2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

FPGA Firmware Verification: a common approach for simulation and hardware tests

Stefano Pavinato, ESS, Lund, Sweden (stefano.pavinato@ess.eu)
Emmanuel D'Costa, ESS, Lund, Sweden(emmanuel.dcosta@ess.eu)
Stephane Gabourin, ESS, Lund, Sweden (stephane.gabourin@ess.eu)

This paper presents a possible approach to a common framework for FPGA firmware verification, that leverages Cocotb for simulation and Pytest for hardware validation. Together with the Fast Beam Interlock System at the European Spallation Source, this approach improves test automation, code reusability, and even test coverage. Having standardized test structures also improves continuous integration, maintainability, test reports, and auditing. Preliminary results show improved efficiency, especially in Factory Acceptance Testing (FAT).

Keywords— *Unified Verification Framework; Cocotb; Pytest; Hardware Validation; Fast Beam Interlock System*

I. INTRODUCTION

The European Spallation Source(ESS) is one of the largest science and technology infrastructure projects being built today. The facility design and construction include the most powerful linear proton accelerator ever built [1]. The Fast Beam Interlock System (FBIS) plays a critical role at ESS, it must protect the whole machine against beam induced damages and activation. In parallel, the FBIS must ensure that the machine operates an availability higher than 95%. Its key role is to gather signals from sensor systems, process them, and act on actuators if the machine and the proton beam must be stopped. Sensor systems include PLC based systems controlling i.e. vacuum and magnets systems, and FPGA based systems managing i.e. Radio Frequency systems and Beam Instrumentation. The FBIS operates at high data speed and requires low-latency decision making capability to avoid introducing delays and to ensure the protection of the accelerator. This is achieved through two main hardware blocks equipped with FPGA based boards: a mTCA 'Decision Logic Node' (DLN), executing the protection logic function and a cPCI form-factor 'Signal Conversion Unit' (SCU), which mainly implements the interface between the sensor systems and DLNs [2].

This paper presents an approach to facilitate the sharing of code, components, methodologies, and workflows between simulation and hardware testing for DLNs and SCUs firmware.

II. ENGINEERING PROBLEM DEFINITION

In the V-Model (Verification and Validation Model) there are usually four stages in the right side of the V. One of the ways to call them are: unit testing, integration testing, system testing and Factory Acceptance Testing. Referring to FPGA firmware verification and validation, the first three stages are performed in a simulation environment, while factory acceptance testing is conducted with real hardware using proprietary tools or customs scripts. Although a component or a group of components is tested through each stage, it can be challenging to reuse and adapt what already exists, especially when transitioning from simulation-based tests to hardware-based validation. This is a well-known problem in the community and this is what the Portable Test and Stimulus Standard (PSS) is trying to address by defining a declarative domain specific language (DSL) intended for modeling scenario spaces of systems, generating test cases, and analyzing test runs [3] .

A. Current Practices

Some widely used FPGA simulation methodologies and testbench base-class libraries are Accellera UVM [4], OSVVM[5], UVVM[6], VUnit[7] and Python-based methodologies. An open-source simulation environment Python-based is Cocotb [8]. For testing FPGA bitstream functionalities common methodologies include real-time hardware in the loop simulation, built-in self-tests, embedded logic analyzers, among others.

B. Challenge

The main challenge is to get some of the key concepts from well-established methodologies such as UVM and PSS and reduce the gap between simulation and hardware test environment. This approach allows a small verification team to verify tens of parametrized firmware versions for SCUs and DLNs using open-source and in-house developed tools, without licensing costs and maintaining good test quality.

C. Our Approach

One of the key concepts behind UVM is to have modular testbenches with reusable components such as drivers, monitors, and scoreboards. The key concept behind PSS is to have a single representation of stimuli and test scenarios to be used across different validation stages. Lastly, to bridge the gap between simulation and hardware test environment we can use a common programming language like Python. These help to build a unified verification framework and approach.

III. Methodology Implementation

Starting with the engineering challenges and verification concepts mentioned above, this section details the actual implementation of a unified methodology and the verification metrics. The main goal has been to reduce the gap between simulation and hardware test environments with a reusable, consistent and automated verification framework. Using Python-based tools and a modular design approach, we developed a scalable solution to validate both DLN and SCU firmware.

A. Hardware Tester

In order to understand the hardware test environment, it is necessary to briefly describe the inputs and outputs of DLNs (yellow circles in Figure 1) and SCUs (green circles in Figure 1).

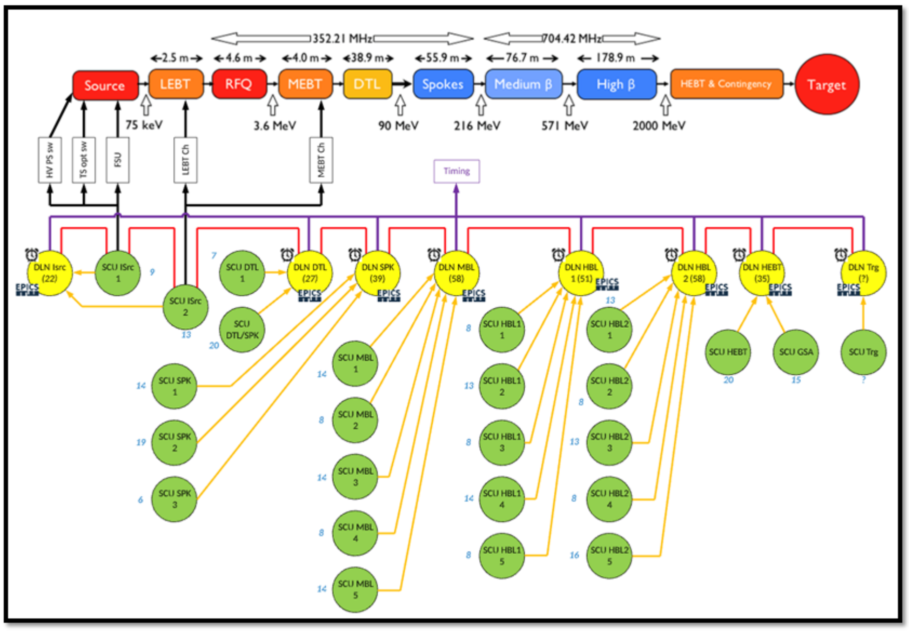


Figure 1. A diagram of the Fast Beam Interlock System.

Each DLN communicates with other DLNs and SCUs via optical links (orange arrows in Figure 1) , SFP modules, gigabit transceivers and Aurora protocol. The registers inside the FPGA can be accessed through the EPICS control system [9].

Similarly, each SCU communicates with a DLN via optical links, SFP modules , gigabit transceivers and Aurora protocol, and supports FPGA register access via EPICS. However, as mentioned in the first section, they also interface FPGA systems and PLCs. PLCs are connected using loop currents and PROFINET protocol, while communication with other FPGA systems is handled with RS485 signals, which comprise of discrete signals, Manchester encoded signals and proprietary Ethernet protocols.

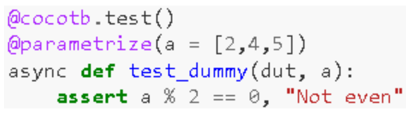
For the sake of clarity, few words on EPICS are mandatory to better follow what is written. EPICS [9] allows to abstract data exchange with heterogeneous devices, regardless of the drivers or protocols they interface with, as standardized process variables (PVs). Thanks to this uniformity of PVs, high-level tools and even some Python libraries can interact with all devices by the same mechanism.

To automate the FAT, the tester system must be able to emulate all the inputs previously listed and observe the corresponding outputs. Hardware-wise, the core of the tester is a mTCA crate equipped with the same types of boards housed in a DLN crate. The test setup additionally includes custom FMCs, redesigned FPGA firmware for SCU and DLN testers, and dedicated drivers that facilitate integration of the tester's firmware with the EPICS layer. To emulate the PLC inputs, the tester includes a PLC equipped with appropriate I/O modules, which can be driven with EPICS too.

In summary the tester can emulate all the inputs and observe all the outputs using python libraries part of the EPICS ecosystem[10]. This gives the opportunity to leverage Pytest [11] as testing framework.

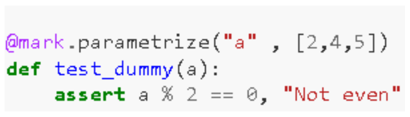
B. Cocotb and Pytest

The framework chosen to simulate the digital hardware designs is Cocotb, while Pytest is used for hardware testing. Both are Python-based. Even if Cocotb 2.0 is still in development, it has been chosen, since it is similar to Pytest. The key new features of Cocotb 2.0 relevant for this paper are: the use of python runner as the new way to execute simulations, "*cocotb.parametrize*" which replaces the TestFactory and is similar to Pytest' s parametrization feature and the ability to mark tests which makes it easier to select them based on custom labels or conditions.
The opportunity to parametrize the tests can be exploited for a uniform application of the same stimuli across both simulation and hardware environments. Figure 2 and Figure 3 show a basic example highlighting the strong similarities between Cocotb 2.0 and Pytest. In this example, the parameter is a short list placed close to the test declaration. There can be more parameters, and the lists can be stored in a shared library, i.e. in a JSON or an Excel file, and loaded during test running.



```python
@cocotb.test()
@parametrize(a = [2,4,5])
async def test_dummy(dut, a):
    assert a % 2 == 0, "Not even"
```

Figure 2. Cocotb parametrized test.



```python
@mark.parametrize("a" , [2,4,5])
def test_dummy(a):
    assert a % 2 == 0, "Not even"
```

Figure 3. Pytest parametrized test.

In both environments the tests are run with the ***pytest*** command. They can effortlessly integrate in GitLab CI/CD pipelines, utilizing the same runners and producing reports in a format consistent with simulation results in JUnit XML format .
At the time of writing this paper, Cocotb 2.0 which is under development, does not support utility libraries such as cocotb-coverage [12], pyuvm [13] or bus extensions as cocotbext-axi [14]. Simulation tests that rely on these packages right now cannot be run with Cocotb2.0.

C. UVM key concepts

The purpose of UVM is to support a standardized and reusable verification methodology. In this work, we believe that it is beneficial to adopt some of these concepts. The key ideas that we exploit are:

- Having a common mechanism to handle sequences and sequencers.

- Structuring both test environments with modular blocks like environment, driver, monitor and scoreboard, as shown in Figure 4.
- Exploiting a factory mechanism to enable component overriding end inheritance.

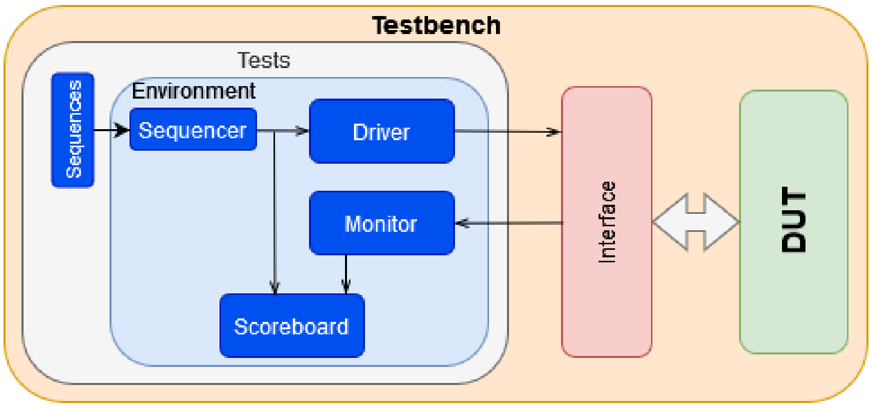


Figure 4. UVM testbench block diagram.

These key concepts have been implemented as follows:

- Sequences and sequencers are basically managed in both environments exploiting the parametrization of the tests.
- An Environment class provides a base structure for a modular verification as shown in Figure 5. The method *run_check_phase()* must be overridden by the subclass in both test environments. The driver and monitor are instance attributes that can be customized independently.
- The scoreboard is also an attribute instance. It can be defined as an abstract base class (Figure 6) with two attributes that store the value observed from the DUT and what is expected from a reference model. Moreover, it defines three abstract methods to interact with the reference model, the monitor and to verify the correctness at the end of the test.

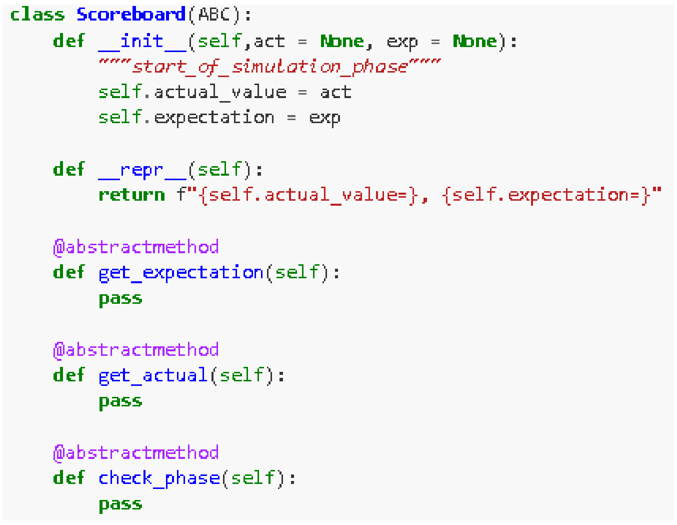
```
class Environment:
    def __init__(self, driver_func, monitor_func, Scoreboard_class):
        self.driver = driver_func
        self.monitor = monitor_func
        self.scoreboard = Scoreboard_class

        if self.__class__.run_check_phase is Environment.run_check_phase:
            raise NotImplementedError(f"{self.__class__.__name__} must be overriden")


    def run_check_phase(self, *args, **kwargs):
        raise NotImplementedError("Subclasses must implement run_check_phase.")
```

Figure 5. Base *Environment* class definition.

With this approach the Driver and the Monitor are tied to the simulation and the hardware test environment, while the scoreboard can be the same, or very similar, across environments.



```
class Scoreboard(ABC):
    def __init__(self,act = None, exp = None):
        """start_of_simulation_phase"""
        self.actual_value = act
        self.expectation = exp

    def __repr__(self):
        return f"{self.actual_value=}, {self.expectation=}"

    @abstractmethod
    def get_expectation(self):
        pass

    @abstractmethod
    def get_actual(self):
        pass

    @abstractmethod
    def check_phase(self):
        pass
```

Figure 6. Abstract Scoreboard class definition.

D. *PSS key concepts*

As previously mentioned, PSS is a standard that defines a way to describe verification scenarios to be portable and reusable across different platforms such as simulation and hardware testing. A test scenario can be viewed as which stimuli to apply and how to apply them to the device under test. In the PSS reference manual [3], it is detailed how to implement these scenarios, but in the context of this work only the main keywords and syntax elements are relevant. Up to this point, a light python-based framework has been introduced where JSON files can parametrize the tests. JSON structure can also be exploited to describe test scenarios.
It may be interesting to interpret and generate test cases directly from the JSON description. This can be implemented easily in Python , but this would require the development and maintenance of a dedicated toolchain, whose workload cannot be afforded. When open-source tools like Zuspec [15] becomes mature, it may be worth to adopt DSL approach instead of relying on JSON. In the context of FBIS, where test scenarios are not expected to be complex and heterogeneous, becoming proficient with DSL should not be too demanding. For now, it is enough to use this JSON as template for developing tests in simulation and in the hardware test environment. An example of such JSON is provided in the Case Study: Test of RISC-V Input Monitoring section.

E. *Specification Coverage*

In recent conferences, the importance of assuring that all requirements are verified has been highlighted [16]. In the functional verification process of the FBIS, once the System Requirements Specification (SRS) has been developed for a system, each of the individual system requirements is then broken down into a number of verifiable design specification items in a Detailed Design Specification (DDS) Document. The SRS and DDS items are linked through an SRS and DDS ID number for traceability [17].

Both in simulation and hardware test environment we get homogenous reports in Junit XML format, detailing the tests carried out. Creating a Requirement Traceability Matrix (RTM) in JSON or YAML format, where YAML may be more readable and descriptive, can allow consistent requirement or specification coverage checks across both environments. Using a unique RTM file and a shared python script, the check can be done regardless of the test environment.

F. *Verification Metrics and Completion Criteria*

We use three metrics to evaluate the tests performed:

- Requirements Traceability Matrix (RTM): Each test is linked to a DDS element and then to an SRS item. Verification is considered complete when all requirements in the RTM are covered by at least one successful simulation and hardware test.
- Scenario Coverage: V&V engineers specify test scenarios in tools like Excel, where each row specifies the input conditions to be applied and the expected outputs for the device under test. These scenarios are

translated into parameterized Cocotb and Pytest tests, ensuring that every functional behavior described in the specification is exercised during verification. In some cases, such as in the next section, explicit scenarios are not required because all possible input combinations are exercised.

- Code Coverage: HDL simulators provide coverage information. Coverage is collected during simulations to confirm that the executed test scenarios sufficiently exercise the firmware implementation. If coverage does not reach 100%, an evaluation is performed: additional scenarios can be added or uncovered code analyzed to determine whether it is unused code or features irrelevant to the current release.

Unlike randomized testbenches, our approach does not rely on functional coverage like those of cocotb-coverage. In fact, for hardware testing it is uncommon to apply randomized stimuli. Completeness is demonstrated through a combination of requirements coverage, deterministic scenario execution, and code coverage analysis.

IV. Case Study: Test of RISC-V Input Monitoring

As mentioned previously, the FBIS gathers data from approximately twenty different sensor system types. Almost each type has a specific set of redundant inputs and in some cases also different types of cables and communication protocols for interfacing. The FBIS must continuously monitor if the redundant inputs coming from a sensor system are consistent. This can be done at the level of the SCUs. Depending on which subset of sensor systems each SCU interfaces, its monitor configuration must be customized accordingly. This customization can be implemented by embedding a lightweight RISC-V CPU into the FPGA firmware.

The RISC-V CPU consists of four main blocks: a memory, an arithmetic-logic unit, an instruction decoder and a control unit. The scope of this section is not the testing of each single block, corner cases or fault injections, that sometimes can't even be carried out during the FAT, but the common tests that can be shared among simulation and hardware test environments. What can be shared is the checking of the functional correctness of logic. Figure 8 describes a simplified test scenario. The two main sections are *data* and *action*.

```
{
  "component": {
    "name": "redundancy_check_rflps_scu03",
    "test_name": "DDS_21_5_rflps",
    "data": {
      "input": [
        { "name": "slot", "type": "int", "domain": [0, 1, 2, 3] },
        { "name": "port", "type": "int", "domain": [0, 1] },
        { "name": "signal", "type": "string", "domain": ["Beam Permit", "Redundant Beam Permit"]
        { "name": "slot_value", "type": "string", "domain": ["OK", "NOK"] },
        { "name": "signal_value", "type": "string", "domain": ["OK", "NOK"] }],
      "bit": [{ "name": "no_redundant_system", "type": "bool" }]
    },
    "action": {
      "name": "run_check_phase",
      "inputs": ["slot", "port", "signal", "slot_value", "signal_value"],
      "activity": [
        {"do": {"action": "driver",
            "inputs": ["slot", "slot_value", "signal_value", "port", "signal"],
            "exec_body": "// Drive actual stimulus to DUT"}},
        {"do": {"action": "get_expectation",
             "inputs": ["slot_value", "signal_value", "no_redundant_system"],
             "exec_body": "// Determine expected output based on inputs and flags"}},
        {"do": {"action": "monitor",
            "inputs": ["slot", "port"],
            "exec_body": "// Observe DUT response"}},
        {"do": {"action": "check_phase",
            "exec_body": "{ assert(actual == expected); }"}}
      ]
    },
    "activity": {
      "do": "run_check_phase"
    }
  }
}
```

Figure 7. JSON-based test scenario.

Figure 8 and Figure 9 illustrate how this test scenario is translated into Cocotb and hardware test respectively. The data from the JSON description are converted into parameters for the test, and the sequence of actions is executed within the run_check_phase function.

```python
@cocotb.test()
@parametrize((("scu", "slot", "port"),  ru.get_system_params(RFLPS)),
              slot_value = [OK, NOK],
              signal = ["Beam Permit", "Redundant Beam Permit"],
              signal_value = [OK, NOK])
async def DDS_21_5_rflps(dut, scu, slot, port, slot_value, signal, signal_value):
    shift = (port-1) * 8 + ((get_data_type("scu_rflps_discrete_signals_index")[signal]-1)*2)
    await environment.run_check_phase(slot, port, slot_value, signal_value, shift)
```

Figure 8. Parametrized cocotb test function.

```python
@pytest.mark.rflps
@parametrize( serializer = ["SerializerB"],
              slot_value = [OK, NOK],
              signal = ["Beam Permit", "Redundant Beam Permit"],
              signal_value = [OK, NOK])
def DDS_21_5_rflps(self, serializer, scu, slot, port, slot_value, signal, signal_value):
   translated_signal = get_data_type("scu_discrete_signal_translator")[signal]
   test_environment.run_check_phase(scu,serializer,mc_type,slot,port,translated_signal,slot_value,signal_value)
```

Figure 9. Parametrized pytest test function.

