A UVM-based Approach for Rapidly Verifying Digital Interrupt Structures

Christoph Rumpler, Alexander W. Rath, Sebastian Simon
Infineon Technologies AG
85579 Neubiberg, Germany
{firstname.lastname}@infineon.com

Heinz Endres
University of Applied Sciences Würzburg-Schweinfurt
97421 Schweinfurt, Germany
Heinz.Endres@fhws.de

Abstract

Nowadays, the development of integrated circuits must be accomplished in short time frames in order to meet high market demands. However, the complexity and miniaturization of these circuits is steadily increasing. Due to these factors pre-silicon verification of such highly integrated systems is a major task during development. In this respect, goals like tight deadlines and high coverage results comes together with greater chip functionality and higher complexity. To take up this challenge, it is indispensable to continuously improve the verification processes and the techniques associated with them. Thus reusing of Verification Intellectual Properties (VIP) and automated testbench generation is inevitable. However, up to now, interrupt structure verification of processor based designs is still barely automated and normally done manually by verification engineers. This paper introduces a new approach to accelerate the process of interrupt structure verification by generating testbench and test sequence automatically.

Index Terms

SystemVerilog, Universal Verification Methodology, System on Chip, Interrupt Verification

I. Introduction

Due to the advances in modern electronic manufacturing technologies, the integration of peripherals and processors into a single chip has become an ongoing trend in many application domains. The high number of peripheral modules leads to an increasing number of interrupt sources and to complex interrupt structures in modern System On Chip (SOC) solutions. In order to guarantee the correct functionality of such interrupts, an extensive pre-silicon verification on module and SOC level is indispensable. Interrupt systems are usually divided into interrupt structure (hardware) (see Figure 1) and service routines (software); both to be verified separately. For verifying the latter, automated solutions which are able to check the correct interrupt functionality regarding interrupt priority and handling exist already. However, there is no standardized verification approach available we are aware of that is able to verify complex interrupt structures on module or top level. Such a verification approach would have to address the correctness of the connectivity between the interrupt sources (e. g. a timer peripheral) and the SOC's microcontroller as well as the functional elements that accompany the connectivity (e. g. register controlled interrupt enable switches or status registers). As of now, the verification of such structures has to be done manually and is therefore a time-consuming and error-prone procedure.

In order to accelerate this process, we will present a new approach for verifying interrupt structures in processor-based designs by using automatically generated testbenches and moreover deploying the newest methodology standard for digital domains – Universal Verification Methodology (UVM) [1].

Our approach is based on a meta model of interrupt structures, i. e. a model that can describe *any* concrete interrupt structure. It can be used to build an individual interrupt structure of the target design, comprising specific information. Out of this structural information, stored in an Extensible Markup Language (XML) file, we are able to generate code for the testbench and test sequences (see section III). The generated code contains individual Verification Components (VC), needed for a UVM-based testbench, and test stimuli respective to the design. With these components we are able to verify each path of the interrupt structure individually by dedicated test sequences. For a realistic test procedure the generated code provides multiple test scenarios like pending and non-pending interrupts. Finally the evaluation of these interrupts is performed during the tests by predicting and monitoring the interrupts dynamically.

II. Related Work

During the last years UVM has become a de facto standard for the verification of digital designs. In order to speed up the procedure of building up a UVM environment, VIPs for standard interfaces as well as tools, which generate UVM testbench templates, were developed. However, up to now, interrupt structure verification of processor based designs is still barely automated and normally done manually by verification engineers. Several approaches were proposed in the past to additionally automate this particular verification task.

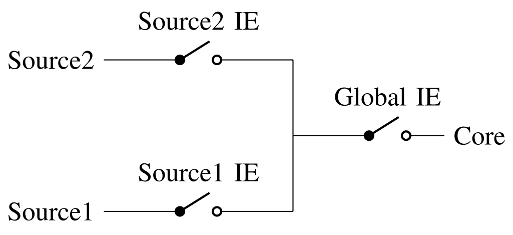

Fig. 1: Excerpt of an interrupt structure. The sources represent hardware that generates interrupts whereas the core represents an interrupt input of microcontroller. The connection from a source to a core is called an interrupt path. The switches activate or deactivate certain sections of the paths and are controlled by IE (interrupt enable) registers. In reality, such structures are typically much bigger than the example provided in this figure, e. g. 100 sources and 20 core nodes connected by a complex connectivity tree.

