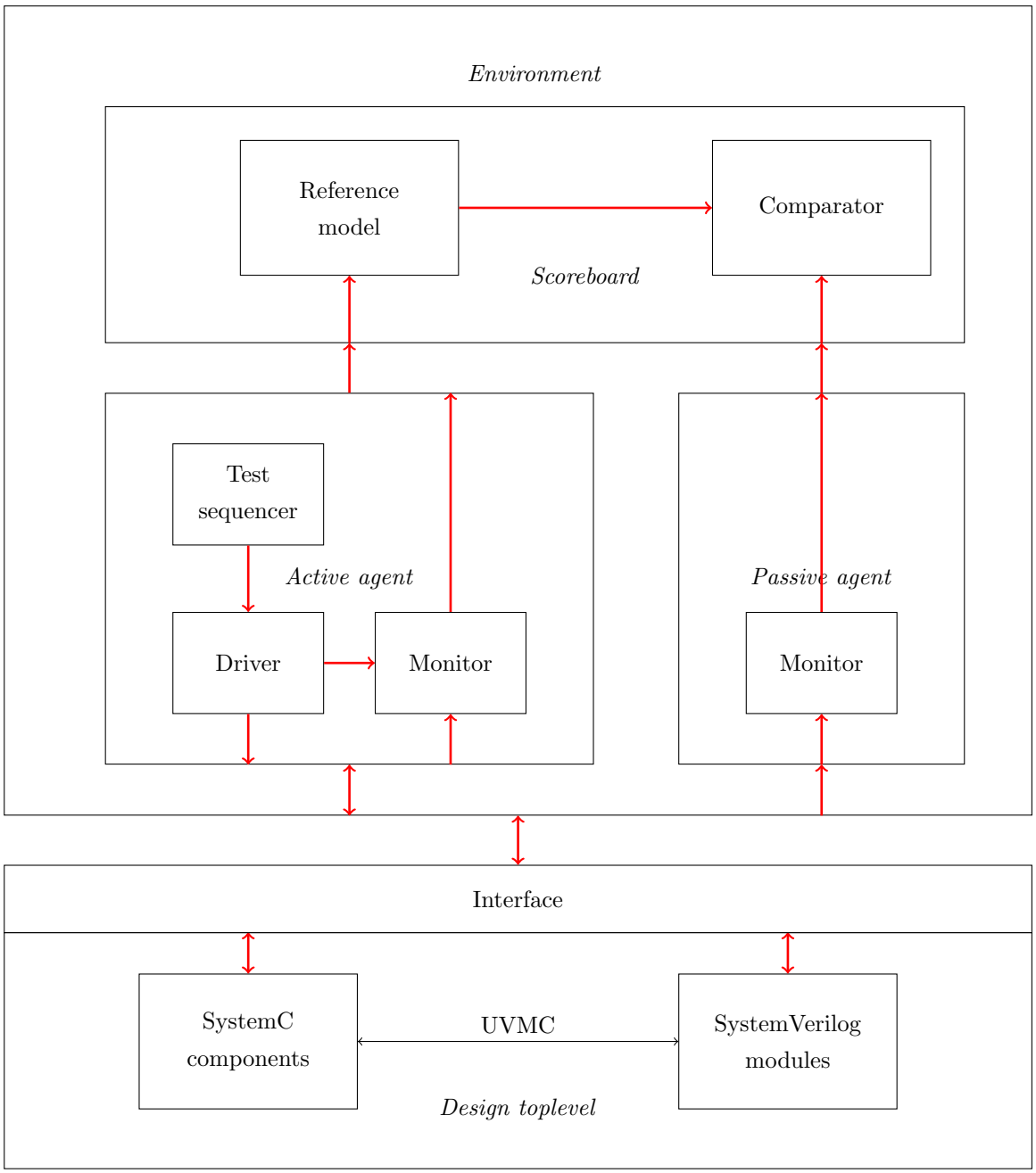


Figure 20: The complete DUT diagram

6 Experimental Setup

6.1 Gitlab and Continuous Integration

As Rhea is currently under development, its state changes regularly and must be constantly tracked to avoid conflicts between versions, reduce bugs, and increase interoperability between teams across the different sites. To achieve these goals while maintaining high security and confidentiality levels, Sipearl uses the Git distributed version control tool on their own private **Gitlab** servers.