Figure 10 and Figure 11 show how the *run_check_phase* has been implemented in both environments according to the test scenario and how scoreboard, driver and monitor are connected. The two *run_check_phase* implementations are very close in the two testing environments, but for different executions modes (async in Cocotb), different arguments and readability their implementations are kept separated.

```python
async def run_check_phase(self, slot, port, slot_value, signal_value, shift, no_red_sys = False):
    self.clock.start()
    self.driver(self.scoreboard.dut, slot, slot_value, signal_value, shift)
    self.scoreboard.get_expectation(slot_value, signal_value, no_red_sys)
    self.scoreboard.get_actual(await self.monitor(self.scoreboard.dut, slot, port))
    self.clock.stop()
    passed,msg = self.scoreboard.check_phase()
    assert passed, msg
```

Figure 10. Implementation in simulation environment (cocotb).

```python
def run_check_phase(self, scu, serializer, mc_type, slot, port, signal, slot_value, signal_value, no_red_sys = False):
    # Drive all the slot signals
    self.driver(mc_type,port,signal,slot_value,signal_value,to_drive_slot = True)
    # Drive the individual signal
    self.driver(mc_type,port,signal,slot_value,signal_value,to_drive_slot = False)
    self.scoreboard.get_expectation(slot_value, signal_value, no_red_sys)
    monitor_value = self.monitor(scu,serializer,mc_type,slot,port)
    self.scoreboard.get_actual(monitor_value)
    passed,msg = self.scoreboard.check_phase()
    assert passed, msg
```

Figure 11. Implementation in hardware testing environment.

Instead of the implementation of the three functions of the Scoreboard class, it is unique and shared across the environments (Figure 12).

```python
class ScoreboardSCURiscV(Scoreboard):
    def __init__(self, dut=None):
        super().__init__()
        self.dut = dut

    def get_expectation(self, slot_value, signal_value, no_red_sys):
        if no_red_sys:
            self.expectation = redundant_signals_check["DONT_CARE"]
        else:
            if slot_value == signal_value:
                self.expectation = redundant_signals_check["OK"]
            else:
                self.expectation = redundant_signals_check["NOK"]

    def get_actual(self, a):
        self.actual_value = a

    def check_phase(self):
        msg = f"actual_value = {self.actual_value}, expectation = {self.expectation}"
        return self.actual_value == self.expectation, msg
```

Figure 12. Implementation of the *Scoreboard* class.

V. Conclusions

So far, we have adopted this approach for some core functionalities**.** The main improvements include enhanced test reusability, increased efficiency in test development, and higher coverage in FAT testing. The ability for more people to review and understand the code, test methodology, and applied stimuli across both environments is an advantage. Furthermore, maintaining virtual test environments has become more efficient, sharing similar dependencies and having the opportunity to run tests into the same virtual environments.

This paper introduces a unified approach for the simulation and hardware testing of FPGA firmware. The proposed methodology has been developed in the context of FBIS, where the test space is quite constrained. This can be tailored to the specific needs of the system under test. By keeping a UVM-like structure in simulation and redesigning hardware testing methodologies around modular components, such as drivers, monitors, and scoreboards, it has become natural to reuse the code across environments. This includes sharing code, test sequences, and scripts all supported by the adoption of a common language. Moreover, a PSS-like test scenario used as a template can further align simulation and hardware test workflows. So far, FAT testing has had the greatest advantage from this integrated approach.

The transition from simulation-based verification to hardware testing presented some challenges, as maintaining consistent timing behavior across different environments. While the structure of drivers, monitors, and scoreboards are aligned, their implementations are very different because of different time scales and observability in hardware. To address these challenges, parameterized scenarios are first validated in simulation, ensuring both coverage and requirements traceability. In hardware, the same requirement-driven test scenarios are executed, with the focus shifted in validating functionalities under real operating conditions. This approach helps to minimize the effort.

Finally, the adoption of a standardized reporting format has simplified test result comparisons and improved traceability.

References

[1] https://ess.eu/about

[2] S. Pavinato, et al. "The ESS Fast Beam Interlock System - Design, Deployment and Commissioning of the Normal Conducting Linac", ICALEPCS2023, Cape Town, South Africa

[3] PSS 3.0 Language Reference Manual

[4] https://accellera.org/downloads/standards/uvm

[5] https://osvvm.org/

[6] https://www.uvvm.org/

[7] https://vunit.github.io/

[8] https://www.cocotb.org/

[9] https://epics-controls.org/

[10] https://epics-base.github.io/p4p/index.html

[11] https://docs.pytest.org/en/stable/

[12] https://cocotb-coverage.readthedocs.io/en/latest/introduction.html

[13] https://pyuvm.github.io/pyuvm/

[14] https://github.com/alexforencich/cocotbext-axi

[15] https://github.com/zuspec

[16] Get the right FPGA quality through Efficient Verification and Requirements Tracking, SEFUW 2025, ESTEC. Espen Tallaksen.

[17] A. Nordt, et al. "Verification and Validation of the ESS Machine Protection System-of-Systems (MP-SoS)", ICALEPCS2023, Cape Town, South Africa