In [2], [3], the authors present a specially developed tool called Processor External Interrupt Verification Tool (PEVT) that is capable of generating test cases, appropriate stimulus and checkers. In addition, an architecture description language is introduced to formally define the interrupt behavior of the processor under verification. However, the resulting verification environment is only making use of Verilog. Therefore it lacks of various capabilities that would come along with SystemVerilog or UVM like object-oriented programming, covergroups, drivers, monitors and many more.

In contrast to this, the authors of [4] propose a UVM-based solution for the verification of interrupts in an Intellectual Property (IP). The downside here is the missing meta description that formally comprises all possible realizations of interrupt structures. For this reason, the UVM environment can not be generated from a user-defined structure description, but is set up by deploying predefined interrupt verification IPs. Apart from this, there are no covergroups generated.

Our approach combines the advantages of the aforementioned approaches while compensating their drawbacks. On the one hand, the generated verification benefits from the SystemVerilog language features and the concepts of UVM. On the other hand, we are able to handle every possible interrupt structure by leveraging the meta-modeling ideas from [5]. Further details on the verification components and the constructed meta model are described in the following section.

III. Implementation

In the very beginning of this section we will describe the meta model of interrupt structures and the process of generating testbench and test sequences. After that, the structure of the verification environment and certain tasks like monitoring and evaluation inside the scoreboard are explained in more detail. Also, the different test scenarios and the individual test sequences will be discussed. At the end of this chapter we will show how test results are covered.

A. Meta Model and Code generation

To generate a verification environment for interrupt structures of a Design Under Verification (DUV) some specific information are necessary. On the one hand we need information like number and names of *Source-* and *CoreNode* (cf. Figure 1) in order to generate the verification components, e. g. one unique *uvm_monitor* for each node [6]. On the other hand we need information about registers and the way they are connected with the nodes to be able to generate specific test sequences; one for each interrupt path.

We decided to use Unified Modeling Language (UML) for creating the meta model of interrupt structures. In order to get a meta model that is valid in general, we only used elements that are really necessary from verification point of view and part of most interrupt structures. The meta model is split into two logical sections to separate the information needed for testbench generation and test sequence generation. First section is called *Component* and contains elements like *Port*, *Register* and *Field*. With these elements the user is able to create common subcomponents that can be part of the overall interrupt structure. Second section is called *InterruptPath* and contains *Line*, *SourceNode*, *CoreNode* and multiple *RegisterSets* (cf. Figure 1). This is needed to model the connections between components. In Figure 2 you can see the relations between the previously explained elements in a UML class diagram. With the help of this meta model the example from Figure 1 can be composed as follows:

- One instance of *InterruptStructure* contains ...
  - Two instances of *InterruptPaths* which consist of ...
    - Three instances of lines between ...
    - Two instances of *SourceNodes*
    - One instance of *CoreNode* and
    - Three instances of *Interrupt Enable* (IE) *RegisterSets*