Since Rhea had to be reaching its next milestone by the end of this internship, it had been decided that the version on which the verification methodology presented in this document would be evaluated had to be as stable as possible. To do this, tests were conducted using a unique specific version of Rhea, identified by its SHA-1 hash checksum.

Further using Gitlab features, Sipearl has decided to use the **Continuous Integration** (CI) tooling to lessen programming conflicts between teams and increase the efficiency of its verification flow:

- each user can freely create branches from existing branches, and edit files on their own local branch independently from the rest of the project.
- when they are satisfied with their changes, the user can push the updated files on their personal branch on the server. Doing so will trigger the verification flow, which consists of a pipeline of multiple testcases run simultaneously according to a special file named *.gitlab-ci.yml*. Their execution and sequencing is managed by the **Slurm** [27] workload manager. Slurm is used on top of the native Gitlab runner as it is able to intelligently take the size of testcases to be run into account when allocating computation resources.
- if the all the testcases pass, the CI will trigger a manual approval process to make sure the relevant authorities are aligned with the changes made. Otherwise, fixes must be made locally and pushed on the server to run the testcases once more.
- once a branch has passed both the automated verification process and the manual review, it can be merged onto the branch from which it originates.
- if the father branch has been updated due to another merge event happening between the creation of the branch and its merge, a merge conflict has to be manually resolved. Otherwise, changes are added to the initial branch on the server.

While this process can be done recursively, creating a branch from a branch that is not the master branch, Sipearl strongly advises against doing so, as it only delays merges on the master branch and paves the way for bigger merge conflicts. Instead, branches should be as small as possible both in size and time to reduce the risk of drifting too much from the master branch and avoid merge conflicts. Figure 21 [28] gives an overview of Gitlab's CI flow and figure 22 shows the current testcase pipeline used for Rhea. As mentioned in section 5.2, each *trigger:*:verif* corresponds to a series of testcases run on the corresponding cosimulation model.

