2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

Closed-Loop Model-First SoC Development With the Intel® Simics® Simulator

Kalen Brunham, Programmable Solutions Group, Intel, Toronto, Canada (kalen.brunham@intel.com)

Anthony Moore, Programmable Solutions Group, Intel, Chandler, AZ (anthony.w.moore@intel.com)

Tobias Rozario, Programmable Solutions Group, Intel, Toronto, Canada (tobias.rozario@intel.com)

Wei Jun Yeap, Programmable Solutions Group, Intel, Penang, Malaysia (wei.jun.yeap@intel.com)

Jakob Engblom, Software and Advanced Technologies Group, Intel, Stockholm, Sweden (jakob.engblom@intel.com)

***Abstract*—Virtual platforms (VP) promise the ability to develop and test software prior to hardware being available. In model-first VP development, the architectural exploration phase of a project is done using transaction-level models (TLM) such that these TLMs represent an executable specification and are used to drive both the hardware and software design. While the promised schedule acceleration is often irresistible to managers, the practical execution of model-first development exposes some real technical challenges that must be solved for it to be successful. Specifically, how does one ensure that the model, the natural language specification, the software, and the hardware design are all in sync? In this paper we describe our experience performing model-first development of an SoC using the Intel® Simics® simulator. We present how Simics models can be embedded into universal verification methodology (UVM) testbenches both as verification IP and as hardware block replacements and how to use this technique to successfully ensure that the Simics model and the hardware are consistent, achieving what we are calling RTL-model consistency.**

Keywords—Simics, FPGA, simulation, virtual platform, co-simulation, verification, RTL, UVM

I. INTRODUCTION

Using virtual platforms (VP) as part of SoC design and verification flows is standard practice amongst large engineering organizations such as Intel [1][2]. VP tools such as the Intel® Simics® simulator or QEMU [3] allow design teams to create fast transaction-level models (TLM) of their systems, including both processors and peripherals, and to use the models to run the exact same software binaries that would run on the hardware. VPs enable software development to occur prior to hardware availability [4] and augment the available execution capacity for specialized hardware in the post-silicon timeframe [5]. A VP for a new hardware design is typically developed by combining existing models for reused hardware blocks with new models for new or updated blocks. The longest pole, and greatest risk, in this development process is the creation of the VP models, development of software, and writing the register transfer level code (RTL) for the **new and updated blocks** in the design.

For each new hardware block, several different engineering teams need different artifacts of the specification to complete their pre-tape-out deliverables. The hardware tape-out requires the fully validated and physically implemented RTL for the hardware design. Software development teams need fast TLM-based VPs that reflect the hardware design to develop and test their software. RTL validation teams need a specification of expected behavior to create testbenches, and the RTL for the design to validate using those testbenches. Critically the software and hardware artifacts must be consistent with each other, to avoid having to rework the software (or worse, hardware) once actual hardware becomes available and final testing is performed.

Conceptually, there should be a single “master specification”, as shown in Figure 1A, which all engineering teams work from. However, these documents are typically written in a natural language like English and are thus open to human interpretation. Different interpretations and misunderstandings are prone to happen and will most likely cause issues, pain, and rework when artifacts from multiple teams are integrated. To ensure that the software developed on a VP also works on the hardware (or RTL) when it arrives, the software, VP models, and RTL must all be consistent with the hardware specification [6], as illustrated in Figure 1B. Only if this consistency is guaranteed can we fully realize and benefit from "shift-left" software and system development.

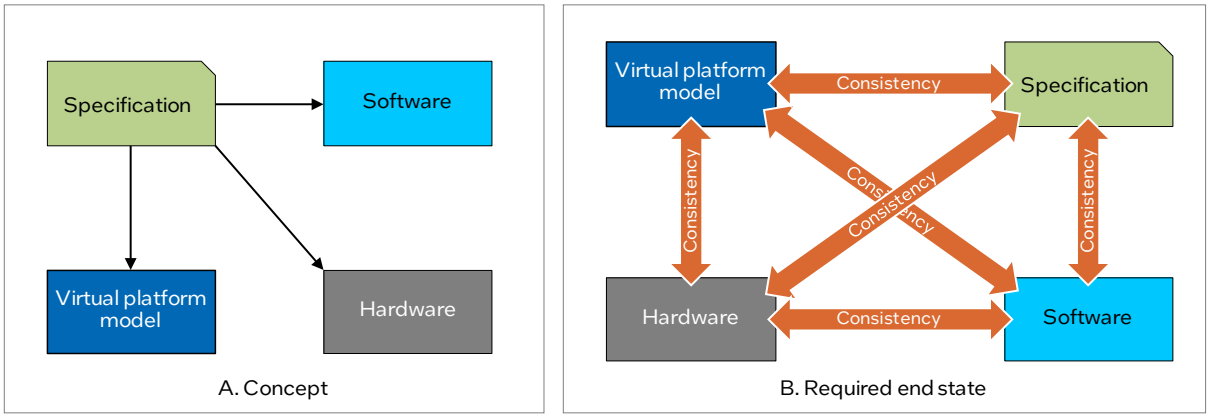

Figure 1: Relationship of specification, VP, software, and hardware

A. Ensuring Consistency

To ensure consistency between the artifacts, our approach is to use the VP model as a golden reference model for the other deliverables, as shown in Figure 2, i.e., **working model first** (similar to what many other teams [7] are doing). In model-first, **the fast TLM VP model acts as an executable specification**. The VP model is used to develop and test the software and validate the RTL. This methodology avoids the problems with different interpretations of the spec, as it makes the interpretation of the VP team the correct interpretation. It does, however, necessitate that the specification and VP model are in sync as the specification evolves, and that there is an organizational understanding/acceptance that the VP model will evolve over time. Indeed, the VP model is not final until the hardware design is frozen.

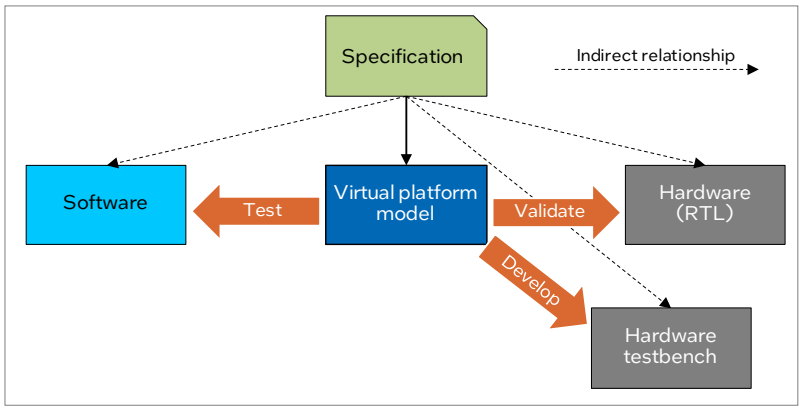

Figure 2 : VP as Golden Reference model

Two key assumptions are made regarding the "golden" VP models in this work. The first assumption is that the models are accurate at the transaction level. A second assumption is that software that runs on these models can run on the hardware, or on RTL simulation/emulation of the hardware, without modification. Based on these assumptions, software and RTL validation teams can begin using the VP models as soon as they are available. This achieves a shift-left in software development and the early creation of RTL verification testbenches.

The RTL testbench is key to the model-first methodology. RTL testbenches apply stimulus to a piece of RTL, known as the *design under test* (DUT), and use a model to predict the output of the DUT. The prediction and observed output are compared to determine if a test passes (i.e., if the hardware under test is correct). The DUT in the testbench can be a single hardware block or can be a subsystem of hardware blocks integrated together.

We use the VP model as the testbench predictor model to enforce consistency between the model and the RTL. In addition, the VP models can be used as a replacement for their respective blocks in the hardware design before the RTL of the hardware block is available. This replacement is made tractable as our modeling methodology aligns the logical hierarchy of the hardware design and the VP model hierarchy.

The result of replacing hardware blocks whose RTL is not yet available with the VP model enables the realization of a complete testbench very early in a program using only the executable specification and existing RTL. Replacing hardware blocks in the DUT with VP models is also beneficial once RTL starts to appear–by replacing parts (hardware blocks) in the DUT with VP models, it is possible to run tests faster and to test hardware blocks that depend on RTL for hardware blocks that are not yet available.

B. A Constantly Updated Virtual Platform Model

Writing golden models for use by hardware and software teams following model-first development means writing the VP model as the specification itself is being written and updated. This approach requires an integrated effort instead of a serialized one (where an architect writes a specification and then hands it off to a modeling team, "after it is done"). It also requires a mindset shift towards a more iterative development methodology around RTL verification components used in a testbench.

Our approach is to integrate modelers into the hardware architecture team. The modelers continuously create and update the VP models as the hardware architecture evolves. The models keep pace with internal drafts of the specification and do not wait for official releases. Indeed, the models are part of the release of any specification drop. This causes some additional work in modeling as the architecture evolves; specifically, the area under the effort curve is greater, but the benefit is that the model is always available and in sync with the spec.

Consumers of the model can begin using it early, but they need to be aware of the model status and that it will evolve over time. In point of fact, driving the understanding that the model will be updated several times as the specification evolves was an initial challenge for us across the engineering organization since previous projects have been based on a frozen specification. This updated understanding has been most pronounced in the RTL validation world where the expectation of a rarely changing predictor in the scoreboard was previously the norm.

II. EMBEDDING INTEL SIMICS SIMULATOR VP MODELS IN UVM TESTBENCHES

The key technology enabling model-first is the embedding of VP models into RTL testbenches as both predictor and a portion of the DUT, as shown graphically in Figure 3. RTL validation at Intel is typically performed using SystemVerilog and the universal verification methodology (UVM) [8]. UVM testbenches generally instantiate a DUT connected to verification intellectual property (VIP) components that drive and monitor traffic. A key component in a UVM testbench is the *scoreboard* which receives monitored transactions for the inputs and outputs of the DUT, runs the input transactions through a reference model based on the specification (the *predictor*) to produce the expected output transactions, and then compares the expected output to the observed output.

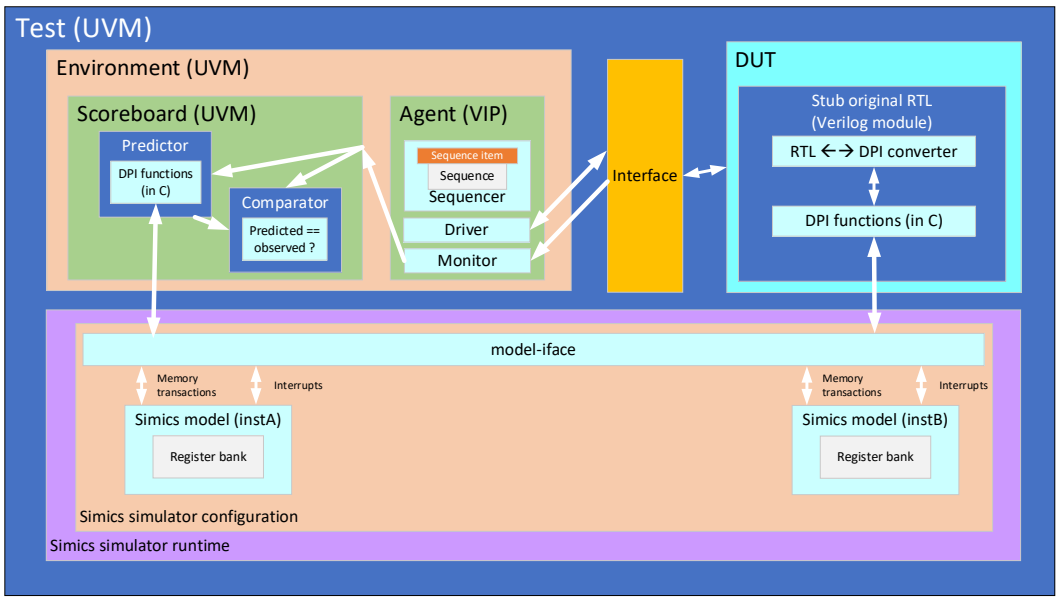

Figure 3 : Block diagram of Simics as an RTL replacement and as the predictor in a scoreboard

Traditionally, the predictor is implemented using a high-level language such as SystemVerilog or SystemC, whereas, in this work, we have chosen to use an Intel Simics virtual platform model. We use the Intel Simics virtual platform model as the scoreboard predictor, and/or as a replacement of one or more hardware blocks in the DUT ahead of RTL being available.

The Intel Simics simulator manages models as opaque binaries using the standard host operating system C ABI. This makes the external interface of the models language-independent, allowing models to be developed in a variety of languages including C, C++, SystemC, Python, and the Device Modeling Language (DML) [9]. The language of the model does not matter at runtime, and users of a model do not need access to the source code of the model. All that is needed is access to the definitions of the interfaces exposed by the model. This is different from existing industry solutions for VP-RTL co-simulation which typically rely on VP models being written in a particular language and having the source for the models available. Such an approach would be too limiting for our needs.

Instead, our solution is to embed the Intel Simics simulator and a set of models as a VP configuration into an RTL simulator. Embedding is a standard feature of the Intel Simics simulator, used by many teams at Intel and outside. The embedded simulator communicates with the RTL simulator over the SystemVerilog direct programming interface (DPI), which is a standard feature in commercial RTL simulators. The DPI capability enables interfacing between SystemVerilog and C, where SystemVerilog can call C functions and where C functions can also call SystemVerilog functions. This enables interacting with the Intel Simics simulator APIs, which are available in C.

III. Implementation Details

The DPI API created in our approach provides hardware-protocol-agnostic generic SystemVerilog functions that exposes memory-mapped transactions, basic signals, and interrupt buses in the embedded virtual platform configuration towards the rest of the testbench environment. These DPI functions are sufficiently generic for sending and receiving transactions to the VP models such that they can be converted into protocol or VIP-specific SystemVerilog classes for use in a UVM scoreboard or for interfacing with bus functional models (BFM). Concretely, an Intel Simics model that exposes an Intel Simics simulator memory transaction interface can ultimately be exposed as a cycle-accurate hardware interface such as AXI, APB, OCP, etc., or easily digested by a UVM transaction object. The use case does not affect the VP model, and this VP model is the same model used for software development.

The DPI functions ultimately facilitate communication initiated from RTL to Simics, and from Simics to RTL, for both the hardware block replacement and UVM scoreboard use cases; the flow of transactions is shown in Figure 4. On the VP side, the interfaces to DPI are implemented by a VP module called *model-iface*. There is only one model-iface instance in the entire VP configuration and the connectivity from model-iface to the target Intel Simics model is compiled in the specific model-iface created for the specific VP. This module combined with the DPI API is conceptually an hourglass that exposes many SystemVerilog interfaces, through the few generic model-iface interfaces, into the many interfaces of the different VP model instances in the VP.

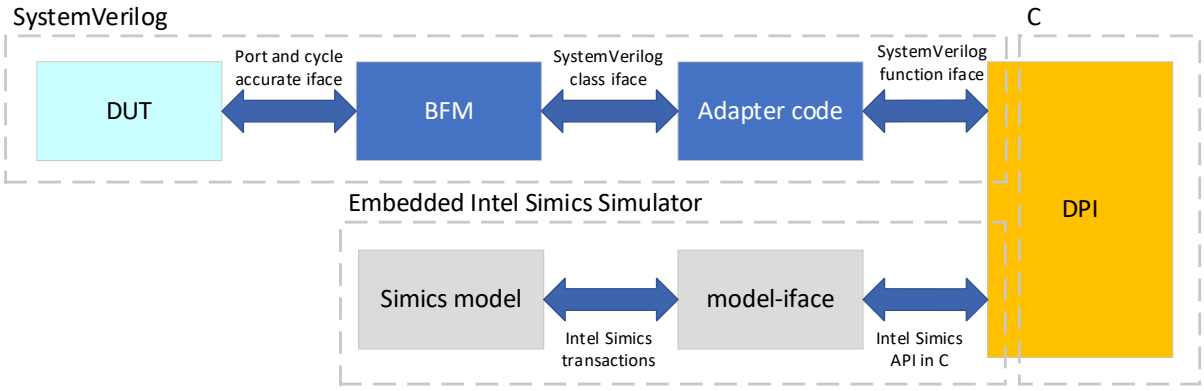

Figure 4 : Flow diagram of RTL through DPI to the VP model

When using a Simics model as a hardware block replacement, a SystemVerilog stub module must be created where the port types and names exactly match the ultimate DUT RTL. Additional adaptation logic or BFMs are used internally in the stub module to implement a transactor and convert the Verilog ports to data types that can call the generic DPI functions. These BFMs can be commercial VIP, or custom created. The BFM handles the conversion logic from cycle-based transaction to class-based transaction depending on the interface's protocol. Adapter code in SystemVerilog is then used to call the DPI API for communication to the VP model. The return values of the DPI function are then used to create the response back to the RTL.

When using a VP model as a predictor in the UVM scoreboard, cycle-based conversion logic inside the scoreboard is not necessary, since all monitored transactions captured in the scoreboard are already at transactional level. The predictor can directly use information extracted from the monitored transaction to construct the required DPI function call, which will then be sent to the VP model. The response from the VP model, which will be the predicted response, can then be pushed into a queue in the scoreboard to be used for comparison when needed.

A. DPI Implementation Details

The DPI functions in our implementation generally fall into three categories: 1) Setup and initialization, 2) time synchronization, and 3) transaction handling. Setup and initialization encompass the creation of an Intel Simics simulator instance and the loading of the simulator configuration into the simulator; a sample code snippet is shown in Figure 5. For any given RTL simulation, there is only ever one instance of the embedded Intel Simics simulator. As a result, the target script executed by the embedded Intel Simics simulator must contain instances for all the different models needed across the entire testbench, and the model-iface.

Time advancement in the Intel Simics simulator time domain is required for VP models that post events in the future. For example, if a simple device that performs a hash over a set of data takes time before signaling a "done" interrupt, then the model would post an event callback that asserts the interrupt signal after a defined amount of time. If Simics time never advances, then the interrupt would not be observed by the RTL simulator. To drive time in the Simics domain, we instantiate a simple clock device in the VP. This clock device is configured with a frequency matching the DUT in the RTL testbench. A DPI function wraps the Intel Simics simulator API call *SIM_continue()*, exposing it to SystemVerilog. This DPI function is then called for every positive clock edge in the RTL simulator time and thereby advancing time in the Simics domain. This ensures a tight coupling of RTL and Simics time.

```
// Initializes the Intel Simics simulator and runs given target script
void start_simics(const char* target_script) {
     …
     SIM_init_environment(argv,1,1);
     …
     // Initialize the simulator
     SIM_init_simulator2(init_args);
     …
     // Run the Intel Simics simulator target script – sets up the Simics configuration for the test
     SIM_run_command_file(target_script, true);
}

// Advance Simics time
void advance_simics_sim(uint32_t num_cycles)
     …
     SIM_continue(num_cycles);
     …
}
```

Figure 5 : Code snippets in C from the Intel Simics simulator DPI setup and initialization functions

B. Model-iface Implementation Details

Transactions initiated by RTL to the VP model call the associated generic DPI function for the interface type, with parameters to identify the specific interface and the details of the transaction. The model-iface VP module exposes to the DPI functions separate interfaces for each interface type, i.e. memory-mapped, interrupt, signal, etc. via the Intel Simics Simulator C API. These DPI functions take as a parameter a unique interface identifier (ID) where the model-iface uses the interface ID to route the transaction to the specific interface of a specific model instance in the VP configuration and then to send the response back to the caller.

The relevant source code for the DPI function of a memory-mapped transaction is shown in Figure 6. The *obj* and *intf* parameters are required to direct the transaction to the correct model and interface of that model via the model-iface.

```
// Function for memory read or write transaction from RTL calling into Simics model
int iface_in_mem_trans(conf_object_t *obj, const transaction_interface_t* intf, uint32_t addr, uint8_t* data, uint32_t len, int
is_write){

    uint8_t* buf = (uint8_t*)malloc(len * sizeof(uint8_t));

    atom_t atoms[] = {
        //… setup transaction params in atoms
    };
    transaction_t t = { atoms };

    if (is_write) {
        //… send data if it is a write
        SIM_set_transaction_bytes(&t, wrdata);
    }

    // Generate transaction
    int ret = intf->issue(obj, &t, addr);
    if (!is_write) {
        //… populate data if it is a read
        SIM_get_transaction_bytes(&t, rddata);
    }
    free(buf);
    return ret == Sim_PE_No_Exception;
}
```

Figure 6 : Code snippets from mem transaction C DPI functions

The more complex transaction case is where the VP model needs to initiate a transaction towards the RTL. For this situation, the RTL defines and exports SystemVerilog functions via DPI to the embedded Intel Simics simulator which are called by model-iface. In our implementation, we create these functions inside a SystemVerilog module called *simics_manager*, which is instantiated at the top level of the testbench. The model-iface module provides the hook to enable the VP model to call into simics_manager functions. The model-iface has special initialization code to resolve the names of the exported DPI functions found in simics_manager using *dlsym* and calls these functions when it receives transactions from the model; a sample code snippet is shown in Figure 7. In this example, the *sig* port of the model-iface calls the *iface_out_sig_trans* function, which is a function pointer initialized when the virtual platform is started and is assigned to be the symbol for the DPI function called *iface_out_sig_trans* defined in SystemVerilog.

```
// Function pointer that the model-iface calls when model sends transaction
extern int (*iface_out_sig_trans)(int interface_id, uint64_t addr, uint8* data, uint32 len, int is_write );

// model-iface port used to send Simics model transaction to RTL
port sig[i<NUM_PORTS] {
    implement signal {
        method signal_raise() {
            log info,4: "signal_raise()";

            // send transaction
            iface_out_sig_trans(I, 1);
        }
        method signal_lower() {
            log info,4: "signal_lower()";

            // send transaction
            iface_out_sig_trans(I, 0);
        }
    }
}

// Run during Simics initialization to resolve function pointer location within the RTL
static void connect_iface_out() {
    /* open the needed object */
    void* handle = dlopen(NULL, RTLD_LOCAL | RTLD_LAZY);

    /* Resolve function location*/
    if((iface_out_sig_trans = dlsym(handle, "iface_out_sig_trans") )) {
        printf("SUCCESS: resolved symbol iface_out_sig_trans\n");
    } else {
        printf("INFO: symbol iface_out_sig_trans not found – assuming running Simics standalone (outside RTL sim)\n");
        iface_out_sig_trans = iface_out_sig_trans_local;
    }
}
```

Figure 7 : DML Code snippet from model-iface

Whenever a SystemVerilog function is called from model-iface, the SystemVerilog function stores the transactions received into a SystemVerilog queue. A sample code snippet of the SystemVerilog function for a

simple signal interface is shown in Figure 8. The queue is accessible from anywhere in the testbench by placing it in the UVM configuration database. This allows the scoreboard to pop the transactions from the queue for comparison purposes, or to allow the stubbed RTL module to pop transactions and send out cycle-accurate transactions when using the VP model as a hardware block replacement.

```
export "DPI-C" function iface_out_sig_trans;

//------------------------------------
//Interrupt Handler
//------------------------------------
function iface_out_sig_trans(input int interface_id, input int is_raise);

    string call_info;
    call_info = $sformatf("[iface_out_sig_trans] interface id = %d; is_raise = %d\n", interface_id, is_raise);
    //`uvm_info("iface_out_sig_trans", call_info, UVM_LOW)
    $display(call_info);

    //Push interrupt transaction into the queue denoted by interface_id index
    simics_q.interrupt_if_queue[interface_id].push_back(is_raise);

endfunction
```

Figure 8 : Code snippet from the simics_manager of a SystemVerilog function for handling transactions coming from the model-iface

IV. RECORD-REPLAY DEBUGGING

Our implementation also provides an easy way to reproduce and debug issues reported during RTL simulation using a record-replay methodology, as illustrated in Figure 9. Since the model-iface module acts as a proxy, it receives all transactions to and from the VP model and can thus perform the following functions; 1) records transactions to a file, 2) plays back transactions in an Intel Simics simulator-only environment for VP model debug, 3) directs transactions to the specific interface of the specific target VP model, and 4) redirects transaction responses from the VP model to the RTL simulator via the interface-specific SystemVerilog DPI function.

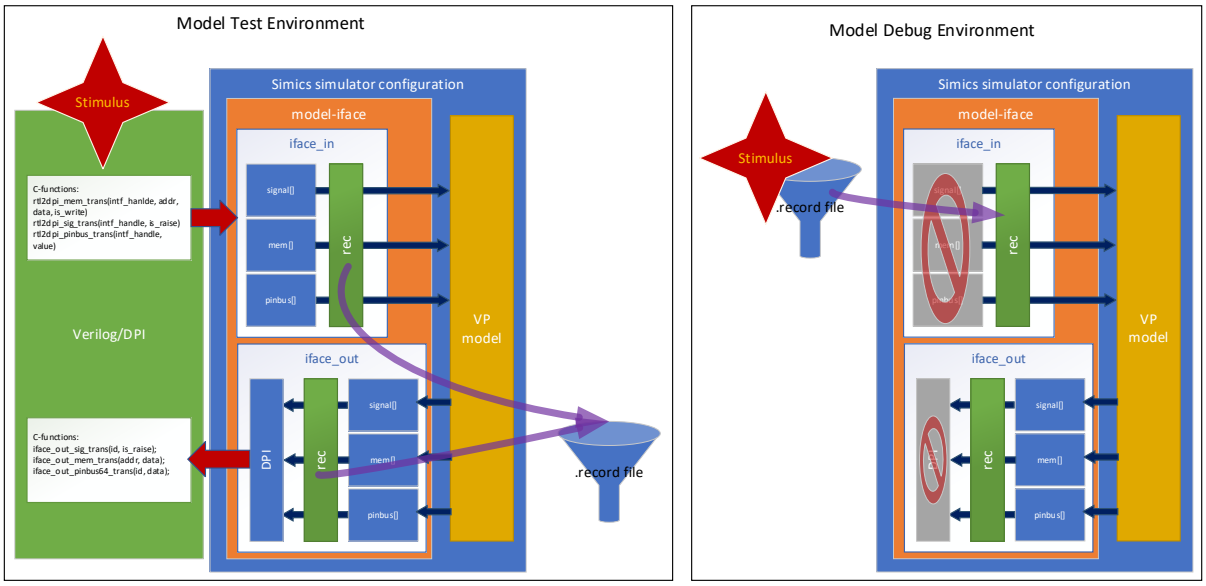

Figure 9 : Block diagram of model test and model debug environment

Recording works by serializing all the transactions that pass through the proxy and sending it to a *recorder* object, a standard feature of the Intel Simics simulator. The recorder saves the transaction timestamp and contents to a text file. Playback is achieved by loading the same simulator configuration used in the embedded Intel Simics simulator into an Intel Simics simulator-only environment and setting the recorder in playback-mode. In this mode, the recorder issues each transaction as specified by the recording file at the point in simulated time indicated by the timestamp. The model developer can then use traditional techniques for debugging the VP model, including using simulator logging and connecting a software debugger to the Intel Simics simulator.

V. RESULTS

The methodology described in this paper has enabled us to achieve the following key results:

- **Use of the VP as the golden reference for software and RTL development and testing**. The implementation presented in this paper embeds the Intel Simics simulator framework along with a VP model into an RTL-level simulator. It uses a VP model as the UVM predictor when testing RTL, providing the ability to maintain consistency between the RTL and the VP model. This consistency mitigates the risks for bugs and spec misalignment when software teams transition from using the VP model to RTL and eventually hardware.
- **Earlier testbench development and RTL validation speedup**. Embedding the VP model into the testbench DUT as a temporary replacement for the RTL while it is unavailable allows the RTL validation teams to start building testcases and testbenches earlier.
- **VP model and software debug in a standalone environment**. The model-iface can use the Intel Simics simulator recorder functionality to save a recording of all inputs to the VP models. When RTL developers and testers run into testbench failures, they can provide the VP model developers the recording, who can then replicate and analyze the issue in a standalone VP simulation without the need for the RTL testbench infrastructure.

We have found the methodology to have the following limitations:

- **RTL developers and testers have limited VP model visibility**. Currently, engineers analyzing the results of an RTL simulation do not have a means of inspecting internal states and signals of the Simics model. The debugging and viewing of signals in the RTL simulator are only on the boundary of the VP model and require the support of VP model developers for internal debugging.
- **Abstraction-level limitations**. The VP models might be unable to mimic certain hardware behavior such as error response signals that are specific to different hardware protocols like OCP and AXI.

VI. CONCLUSIONS

In this paper, we've presented a technique for model-first development using fast VP models created using the Intel Simics simulator as the executable specification for an SoC. To guarantee that the specification, hardware design, and software are all in sync, these models are consumed by an embedded Simics simulator inside a standard UVM scoreboard, thereby closing the loop between the models and the RTL. These models can also be used as a replacement of a hardware block prior to the RTL being available to shift-left RTL testbench development. This has allowed us to use the models for early software development with the confidence of model-RTL consistency.

REFERENCES

[1] Daniel Aarno and Jakob Engblom, Software and System Development using Virtual Platforms*:* Full System Simulation with Wind River Simics. Waltham, MA: Elsevier, Morgan Kaufmann Publishers, 2015.

[2] Kalen Brunham and Jakob Engblom, "Challenges and solutions for creating virtual platforms of FPGA and SASIC designs". Design and Verification Conference and Exhibition Europe (DVCon Europe), 2022.

[3] Fabrice Bellard, “Qemu, a fast and portable dynamic translator” in Proceedings of the 2005 USENIX Annual Technical Conference, Vol. 41. 2005.

[4] David Black, Jack Donovan, Bill Bunton, and Anna Keist, SystemC From the Ground Up. New York, NY, Springer, 2010.

[5] Jakob Engblom, “Continuous integration for embedded systems using simulation”. Embedded World 2015 Congress, Nürnberg, Germany, 2015.

[6] Jakob Engblom and Ola Dahl, “Verification of virtual platform models – What do we mean with good enough?”. Design and Verification Conference and Exhibition Europe (DVCon Europe), 2022.

[7] Ericsson, "TLM use cases at Ericsson AB, in Case Studies in SystemC". Design and Verification Conference and Exhibition Europe (DVCon Europe), 2014.

[8] Accellera, Universal Verification Methodology (UVM) 1.2 User’s Guide. Elk Grove, CA, Accellera , 2015.

[9] Device Modeling Language, https://github.com/intel/device-modeling-language