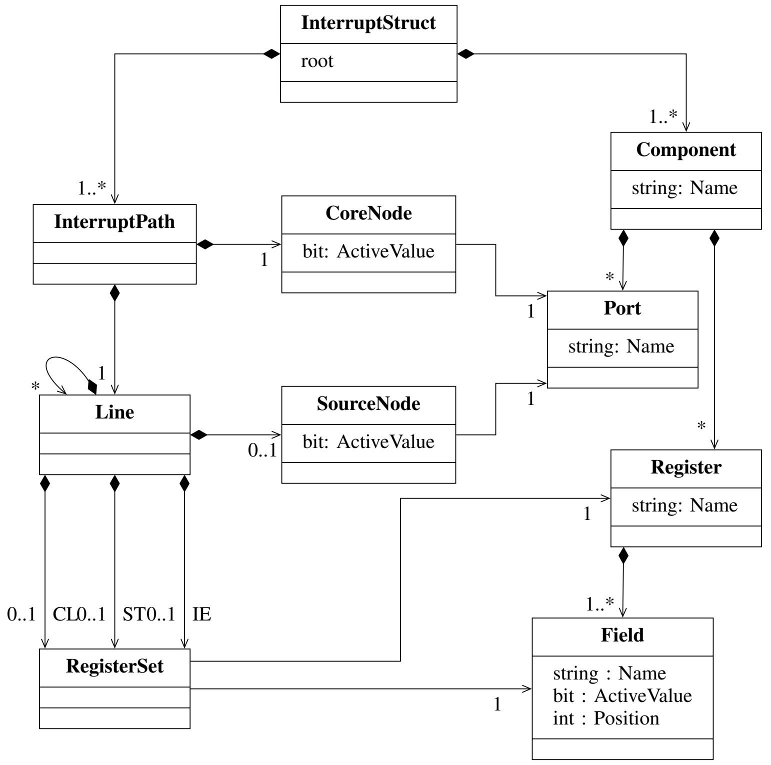

Fig. 2: UML diagram of the meta model used to describe possible variations of interrupt structures

For further processing, the structural information is stored in Extensible Markup Language Schema Definition (XSD) format. With this format verification engineers can easily load the meta structure and are able to create their own interrupt structure model of the design. In our case we used a generation framework called MetaGen [5] to load the UML model directly. After loading the meta structure we started with adding components like processor and peripheral. These components were extended by register and register fields of the target design. In the next step, we created interrupt paths by adding lines and expanding these with further lines, nodes and different registers as for example:

- Interrupt Enable register (IE): A register that activates or deactivates a particular branch of the interrupt structure.
- Status register (S): A register that can be read to determine the exact source of an arriving interrupt signal.
- Clear register (C): A register that is used to reset status registers.

Each line is able to contain one of each register type, i. e. three elements in total. In case of more registers of the same type, a new line can easily be added. With this aforementioned procedure, we are able to model a complete interrupt structure of a design, containing several trees of interrupt structures. An interrupt structure tree starts with an core node at processor side and ends with one or more source nodes at peripheral side. Finally the structural information is stored in XML format.

From this file, we generate design-specific testbench code like *uvm_environment* and test sequences by using parser scripts. Together with testbench files, that come with our interrupt verification package, the *uvm_environment* and test sequences can be used to verify the interrupt structure of the DUV. The package itself includes an interface module, a *uvm_monitor*, a base sequence with different test scenarios and macros for simpler monitor instantiation and configuration. The aforementioned parser scripts were written in Python and use an existing template engine called mako [7]. With that engine it is possible to run through the XML file and filter out information that are needed for each file. For the first generated file, all names of source and core nodes of the interrupt structure model are collected and combined in an new type *node_name*. This is done by searching all components for existing ports and collecting all created ports. The second file contains all test sequences; one test sequence for each interrupt path. Each interrupt path can be verified standalone due to two reasons:

1) Every interrupt enable register can be replaced by an *AND* gate and
2) all parallel structures within the interrupt tree are combined with a *OR* gate.

In Figure 3 the functional behavior of the interrupt structure model of Figure 1 is shown. The boolean equation of this structure can be seen in Equation (1).

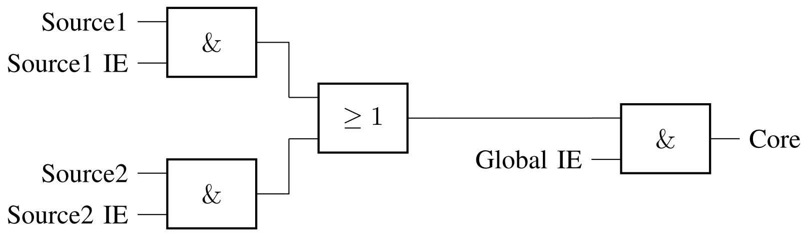

Fig. 3: Extract of an interrupt structure with logic gates

$$Core = ((S1 \wedge S1IE) \vee (S2 \wedge S2IE)) \wedge GIE \tag{1}$$

Now we can transform equation 1 in equation 2 by applying simplification rules.

$$Core = (S1 \wedge S1IE \wedge GIE) \vee (S2 \wedge S2IE \wedge GIE) \tag{2}$$

The interrupt structure in Figure 3 consists of two interrupt paths. Each bracket in equation 2 represents one of these paths. Both are combined by an logical *OR* gate so that the complete interrupt structure can be verified by verifying the interrupt paths standalone.