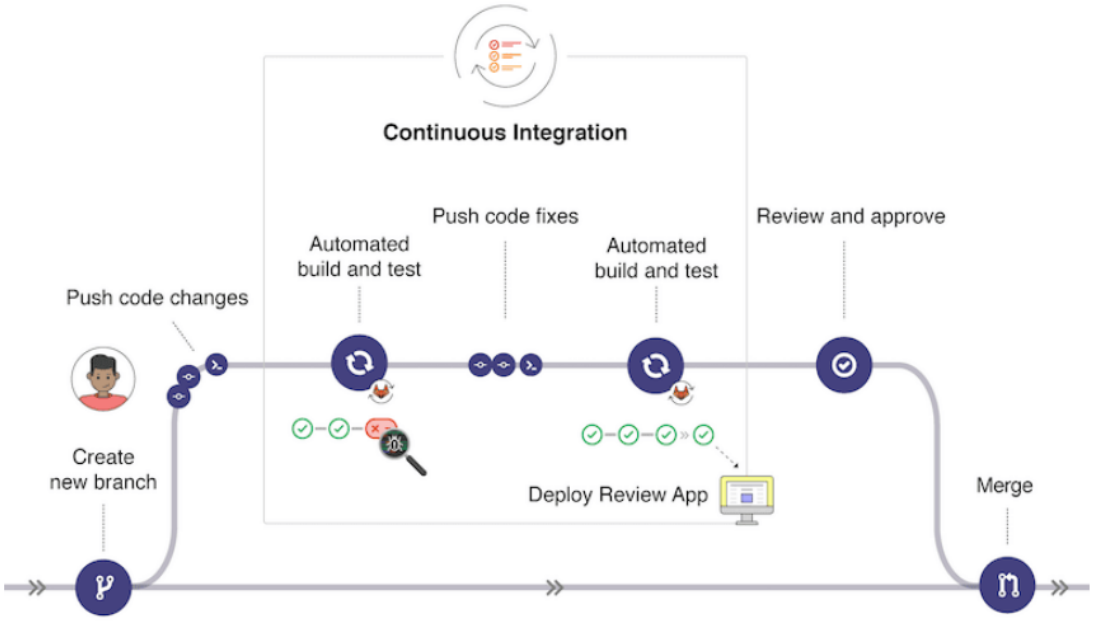


Figure 21: The Gitlab Continuous Integration flow

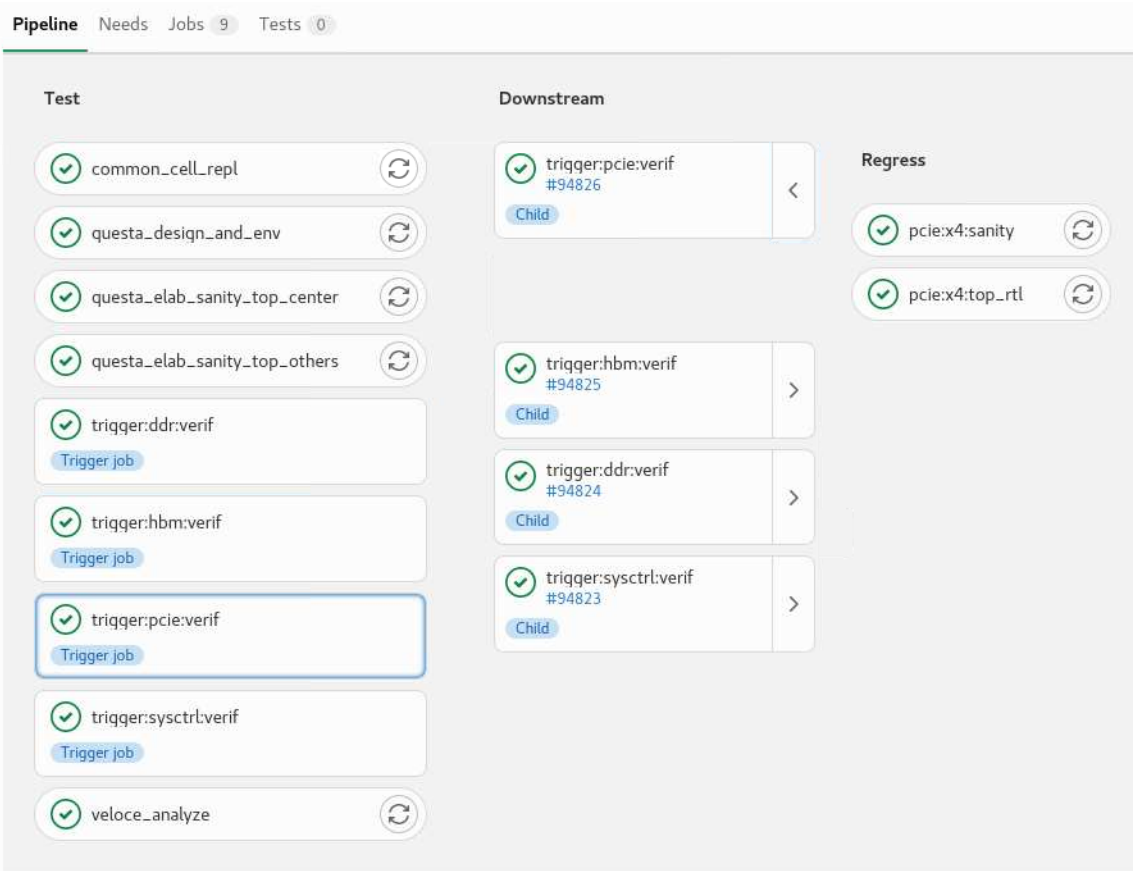


Figure 22: Rhea's current pipeline for the CI

6.2 Hierarchy Dependency Analyser

Before being able to effectively run the testcases with Slurm, they must be compiled in a certain order to ensure the DUT working properly. The correct compilation and simulation sequence for a DUT is the following:

- cleaning any residual file from previous compilation attempts to avoid conflicts with previous versions
- compiling and linking (*i.e.* compiling) the SystemC testcases
- compiling the SystemVerilog/SystemC DUT testbench. A complete testbench consists of the design to be tested, the agents and the scoreboard.
- elaborating the testbench. Elaboration is the hardware equivalent of linkage and consists of binding modules to module instances, building the model hierarchy, and establishing net connectivity between instances.
- running the simulation and executing the testcases
- displaying human-readable results

To perform each of these steps, Sipearl has developed the **Hierarchy Dependency Analyser** (HDA), a tool written in Pearl and wrapped in Makefile scripts. Figure 23 shows the terminal commands to be executed to run the *gic_spi_col* testcase for the CC cosimulation model.

DUT step	**HDA command**
clean	make clean
build tests	make tests
compile testbench	make compile CMN_DIR=gen_ori
elaborate testbench	make elab CMN_DIR=gen_ori ELAB_BEH_MEM=1 SC=1 CDL=0 TTNDPI=0 SVA=0 DEBUGDB=1 TOP=tb_rhea_top_sysctrl_dnoc_q0_cfg BATCH=0
run simulation	make run_opt ELAB_BEH_MEM=1 SC=1 CDL=0 TTNDPI=0 SVA=0 DEBUGDB=1 TESTNAME=gic_spi_col TESTNAME_AP=hello_world PREBUILT_ELF=0 TIMEOUT_CYCLES=8000000 TOP=tb_rhea_top_sysctrl_dnoc_q0_cfg BATCH=0 &

Figure 23: An example HDA sequence for a testcase

HDA uses the Siemens **Questasim** [29] simulator to run the testcases. It is a multi-language, high performance simulation tool widely used in the industry for verification purposes, and provides an extensive set of tools for simulation-based verification and debugging. It is able to explore the structure of a hardware design, execute automated **Tcl** scripts, run simulations and produce human-readable simulation reports in the form of **waveforms**. It can also produce the coverage report described by figure 7 in section 3.6. HDA uses Tcl scripts to compute the status (not evaluated, passed, or failed) of each coverage item in anticipation for the coverage report. Figures 24, 25 and 26 respectively show QuestaSim's structure explorer, Tcl terminal and waveform tabs for the *gic_spi_col* testcase.