Therefore a special algorithm was developed, which searches for every possible path inside the interrupt structure stored in the XML file. That means, for every interrupt structure tree with one core node and multiple source nodes several test sequences are generated. For that, the algorithm starts at the core node and is looking for the next line element. In the case of multiple line elements the elements are chosen sequentially until no further line exists. This situation is equivalent to having found a complete interrupt path. Therefore, a test sequence is generated and the algorithm starts again for the next test sequence. These sequences contain information about registers, register fields and ports within the interrupt path. For the last file, the parser uses the information about all ports to generate the *uvm_environment* with one unique *uvm_monitor* instantiation for each node. This can be done because all core and source node elements refer to one port element. Listing 1 and Listing 4 show excerpts of the generated environment and test sequences.

B. Verification Environment, Monitoring and Evaluation

The generated *uvm_environment* can be seen as shell for all needed verification components (see Figure 4). Inside this shell the *uvm_scoreboard* and one *uvm_monitor* for each core and source node of the interrupt structure are instantiated. The number of monitors inside the environment is identical to the number of elements inside the generated type *node_name*.

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ...
    monitor_SOURCE1 = signal_monitor::type_id::create("monitor_SOURCE1", this);
    uvm_config_db#(string)::set(uvm_root::get(), "*monitor_SOURCE1", "signal_name", "
        SOURCE1");
    uvm_config_db#(int)::set(uvm_root::get(),  "*monitor_SOURCE1", "sensitive_level"
        ,1);
    ...
        set_config_int("*","recording_detail",UVM_FULL);
endfunction : build_phase

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    ...
    monitor_SOURCE1.item_collected_export.connect(interrupt_sb.item_collected_import);
    ...
endfunction : connect_phase
```

Listing 1: SystemVerilog implementation of the interrupt environment

During the *build_phase* in the *uvm_environment*, each component is created and configured with the information stored in the XML file. Of course, each component can also be extended using type-overrides and adapted to specific needs. With the *connect_phase*, the Transaction Level Model (TLM) port of each *uvm_monitor* is connected to the TLM port of the scoreboard. That enables monitors to send their transaction items to the scoreboard, which is necessary for data exchange.

Monitors itself have two parameters, namely *signal_name* and *sensitive_level*, which are both provided by the model of the concrete interrupt structure. These are automatically configured inside the *build_phase* based on the information provided by the XML. Instead of connecting monitors to physical DUV signals directly, they're use interface modules to observe internal design signals via interfaces (Listing 2).

Inside the DUV, interface modules are instantiated by using bind statements and are connected to the testbench with virtual interfaces [8]. Using System Verilog (SV)'s bind statement [9] [10] yields the advantage that no vendor dependent commands are needed. Of course, design internal signal names must be known for this approach. Figure 4 gives an overview of this concept.

```
module signal_connect#(string NAME="") (in logic Data);
  signal_if virtual_signal_if();

  initial
  begin
    uvm_config_db#(virtual signal_if)::set(uvm_root::get(),{"*.monitor_",NAME}, "
        vif", virtual_signal_if);
  end