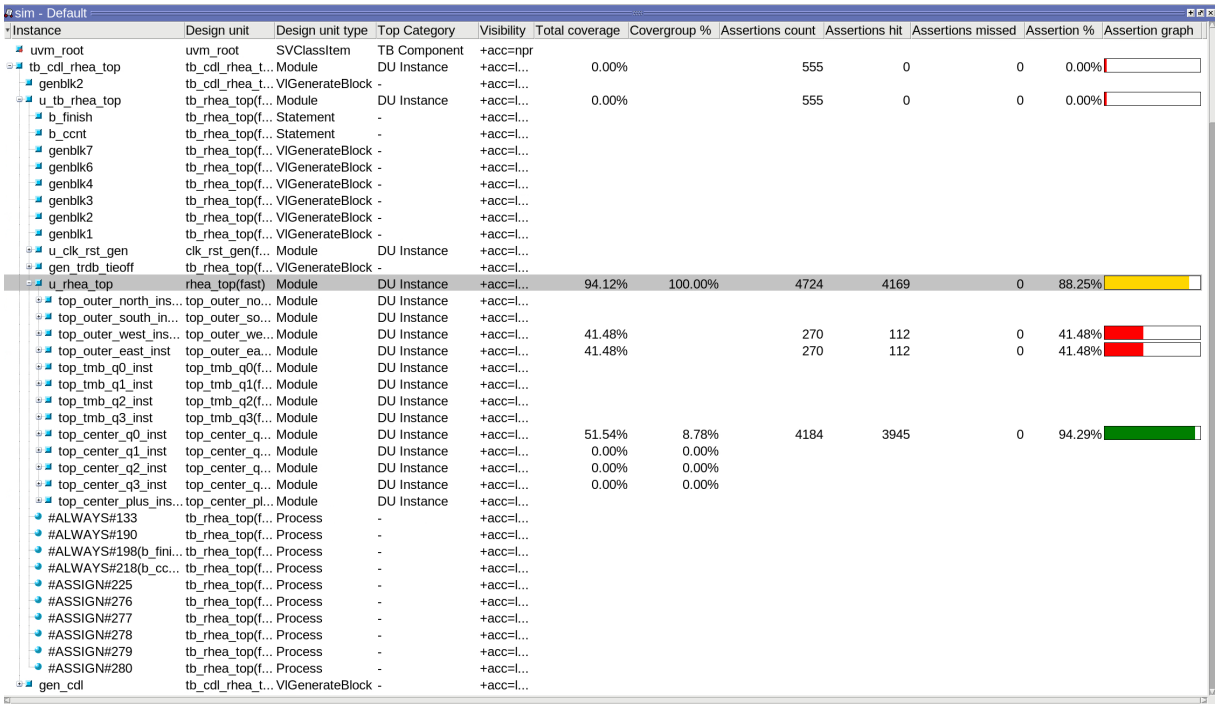


Figure 24: QuestaSim's architecture explorer tab

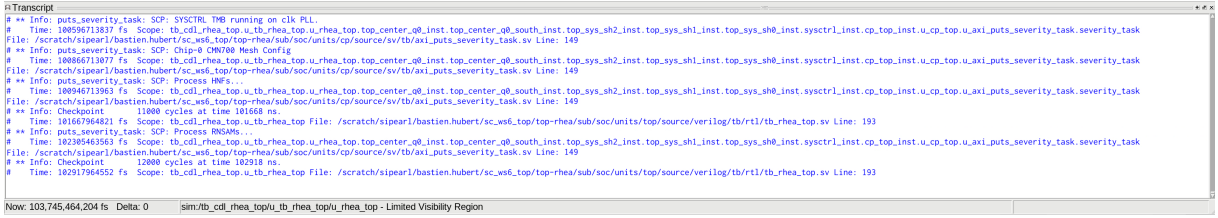


Figure 25: QuestaSim's Tcl terminal tab

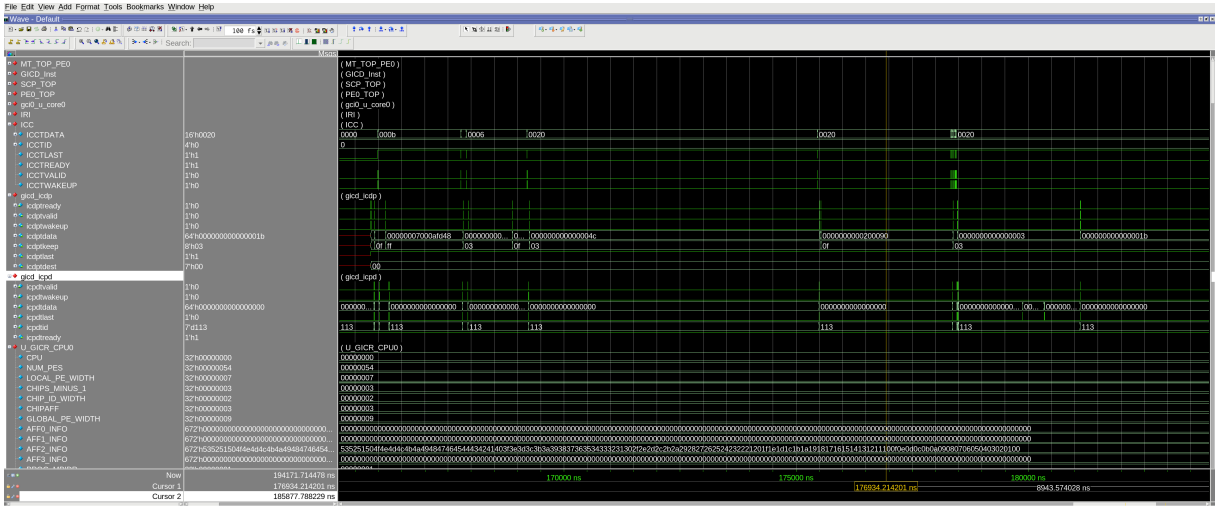


Figure 26: QuestaSim's waveform tab

6.3 Results and Analysis

To evaluate the verification flow presented in this document, seven testplans have been used:

- CC and SYSCTRL
  - T_{CC_1}) dnoc_reg_access_testcase
  - T_{CC_2}) dnoc_ap_boot_testcase
  - T_{CC_3}) sysctrl_cnoc_tiles_testcase
- DDR
  - T_{DDR_1}) ddr_init_test_cosim
- HBM
  - T_{HBM_1}) hbm_cosim_init
- PCIe
  - T_{PCIe_1}) COSIM__pcie_x4__testcase__traffic_cfg_1
  - T_{PCIe_2}) COSIM__pcie_x4__testcase__traffic_cfg_2

By modifying the *.gitlab-ci.yml* file, we were able to change the CI to only take those testplans into account and run the corresponding testcases on the cosimulations models. The success rate of each testcase, the Jaccard index of each pair of testcases, and the global coverage rate of the six testcases were computed from their coverage reports using a Python 3.6.0 script. To reduce calculations, we used to usual relation:

$$\forall\, A, B \text{ two sets, } \mu(A \cup B) = \mu(A) + \mu(B) - \mu(A \cap B) \tag{29}$$

to derive the following relation between coverage rate and redundancy rate:

$$\mu_c(T_{M_1}, T_{M_2}) = \mu_c(T_{M_1}) + \mu_c(T_{M_2}) - \mu_r(T_{M_1}, T_{M_2}) \tag{30}$$

Using this relation, the Jaccard index of two testcases can be written as:

$$J(T_{M_1}, T_{M_2}) = \frac{\mu_r(T_{M_1}, T_{M_2})}{\mu_c(T_{M_1}) + \mu_c(T_{M_2}) - \mu_r(T_{M_1}, T_{M_2})} \tag{31}$$

Figure 27 shows the number of coverage items evaluated by each pair of the seven mentioned testcases. Since $A \cap A = A$, we can use a condensed form where diagonal elements represent the number of coverage items evaluated by a single testcase.

	T_{CC_1}	T_{CC_2}	T_{CC_3}	T_{DDR_1}	T_{HBM_1}	T_{PCIe_1}	T_{PCIe_1}
T_{CC_1}	**4579**	2837	1619	841	925	454	557
T_{CC_2}	2837	**7312**	5452	1037	1129	236	325
T_{CC_3}	1619	5452	**6821**	769	631	378	417
T_{DDR_1}	841	1037	769	**3427**	947	158	206
T_{HBM_1}	925	1129	631	947	**3546**	141	189
T_{PCIe_1}	454	236	378	158	141	**1658**	1246
T_{PCIe_2}	557	325	417	206	189	1246	**2246**

Figure 27: Number of coverage items evaluated by pairs of testcases

The coverage rate of the seven testcases is **36.63%** of the **37920** coverage items contained in the SLM. The sum of the coverage items evaluated by each testcase individually (numbers in bold) is 29589, which corresponds to 78.03% of the total coverage items. The comparison between the coverage rate without taking redundancy into account and its actual value proves that the testcases that were used are highly redundant. It can be explained by the long synchronisation sequences used at the beginning of each cosimulation to place the model in a nominal operating state. Figure 28 shows the associated redundancy rates of each pair of testcases in the same condensed form as before (because $\mu_r(T_M, T_M) = \mu_c(T_M)$).

	T_{CC_1}	T_{CC_2}	T_{CC_3}	T_{DDR_1}	T_{HBM_1}	T_{PCIe_1}	T_{PCIe_1}
T_{CC_1}	**12.08 %**	7.48 %	4.27 %	2.22 %	2.44 %	1.20 %	1.47 %
T_{CC_2}	7.48 %	**19.28 %**	14.38 %	2.73 %	2.98 %	0.62 %	0.86 %
T_{CC_3}	4.27 %	14.38 %	**17.99 %**	2.03 %	1.66 %	1.00 %	1.10 %
T_{DDR_1}	2.22 %	2.73 %	2.03 %	**9.04 %**	2.50 %	0.42 %	0.54 %
T_{HBM_1}	2.44 %	2.98 %	1.66 %	2.50 %	**9.35 %**	0.37 %	0.50 %
T_{PCIe_1}	1.20 %	0.62 %	1.00 %	0.42 %	0.37 %	**4.37 %**	3.29 %
T_{PCIe_2}	1.47 %	0.86 %	1.10 %	0.54 %	0.50 %	3.29 %	**5.92 %**

Figure 28: Redundancy rates of the simulated testcases

From this table, it is possible to compute the **Jaccard matrix**:

$$\mathbb{J} \stackrel{def}{=} \begin{bmatrix} \\ (J(T_i, T_j))_{i,j} \\ \\ \end{bmatrix} \tag{32}$$

Figure 29 presents the Jaccard matrix of the seven testcases:

	T_{CC_1}	T_{CC_2}	T_{CC_3}	T_{DDR_1}	T_{HBM_1}	T_{PCIe_1}	T_{PCIe_1}
T_{CC_1}	**100 %**	31.32 %	16.55 %	11.75 %	12.85 %	7.87 %	8.89 %
T_{CC_2}	31.32 %	**100 %**	62.82 %	10.67 %	11.62 %	2.69 %	3.53 %
T_{CC_3}	16.55 %	62.82 %	**100 %**	8.12 %	6.46 %	4.68 %	4.82 %
T_{DDR_1}	11.75 %	10.67 %	8.12 %	**100 %**	15.73 %	3.23 %	3.74 %
T_{HBM_1}	12.85 %	11.62 %	6.46 %	15.73 %	**100 %**	2.77 %	3.39 %
T_{PCIe_1}	7.87 %	2.69 %	4.68 %	3.23 %	2.77 %	**100 %**	47.00 %
T_{PCIe_2}	8.89 %	3.53 %	4.82 %	3.74 %	3.39 %	47.00 %	**100 %**

Figure 29: The Jaccard matrix of the simulated testcases

As mentioned before, the Jaccard index is a much better metric to evaluated the similarities between two testcases than the redundancy rate, because it is independent of the total number of coverage items and only takes relevant coverage items into account. Figure 29 shows in a much clearer fashion why the global coverage rate was much lower than the sum of individual coverage rates: testcases that use the same cosimulation model are much more alike than two testcases from different cosimulation models.

For instance COSIM__pcie_x4__testcase__traffic_cfg_1 (T_{PCIe_1}) and COSIM__pcie_x4__testcase__traffic_cfg_2 (T_{PCIe_2}) have a Jaccard index of 47% because they both attempt to prove coverage items for the PCIe and use the same signals from other subsystems to do so. Manual analysis of the waveform would conclude the same, but the Jaccard matrix provides a clear overview of the likeness between testcases and is much simpler to use. Similarly, sysctrl_cnoc_tiles_testcase (T_{CC_3}) has a high Jaccard index with dnoc_ap_boot_testcase (T_{CC_2}) because they both use buses in the CC intensively, but it has a lower index with dnoc_reg_access_testcase (T_{CC_3}) because they mostly aim to prove coverage items located on the C-NoC and the D-NoC respectively. T_{CC_2} also mostly focuses on the D-NoC, but the APs generate an important traffic on the C-NoC when communicating with other APs.

In cases where the Jaccard index of two testcases is considered too high, it might be interesting to merge their code to create a single new testcase encompassing the two previous ones. Doing so will greatly reduce the overall verification time, as it will allow the workload manager to allocate resources on other testcases instead of having to wait for the two highly redundant testcases to finish their jobs.

The success rate of each of the seven testcases were also computed, but they all passed every coverage items they evaluated. This is because the simulations were performed at an advanced state of the chip, so most of the chip RTL was already aligned with the SLM. Figure 30 gives the success rates of the evaluated testcases.

T_{CC_1}	T_{CC_2}	T_{CC_3}	T_{DDR_1}	T_{HBM_1}	T_{PCIe_1}	T_{PCIe_1}
100 %	100 %	100 %	100 %	100 %	100 %	100 %

Figure 30: Success rates of the simulated testcases

However, the success rate is not a measure of the quality of the verification flow, but rather of the evaluated model. Since the goal of this paper is to describe improvements on the verification flow itself, success rate is irrelevant in itself, and more effort should be dedicated to reducing the Jaccard index of every pair of testcases to

make sure the Jaccard matrix is as close to the identity matrix as possible.

6.4 Future work

As mentioned in section 5.2, a lot can be done to further improve the verification process described in this document. For example, having a single cosimulation model with encased slots for subsystems could greatly improve the overall reliability, as it would prevent code duplication and optimise the debugging process. Being able to swap the model used for each subsystem would also mean being able to more efficiently verify the VP model, as most testcases currently used for Rhea are dedicated to verifying VRs for the RTL model.