  always@(Data)
  begin
    virtual_signal_if.probe_value = Data;
  end
endmodule : signal_connect
```

Listing 2: SystemVerilog implementation of the interface module

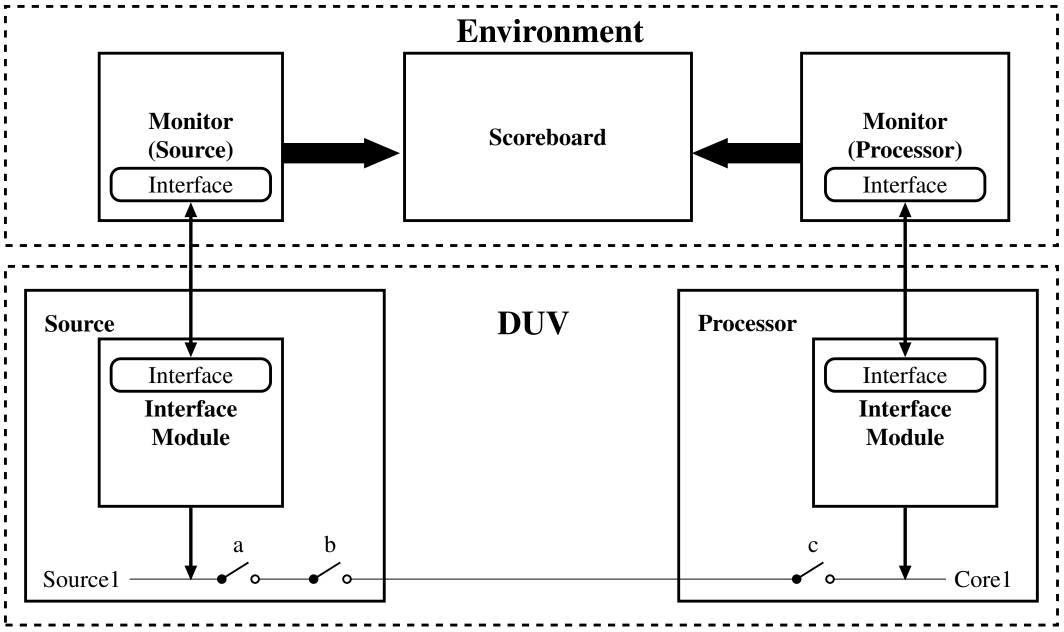

Fig. 4: Example for a UVM testbench that illustrates how interrupt signals are monitored

Inside the monitors, the XML-based parameter *sensitive_level* is used to select the triggered edge of the probed signal. After detecting the needed edge type, a transaction item containing the signal name is created and sent to the scoreboard for evaluation (Listing 3).

```
// run phase
virtual task run_phase(uvm_phase phase);
   fork
      if(sensitive_level)
         monitoring_high();
      else
         monitoring_low();
   join
endtask : run_phase

virtual protected task monitoring_high();
   forever begin
   @(posedge vif.probe_value);
      item_collected = new();
      void'(this.begin_tr(item_collected));
      item_collected.signal_name = signal_name;
      item_collected_export.write(item_collected);
      this.end_tr(item_collected);
   end
endtask : monitoring_high
...
```

Listing 3: SystemVerilog implementation of the interrupt monitor

The evaluation inside the scoreboard is split into two tasks. One for evaluating detected interrupt signals signalized by incoming transaction items and another for interrupt signals predicted by different tests. The information of incoming transactions items are used to generate global events in the verification testbench. These events have unique identifiers; a combination of the respective node's name and a postfix. Tasks inside tests and inside the scoreboard are waiting for such events to go on with further processing. For every detected interrupt, a data structure containing predicted interrupts is searched and in case of a mismatch a *UVM_ERROR* is created.

During the test, the generated sequences automatically predict different upcoming interrupts depending on the tested interrupt path. The interrupt names are added to the aforementioned field of predicted interrupts for evaluation of detected interrupts. After that, two tasks are created and running in parallel. First one is waiting for an upcoming interrupt event with the respective event identifier. Second one is waiting for a defined timeout, which is set from the test sequence. In case that the respective interrupt is detected in time, an *UVM_INFO* is shown, a covergroup is sampled and a semaphore is put. Otherwise if the timeout process is over before an interrupt is detected, a *UVM_ERROR* is generated and the semaphore is put. If one of both tasks is finished, a further task gets the semaphore and kills all other tasks. Finally the interrupt is deleted from the list of predicted interrupts.

C. Test Scenarios, Sequences and Covergroups

To test the functionality of each interrupt path lots of test sequences are generated by parser scripts; one for each path. All sequences are derived by a base class called *interrupt_base_seq*, that comes with the interrupt verification package. Inside this class, all test parameters are stored and test scenarios are implemented. Derived sequences can overwrite these parameters with special methods implemented in the base sequence. With these functions all needed information like source and core node names, register field names and activity levels can be set. Tasks to trigger and release an interrupt node depend on the module functionality and must therefore be implemented by the verification engineer. In a top level verification environment, this can be done by reusing already existing sequences that cause the peripheral hardware to generate an interrupt. Alternatively, an automatically generated sequence that is based on force and release can be used.

```
class seq_source1_core1_intr extends interrupt_seq;
   `uvm_sequence_utils(seq_source1_core1_intr, interrupt_virtual_sequencer)

   function new(string name = "seq_source1_core1_intr");
      super.new(name);
      path.set_name("seq_source1_core1_intr");
```

```
        path.set_source_node_name("source1");
        path.set_core_node_name("core1");
        path.ie_registers.add("interrupt_ctrl","global_en",1);
        path.ie_registers.add("module1_ctrl","module1_en",1);
        path.status_registers.add("module1_sts","source1_is",1);
        path.clear_registers.add("module1_clr","source1_ic",1);
    endfunction

    virtual protected task trigger_node();
    ...
    endtask

    virtual protected task release_node();
    ...
    endtask
endclass
```

Listing 4: SystemVerilog implementation of the interrupt sequence library

The different test scenarios are implemented inside the base sequence class. There are four different test scenarios which cover the most common use cases that occur in real applications. The most important scenarios are *pending* and *non-pending* interrupts: They only differ in the time of setting the interrupt enable register. For pending scenarios the interrupt enable registers are set after the interrupt event occurred. Whereas for non-pending scenarios the occurred interrupt is propagated directly to the core because all interrupt enable registers are already set. In addition, there is a scenario that checks the enable and disable functionality of each interrupt enable register. The last scenario triggers no interrupts and checks that no monitor detects an interrupt.

Before entering the body task of the base sequence the scenarios and open interrupt enable registers are randomized to obtain different combinations of scenario and parameter. Depending on the scenario generated during the pre-randomization, the corresponding task is chosen. For the verification of status and clear register inside the interrupt path a *uvm_register_model* is used to predict and read back the values of the register. This is done to make sure that all status and clear registers inside the DUV work properly.

In order to test the pending interrupts, all interrupt enable registers are being disabled and all status registers of the interrupt path are written to the *uvm_register_model*. After that the interrupt of the source node is predicted with the previously described scoreboard function and the interrupt is eventually triggered. The sequence is now waiting for an event generated by the scoreboard or for an predefined timeout. After that the interrupt is released and only be stored inside the status register. In the next step the interrupt is predicted in the destination module, which is in most cases the processor, and all interrupt enable registers are set. From a functional point of view this can be seen as closed switches. Finally, the task is waiting for the interrupt event or a timeout. After an individual clearing delay the clearing registers are set and the all registers inside the *uvm_register_model* are read back.

Non-pending interrupts are tested in a similar way. There is only the difference that all interrupt enable registers are set in the very beginning and not during the scenario. Everything else is handled in the same way.

The scenario to test the disable functionality of interrupt enable registers is quite similar to the scenario for pending interrupts. But there are some important differences. After releasing the interrupt in the source, not all interrupt enable registers are set. This is accomplished by evaluating a variable that is randomized at the beginning of the test scenario and has a value between zero and the number of interrupt enable registers within the tested interrupt path. With the help of this mechanism it is ensured that different interrupt enable registers within the path to be verified are not set. At the end of this scenario a delay is applied to check that no interrupt occurred. In case there is an interrupt enable register with a constant high value we are able to detect this by receiving an interrupt on core side which was, however, not predicted.

With the last scenario it is checked that no status register has a wrong default value. Therefore, all interrupt enable registers are set at the beginning. If an interrupt is now detected by a monitor, an *UVM_ERROR* is thrown since such an interrupt was not predicted.

For each interrupt path an own covergroup is created with one bin for each test scenario. After randomizing the scenario, this covergroup is sampled and the corresponging bins are filled. The minimum number of hits can be set by the verification engineer by modifying the covergroup options [11]. Also the open interrupt enable registers of one path are evaluated if the third scenario is selected. This is important to guarantee that each interrupt enable register was tested individually. Such a test

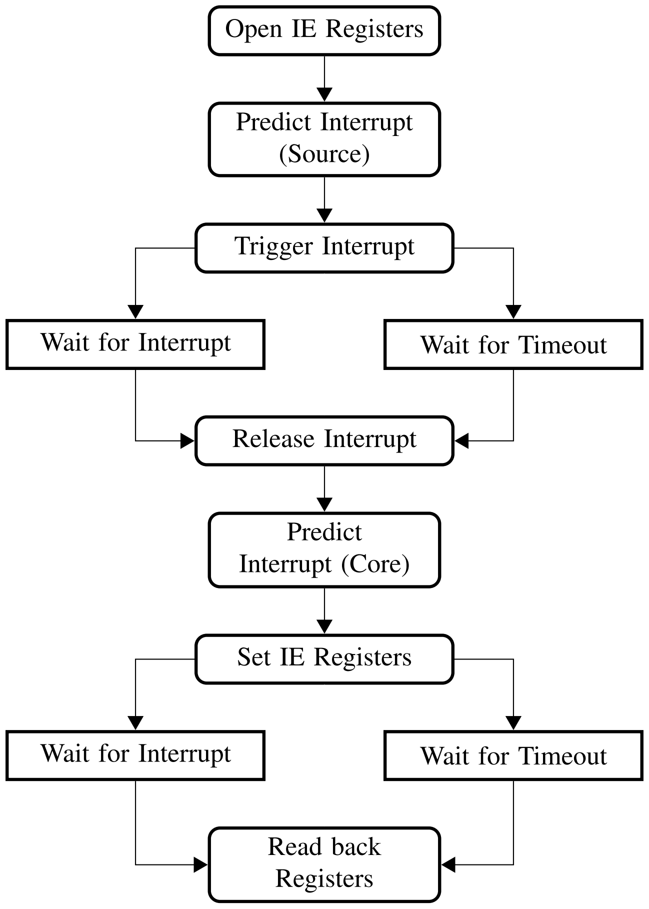

Fig. 5: Decision diagram of pending interrupt tests

only makes sense if there was a pending or non-pending interrupt scenario, which means that all interrupt enable registers were set and an interrupt occurred on core side.

```
covergroup path_cg ();
  option.name = {path.get_name(),"_cg"};
  option.per_instance = 1;

  test_case : coverpoint intr_case {
    option.at_least = 1;}

  open_ie : coverpoint open_ie iff(intr_case == IE_NOT_CLOSED){
    option.at_least = 1;}
endgroup : path_cg
```

Listing 5: Generated covergroup for test scenarios

IV. Application and Results

The previously described approach for verifying interrupt structures was applied to one of our SOC projects. This micro-controller-based chip for automotive applications focuses on driving electric motors e. g. for window lifts or cooling fans. The design includes an ARM^{TM} processor [12] and several peripheral modules (each with an own interrupt structure) and is ideally suited for applying our method. The overall structure of the SOC can be seen in Figure 6. We decided to use our method for interrupt structure verification on module level since the top-level was not available at this time. For testing our new approach we started with the High Side Switch (HS) module. This module is optimized for driving resistive loads like single or multiple

Light Emitting Diodes (LEDs) or other loads that require a high side switch. Compared to other peripheral modules, the High Side Switch has a complete interrupt structure including status, clear and interrupt enable registers.

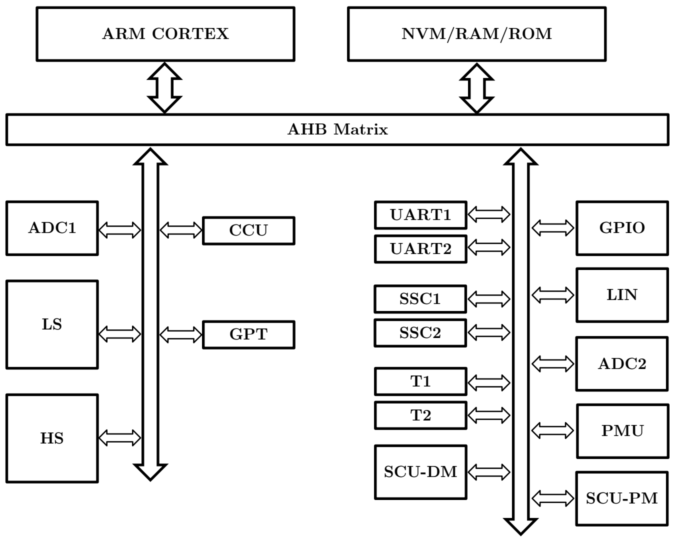

Fig. 6: Block diagram of the SoC used for our interrupt verification.

First of all we created a model description of the corresponding interrupt structure to generate the *uvm_environment* and test sequences.

For this purpose, we used our meta model for interrupt structures and created a component with the name of the module. After that we added ports for inputs (Over Temperature (OT), Over Current (OC) and Open Load (OL)) as well as for outputs (INT_HS) and defined their active values. Furthermore, we added status, clear and interrupt enable register for the module. One enable register to enable all interrupts for the complete module and one enable register for each interrupt input. For all registers we also defined the required register bit fields. After that we modeled the interrupt structure by connecting lines, adding source and core nodes as well as referencing ports and register fields. Finally we executed our parser script to generate the aforementioned files.

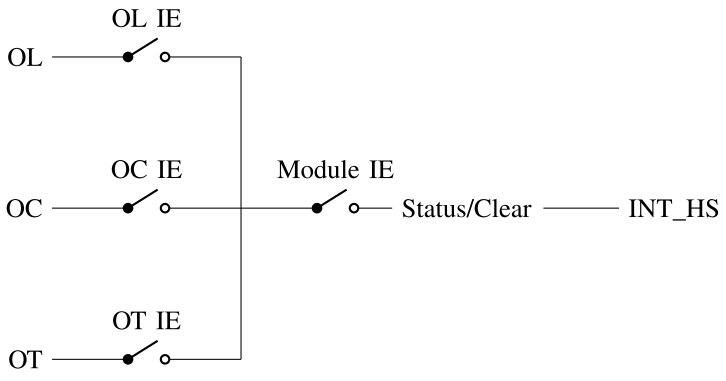

Fig. 7: Interrupt structure of the High Side Switch module

In the next step we added the virtual tasks inside the test sequences to trigger interrupts for over current, over temperature and open load. After that we implemented the tasks for register model access to use the full functionality of our register tests. Running our test sequences with different test scenarios eventually revealed a bug in the design. One of the status registers was wrongly connected and lead to an mismatch during read back of the registers.

V. Conclusion and Outlook

In this paper we presented a new approach for verifying interrupt structures in processor based designs by leveraging the capabilities of System Verilog and UVM. For this purpose we created a generic meta model that covers all interrupt structures and their most common elements. With the help of this structure description we are able to automatically generate a UVM testbench and appropriate test sequences. Since no initial testbench setup is required we eventually could achieve considerable time savings and reduce project costs. Our concept moreover establishes a methodology for interrupt verification within our projects resulting in better reusability and maintainability.

For future work we plan to extend our meta model in such a way that it offers better support for edge selection mechanisms within the interrupt paths. In addition, a major task will be to enable hierarchical reuse of already existing module interrupt models for top level verification. Furthermore it is planned to introduce a concept for tagging generated code and user-defined code in the testbench. This should lead to a faster integration into existing environments and therefore to faster verification results.

Our final goal is to provide an easy-to-use verification method that is able to verify interrupt structures of any design in a short period of time.

References

[1] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.1 User's Guide," 2011.

[2] F.-C. Yang, W.-K. Huang, J.-K. Zhong, and I.-J. Huang, "Automatic Verification of External Interrupt Behaviors for Microprocessor Design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 9, pp. 1670–1683, September 2008.

[3] F.-C. Yang, J.-K. Zhong, and I.-J. Huang, "Verifying external interrupts of embedded microprocessor in SoC with on-chip bus," in *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, November 2008, pp. 372–377.

[4] T. Prasad and S. Damani, "Plug-n-play UVM Environment for Verification of Interrupts in an IP," in *Design & Reuse IP-SoC Conference*, 2013.

[5] W. Ecker, M. Velten, L. Zafari, and A. Goyal, "Complementing EDA with Meta-Modeling and Code Generation," in *Design and Verification Conference*, March 2014.

[6] J. Bergeron, *Writing Testbenches using SystemVerilog*, rev. ed. New York: Springer Science+Business Media, 2006.

[7] M. Bayer, "Mako Templates for Python," http://www.makotemplates.org/, Accessed: 2016-01-01.

[8] D. Rich, "The Missing Link: The Testbench to DUT Connection," in *Design and Verification Conference*, 2012.

[9] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, 2013.

[10] D. Smith, "Using bind for Class-based Testbench Reuse with Mixed-Language Designs," in *Synopsys Users Group*, 2009.

[11] Graphics, Mentor Verification Methodology Team, "Coverage Cookbook: Online Methodology Documentation from the Mentor Graphics Verification Methodology Team," 2014.

[12] "ARM Cortex Processors," http://www.arm.com/products/processors/index.php, Accessed: 2016-01-03.