The Jaccard index could be modified to only take "effective" coverage items into account and ignoring the ones evaluated during the booting sequence, as it gives a false impression of redundancy in cases where a unique synchronisation sequence is able to make the model reach its nominal state. Finally, while the success rate has not had an impact in this study, it would be interesting to use a modified index that only counts verified coverage items, instead of all the ones that have been evaluated.

7 Conclusion

To conclude, this report has taken significant strides in advancing the verification flow used for Rhea. A simple yet robust mathematical foundation that underpins a simulation-based verification process between an existing reference model and functional processor models has been introduced, alongside three important metrics. The importance of each of these metrics has been presented, and the theoretical framework required for their computation has been laid down.

Furthermore, a comprehensive overview of the current processor architecture has been described, shedding a light on the intricate relations between its subsystems, and allowing too deepen our understanding of Rhea's complexity and understand its remarkable power efficiency and high-performance capabilities.

The presentation and execution of an experimental setup, guided by the principles set forth in this document, provided us with results that prove the validity and practical utility of our approach in assessing cosimulation equivalence with the reference model. The results were analysed and proved to be both easy to read and informative of the changes to be done in order to improve the verification testcase distribution.

Finally, suggestions have also been made to refine the existing verification process and help smoothly integrate the solutions presented in this document, in hope they are taken into account for future iterations of Sipearl's verification flow.

In summary, this document marks a significant milestone for Rhea's verification flow, offering a solid mathematical foundation, practical metrics, and an improved understanding of Rhea's architecture. As we continue to explore and refine verification methodologies, we anticipate continuous advancements that will contribute to the ongoing evolution of processor design and verification process.

References

[1] https://www.european-processor-initiative.eu

[2] https://sipearl.com/en

[3] *Eléments de théorie des automates*, J. Sakarovitch, 2003

[4] Switching and Finite Automata Theory, Z. Kohavi, 1978

[5] https://polarion.plm.automation.siemens.com

[6] "Formal verification in hardware design: a survey", C. Kern, M. R. Greenstreet, 1999

[7] "A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip", T. Grimm, D. Lettnin, M. Hübner, 2018

[8] "Writing testbenches: functional verification of HDL models", J. Bergeron, 2012

[9] "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques", M. E. Khan, F. Khan, 2012

[10] "Black box and White box Testing Techniques - A Literature Review", S. Nidhra, J. Dondeti, 2012

[11] "What's between simulation and formal verification?", D. L. Dill, 1998

[12] "Simulation-based equivalence checking between SystemC models at different levels of abstraction", D. Große, M. Groß, U. Kühne, R. Drechsler, 2011

[13] "Principles of Sequential-Equivalence Verification", M. N. Mneimneh, K. A. Sakallah, 2005

[14] "Sequential equivalence checking between system level and RTL descriptions", S. Vasudevan, V. Viswanath, J. A. Abraham, J. J. Tu, 2006

[15] https://en.wikipedia.org/wiki/Formal_equivalence_checking#Generalizations

[16] *Introduction aux probabilités et statistiques*, L. Decreusefond, 2020

[17] "*Distribution de la Flore Alpine dans le Bassin des Dranses et dans quelques régions voisines*", P. Jaccard, 1901

[18] https://learnopencv.com/intersection-over-union-iou-in-object-detection-and-segmentation

[19] https://www.european-processor-initiative.eu/accelerator

[20] https://cva6.readthedocs.io/en/latest

[21] https://developer.arm.com/Tools%20and%20Software/Fast%20Models

[22] https://verificationacademy.com/topics/verification-methodology/uvm-connect

[23] https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf

[24] "Hardware/Software Co-Verification Using the SystemVerilog DPI", A. Freitas, 2007

[25] https://verificationacademy.com/va_aws/get?uri=https%3A//s3.amazonaws.com/courses.verification.academy/module_uvm_connect_session2_connections_aerickson.pdf

[26] https://vlsiverify.com/uvm

[27] https://slurm.schedmd.com/documentation.html

[28] https://docs.gitlab.com/ee/ci/

[29] https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator