Coverage Driven Verification of IEEE P1500-compliant Embedded Core Test Infrastructures

Thanasis Oikonomou
poisson@globetechsolutions.com

Iraklis Diamantidis
iraklis@globetechsolutions.com

Stylianos Diamantidis
stelix@globetechsolutions.com

Abstract

Core-based design and reuse have been the key elements of efficient System-On-Chip (SoC) development. Testing of the embedded cores, however, introduces important challenges, such as core test reuse and interoperability at the SoC level, as well as the need for defining a common test infrastructure among cores from different suppliers. The IEEE 1500 Proposal for a Standard for Embedded Core Testing addresses these issues by proposing a flexible hardware test wrapper architecture for embedded cores together with a Core Test Language. In this paper we justify the need to thoroughly verify the functionality of the complete testing hardware infrastructure in P1500-compliant SoCs. We present a coverage-driven verification approach based on an eVC architecture, which can be part of the overall SoC level validation strategy, being equally flexible and extensible to the IEEE P1500's proposed hardware infrastructure.

1. Introduction

Design reuse has been the key element to designing increasingly complex Systems-On-Chip. The model of PCB development in which pre-manufactured ICs are reused has been transferred to the chip level. The designers create libraries of predesigned and preverified building blocks, or *embedded cores*, which make it easier to reuse complete functionality to new Systems-On-Chip.

However, while an IC provider delivers manufactured and tested components to be used in a PCB, embedded core providers deliver components in a range of hardware description levels. It is the SoC integrator's responsibility to put together all the embedded cores and test the manufactured SoC. With the embedded cores typically coming from different suppliers it has become a necessity to provide a standard test infrastructure to address the issues of test reuse and interoperability at the SoC level.

The IEEE 1500 Proposal for Standard for Embedded Core Testing proposes a flexible test infrastructure comprising of a hardware wrapper architecture and a Core Test Language [1-3]. With the imminent release of IEEE 1500 standard, industry support is growing significantly. EDA tools capable of generating IEEE P1500 core wrappers have emerged in addition to the ATE/Tester extensions to support the standard. Both sides exploit the CTL modeling information of the embedded cores. Also, several significant IP providers have announced IEEE P1500 compliance in both existing and future design blocks.

Related publications have presented work done on SoCs built with IEEE P1500 testability features [4, 5]. Extensions to the mandatory IEEE P1500 register set in order to support already working testability hardware, e.g. BIST is covered in [6]. IEEE P1500 has borrowed many features from its IC counterpart standard, IEEE 1149.1 JTAG [7]. The authors of [8] describe an approach of verifying JTAG logic using a combination of simulation of black-box checks and tracing.

We recognize the need for taking a comprehensive approach to thoroughly verify the functionality of IEEE P1500 wrappers and wrapped cores in a SoC environment. The need originates from the fact that an IEEE P1500 wrapper is itself a hardware design, which can be created in-house or sourced externally, designed by engineer or generated by tool. In any case a P1500-compliant design is subject to a range of possible errors so complete and methodical verification of the IEEE P1500 test logic is needed.

It is of great importance to understand the challenges arising from the IEEE P1500 standard and create a verification environment that faces them efficiently. First of all, P1500-compliant *core providers* need to ensure that their deliverable, most likely *soft IP*, complies to the standard and is functionally sound. On the other hand, *SoC integrators* dealing with P1500-compliant embedded cores need to validate wrapper functionality both at the *standalone* and *SoC levels*. Moreover, correct wrapper functionality needs to be ensured upon synthesis of the design, now *at the gate level*.

The nature of the IEEE P1500 standard itself poses two important challenges. Firstly, IEEE P1500 allows for a plethora of cell behaviors and implementations resulting in great *flexibility*. In addition, the standard leaves room for the users to define their own wrapper instructions and registers *extending* the standard's mandatory set.

All the above can be summarized in a list of features that the verification environment must offer to address all the challenges posed:

– ***Core and Coreless Operation*** – The ability to verify a wrapper with and without its core.

- ***Single and Multi-wrapper Operation*** – The ability to verify a standalone wrapper and an IEEE P1500 daisy chain of wrappers.
- ***Layered Monitoring*** – Observing behavior in environments ranging from white-box to black-box.
- ***Flexibility*** – Ensuring that all configuration options within the standard can be satisfied.
- ***Extensibility*** – Providing as much support for user defined extensions as possible.
- ***Reusability*** – Being able to apply the environment across providers, projects and hardware description levels.
- ***Input Abstraction Layering*** – Specifying vector stimuli at different levels of abstraction.
- ***Functional Coverage Assessment*** – Measuring the extent of functional coverage that has been exercised in the system.

2. *e*VC Architecture

Due to the nature of the IEEE P1500 standard, the ***e***VC has been designed to be flexible and extensible, under the recommendations of ***e***RM™ (***e*** Reuse Methodology) [9]. While flexibility allows for modeling virtually any IEEE P1500 cell and wrapper configuration, extensibility enables adding user-defined registers, instructions, checks and coverage items and facilitates future work.

Figure 1 shows the modules of the ***e***VC by using a typical testing scenario: two daisy-chained IEEE P1500 wrapped cores, connected through the mandatory IEEE P1500 serial interface. The first IEEE P1500 wrapper contains the mandatory registers only, i.e. WIR, WBY and WBR. The second one contains two user-defined registers in addition to the mandatory set: a Wrapper Defined Register (WDR) and a Core Defined Register (CDR).

We associate an ***e***VC Agent with every wrapper in a IEEE P1500 serial daisy chain. The agent associated with the first wrapper in the chain is an ACTIVE one, while the rest of them are PASSIVE. As the figure shows, each agent has a Monitor (providing event identification, checking and coverage capabilities) and a Reference Model (which mirrors the behavior of the associated wrapper). The ACTIVE Agent also encapsulates a Sequence Driver and a BFM implementing constrained-random traffic generation and low-level signal driving respectively.

All agents are linked together forming an ***e*** list. This way, inter-agent communication can be achieved. For example, as we will see in Section 3.2, the BFM communicates with the Reference Model of each agent to learn the properties of each register.

3. Constrained-Driven Input Generation

Dynamic, constrained-random traffic generation in a IEEE P1500 daisy chain is carried out by the ACTIVE Agent. Data generation takes place in the Sequence Driver, while the task of actual signal driving is done by the BFM.

3.1. Sequence Driver

The Sequence Driver is a unit, providing a single point-of-control for dynamic, constrained-random generation of input data at three distinct levels of abstraction: P1500 event level, transaction level and test level.

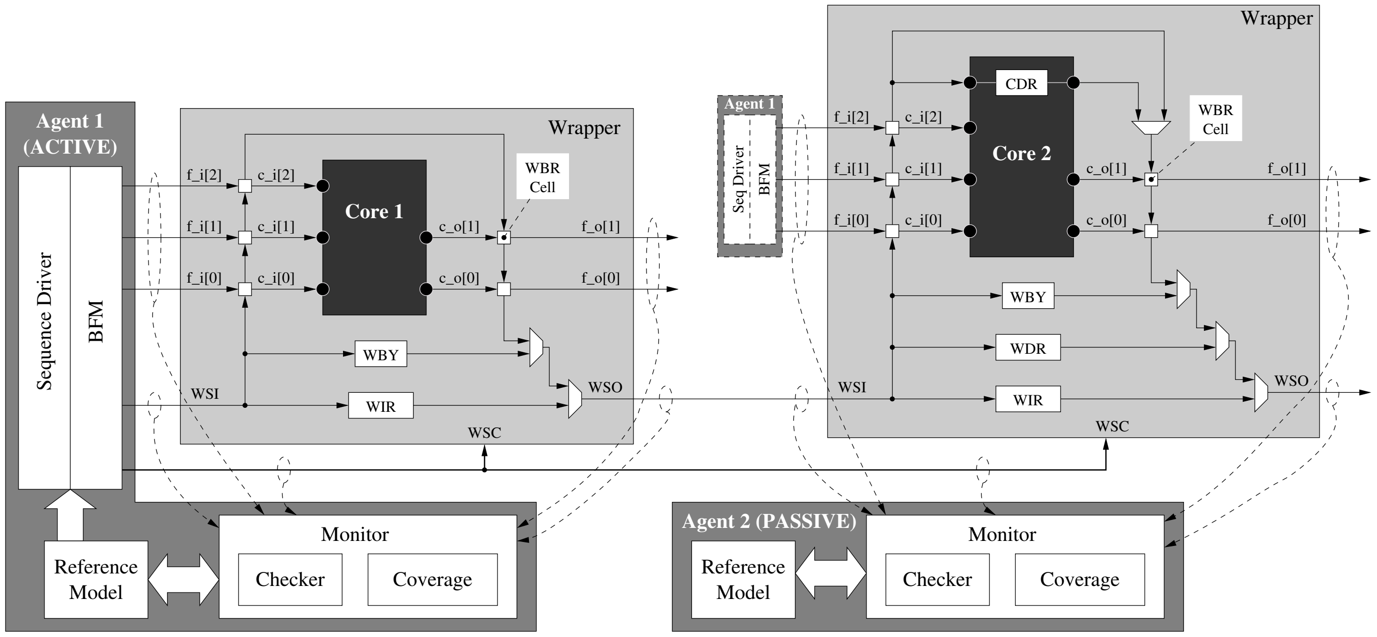

Figure 1. *e*VC architecture in a typical P1500 testing scenario

3.1.1. P1500 Event Level

P1500 events[1] control the operation of the wrapper registers. This is the lowest level of abstraction inside the Sequence Driver and includes the following P1500 events: SHIFT_WIR, CAPTURE_WIR, UPDATE_WIR, SHIFT_DR, CAPTURE_DR and UPDATE_DR. They are implemented as ***e***RM items, as shown in Figure 2.

```
type glbt_sect_action_t:
   [SHIFT_WIR, UPDATE_WIR, CAPTURE_WIR,
    SHIFT_DR, UPDATE_DR, CAPTURE_DR];

struct glbt_sect_item_s like any_sequence_item {
    -- Defines the P1500 event
    action: glbt_sect_action_t;
    -- Bit list used for shifts
    %serial_data: list of bit;
    -- List of list of bits used for captures
    %parallel_data: list of glbt_sect_bit_list_s;

    when SHIFT_DR glbt_sect_item_s {
        -- Generate up to 1024 shift bits unless
        -- otherwise constrained
        keep soft serial_data.size() > 0 and
                  serial_data.size <= 1024;
        -- No parallel data need to be generated
        keep parallel_data.size() == 0;
    };
};
```

Figure 2. P1500 events modeled as *e*RM items

P1500 events define both the way control signals (i.e. WSC) are manipulated and the actual data to feed the registers with. For a P1500 Shift event, a constrained-random list of bits to feed WSI is also generated (`serial_data`). Both the size and the contents of the list can be constrained by the verification engineer. As an example, shown in Figure 2, when SHIFT_DR is generated the size of the list is soft constrained to be up to 1024.

For a P1500 Capture event, the accompanying data generated consist of a list of bits for each wrapper (`parallel_data`). For a certain wrapper, the respective list of bits will feed the parallel inputs (if any) of the wrapper's active register. If it is a WIR-related P1500 event, the active register is WIR. For DR-related P1500 events, the active register is determined from the instruction currently loaded in the wrapper. All necessary active register information for a wrapper is provided by its associated Reference Model. The size of each generated list for a Capture cannot be changed by the verification engineer. It is always equal to the size of the active register, independently of the register's capability to capture. It is BFM's task to discover each register attributes and decide whether to apply the data or not. However, the contents of each list can be constrained at will.

[1] To avoid confusion with Specman and ***e*** events, we will always refer to P1500 events using its "P1500"prefix.

Finally, P1500 Update events have no actual data associated with them.

3.1.2. Transaction Level

In this level, series of P1500 events are combined together to form meaningful transactions. Higher-level transactions can also be formed by previously defined lower-level transactions, allowing for full flexibility and reusability of code.

This level of abstraction is implemented with ***e***RM sequences. We have defined a number of interesting sequences that are expected to be used in a P1500-compliant testing environment. The set of pre-defined sequences is implemented as an ***e***RM sequence library.

```
type glbt_sect_instr_name_t: [W_BYPASS, W_EX_TEST_S,
  W_CORE_TEST_WS, W_PRELOAD_S];

extend glbt_sect_sequence_s_kind: [LOAD_INSTRUCTIONS];

extend LOAD_INSTRUCTIONS glbt_sect_sequence_s {
  -- Constrainable field: List of instructions to load
  instructions: list of glbt_sect_instr_name_t;
      -- One instruction per agent
      keep instructions.size() == env_p.num_of_agents;

  -- Actual bit stream to shift in
  !instr_stream: list of bit;

  body() @driver.clock is only {
    -- For each wrapper in the chain..
    for w from (instructions.size() - 1) down to 0 do {
      var instr_opcode: list of bit;
      -- ..get the opcode for the instruction to load..
      instr_opcode =
      env_p.agents[w].instr_opcode(instructions[w]);
      -- ..and add it to the stream
      instr_stream.add(instr_opcode);
    };
    -- Shift the instruction bit stream into all WIRs
    do SHIFT_WIR my_seq_item keeping {
        .serial_data == instr_stream;
    };
    -- Finally, update all WIRs
    do UPDATE_WIR my_seq_item;
  };
};
```

Figure 3. Transaction LOAD_INSTRUCTIONS implemented with *e*RM sequence sub-typing

As an example, consider the typical transaction that loads instructions to wrappers in an IEEE P1500 daisy-chain. Figure 3 shows LOAD_INSTRUCTIONS ***e***RM sequence sub-type that implements the transaction. The sequence generates two P1500 events, a SHIFT_WIR followed by an UPDATE_WIR. The contents of the data to be shifted into the WIRs are computed to be equal to the opcode bits of the instructions we want to load. The final P1500 UPDATE_WIR event loads the shifted data to the update stage of each WIR. Note the extensibility of the sequence sub-type code: it can be used in any testing scenario employing anywhere from one to many wrappers in an IEEE P1500 daisy chain.

3.1.3. Test Level

This is the highest level of abstraction, in which transactions and/or P1500 events are combined to form meaningful test scenarios. It is also implemented using ***e***RM sequences.

We have prepared a set of interesting tests that are based on a strategically defined test plan for IEEE P1500 daisy chains, consisting of *N* wrappers, $N \geq 1$ [10]. This test suite makes use of the sequences defined in the sequence library of the lower level of abstraction.

Considering the two-wrapper configuration of Figure 1, a meaningful test scenario could be the following. Load the first wrapper with W_BYPASS and the second with W_EX_TEST_S. Then, apply a sequence of CAPTURE_DR, SHIFT_DR, UPDATE_DR P1500 events 1000 times, each with random Capture Data contents and random Shift Data length and contents. Figure 4 shows the code that implements this scenario.

```
extend MAIN glbt_sect_sequence_s {
    !instr_seq: LOAD_INSTRUCTIONS glbt_sect_sequence_s;

    body() @driver.clock is only {
        do instr_seq keeping {
            .instructions == {W_BYPASS; W_EX_TEST_S};
        };
        for i from 0 to 999 do {
            do my_seq_item keeping {
                .action in [SHIFT_DR, CAPTURE_DR, UPDATE_DR];
            };
        };
        stop_run();
    };
};
```

Figure 4. Test example of W_EX_TEST_S instruction

3.2. BFM

The BFM is a unit whose major responsibility is to do the actual driving of items (i.e. P1500 events) to specific DUT channels. Hence, it hides low level signal interfacing from the rest of the ***e***VC. In our ***e***VC, BFM operates in PULL_MODE, i.e. it explicitly requests a new item from the Sequence Driver as long as it finishes with the driving of the current one.

For each P1500 event, the BFM drives the IEEE P1500 WSC signals with the required values. For example, if the current item represents a CAPTURE_WIR, then it asserts two WSC signals (namely SelectWIR and CaptureWR) for one WRCK cycle.

Depending on the P1500 event pulled, the BFM may either drive WSI, register parallel inputs or nothing. For Shift events, the `serial_data` list of bits that accompanies them is fed to WSI bit-by-bit per WRCK cycle.

For Capture events, the item is accompanied by a list of bits for every wrapper. The BFM communicates with all Reference Models to find out the active register of each wrapper. Then, for each active register it finds out the properties of every cell[2]. A certain cell's parallel input signal is driven with the respective bit from the `parallel_data` lists accompanying the CAPTURE_DR event unless:

- The cell cannot capture while its wrapper is loaded with a specific instruction (e.g. WBR input cells at W_CORE_TEST_WS).
- The cell's parallel input signal is characterized as "monitor only", being driven by another HDL module (e.g. WBR output cells driven by the core).
- The cell's parallel input signal is not accessible to the ***e***VC (e.g. a WDR cell in a black-box wrapper implementation).
- The cell does not have a parallel input signal.

What is of great importance to notice is the way the ***e***VC driving modules (i.e. sequence driver and BFM) fulfill the requirements posed in Section 1. Specifically, they support virtually any IEEE P1500 wrapper testing scenario with combinations of the following alternatives:

- coreless IEEE P1500 wrappers or P1500 wrapped cores
- black-box or white-box implementations
- soft, firm or hard hardware description levels
- any test scenario ranging from standalone wrapper to SoC level

4. Functional Coverage

Functional coverage collection is a sub-task of the Monitor. Each Monitor identifies individual P1500 events by watching the signals of its associated wrapper. Events produced by the Monitor are used by the Functional Coverage sub-module in order to fulfill its goals.

It is imperative that we strategically select the functional coverage metrics in order to provide us with as much information as possible on the functionality exercised by running certain tests. We have defined a representative set of functional coverage metrics that can be used for measuring verification progress of black-box IEEE P1500 wrappers, i.e. wrappers for which we have no information on the way their cells and control logic have been designed and no observability of wrapper internal signals. It is obvious that the same metrics can be applied to white-box or partially white-box implementations. Of course, access to white-box wrapper internal structures can lead us to the definition of more coverage metrics giving a better insight of the

[2] Cells inside a register may have different properties, resulting in different behavior. For example WBR input and output cells differ in their behavior upon CAPTURE_DR, when W_EX_TEST_S or W_CORE_TEST_WS is loaded.

functionality that has been exercised. The extensibility feature of the ***e***VC allows us to define new coverage metrics a posteriori with little effort. Also, the metrics presented here apply to both coreless IEEE P1500 wrappers and to IEEE P1500 wrapped cores. Finally, the set is suitable to measure coverage in a multi-wrapper scenario, in which all wrappers are connected through their mandatory serial TAM in a daisy-chain way. Results will be represented on a per wrapper basis for metrics that may vary among wrappers.

In what follows we list some of the metrics we find interesting to demonstrate.

Instructions loaded **[per wrapper]**. Each reference model is capable of discovering which instruction is loaded in its wrapper upon each UPDATE_WIR P1500 event. In a multi-wrapper scenario, the instructions loaded vary among the wrappers so this information will be gathered on a per wrapper basis. This metric will help us discover if there are untested instructions in a wrapper.

P1500 events applied. P1500 events are caught by each Monitor. Coverage information is unique for all wrappers in an IEEE P1500 daisy chain because they share the same control lines of the mandatory serial TAM. This metric helps us discover if we have applied all possible P1500 events to the wrappers.

Instructions × ***P1500 events*** **[per wrapper]**. Using coverage collector's ability to cross two or more metrics, we implemented the cross coverage metric of instructions and P1500 events per wrapper. Notice that in a multi-wrapper scenario this cross metric has to be presented on a per wrapper basis since the ***instructions*** metric varies for each wrapper. This metric will help us discover if all P1500 events have been tested for each instruction loaded on every wrapper.

Number of successive shifts **[per register] [per wrapper]**. This metric shows the number of successive shifts on each register of a wrapper. The result of this metric for each register is shown as a comparison to the register's size. For example, for a WIR of 4 bits, we are interested in seeing if there have been less than 4 consecutive shifts, more than 4 consecutive shifts and exactly 4 consecutive shifts. It is obvious that this metric is unique for each register of each wrapper.

For implementing coverage collection on a per wrapper basis we used the `per_instance` coverage item option on the agent name. Figure 5 shows its usage in implementing the ***number of successive shifts*** metric. The coverage metric is defined inside the register struct so that it is inherited in every register of the environment, including user-defined ones, resulting in great extensibility.

```
type glbt_sect_reg_kind_t: [WIR, WBY, WBR];

extend glbt_sect_ref_model_register_s {
    cover shift_done_cov_e is {
        -- The "per wrapper" implementation
        item agent_name using per_instance;
        -- "kind" is of type "glbt_sect_reg_kind_t"
        item kind using using no_collect;
        -- "shifts" are automatically counted
        item shifts using no_collect;
        -- Present results per register
        cross kind, shifts;
    };
};
```

Figure 5. Implementation example of "per wrapper" coverage collection.

5. Error Checking

Error checking is also a sub-task of the Monitor. Events emitted by the Monitor provide the timing at which error checking is performed. The checking is done on a per-wrapper basis and is based on the use of a Reference Model which is manipulated by the Monitor.

5.1. Reference Model

The Reference Model is a struct instantiated under every agent and is closely associated with its IEEE P1500 wrapper modeling its structure and behavior. It uses certain events emitted by the Monitor to update its internal state. Developed with both flexibility and extensibility in mind, the Reference Model can support virtually all P1500-compliant structures, whether P1500 or user defined:

– ***P1500- compliant registers*** - any number, any size.
– ***P1500-compliant cells*** - any implementation, any isolation behavior.
– ***P1500-compliant instructions*** - any number, any opcode, any behavior.

The critical entity in the reference model is the cell. The verification engineer is able to define each cell's configuration by simply constraining certain fields in the configuration file. Figure 6 shows how the Reference Model of the first agent of Figure 1 can be configured to model its respective WBR. The first extension configures the register size, while the second one sets the attributes of WBR cells. Notice that attributes can apply to all cells or differentiate between input or output cells.

Figure 7 depicts the way to extend the environment in order to define a new register: WDR at the second agent of Figure 1. It is easily done by adding its kind to the register kind type, defining it at the reference model struct and configuring it like WBR was in Figure 6.

5.2. Checking

The nature of the IEEE P1500 standard allows for many alternative internal IEEE P1500 wrapper implementations concerning cells, WIR decoder and other structures, so that exact internal signal names and timing can only be provided by the wrapper designer. In other words, checking assertions have to be defined by considering the IEEE P1500 wrapper as a black-box in general but also giving the verification engineer the ability to define their own assertions in the case internal implementation details are known. Moreover, to address the extensibility nature of IEEE P1500 standard, the ***e***VC allows for error checking to be inherited in user-defined registers minimizing effort and maximizing reuse.

```
extend AG_0 WBR glbt_sect_register_s {
    keep size == 5;
};

extend AG_0 WBR glbt_sect_register_cell_s {
    -- Constraints for all cells of WBR
    keep reset_value == 0;
    keep value_known == FALSE;
    keep has_update_stage == TRUE;

    -- WBR[4..2]: Input cells
    keep (cell_index >= 2) =>
        can_capture_on == {W_EX_TEST_S};
    keep (cell_index >= 2) =>
        can_update_on == {W_CORE_TEST_WS};
    keep (cell_index >= 2) =>
        p_in == append("f_i[", dec(cell_index-2), "]");
    keep (cell_index >= 2) =>
        p_in_monitor_only == FALSE;
    keep (cell_index >= 2) =>
        p_out == append("c_i[", dec(cell_index-2), "]");

    -- WBR[1..0]: Output cells
    keep (cell_index < 2) =>
        can_capture_on == {W_CORE_TEST_WS};
    keep (cell_index < 2) =>
        can_update_on == {W_EX_TEST_S};
    keep (cell_index < 2) =>
        p_in == append("c_o[", dec(cell_index),"]");
    keep (cell_index >= 2) =>
        p_in_monitor_only == TRUE;
    keep (cell_index < 2) =>
        p_out == append("f_o[", dec(cell_index),"]");
};
```

Figure 6. Reference Model configuration example

We propose a list of error checks that fulfill the requirements posed in Section 1.

WSO checking – The value of WSO signal is compared against the LS bit of the wrapper's active register to verify this register's integrity. A DUT error is reported if the values do not match. The active register and its LS bit value are provided by the Reference Model.

Wrapper normal mode checking – When wrapper is in normal (i.e. pass-through) mode (e.g. W_BYPASS is loaded), the values applied on parallel input signals of WBR cells (i.e. f_i[2:0] and c_o[1:0] in the example of Figure 1) are checked against values on their parallel output signals (i.e. c_i[2:0] and f_o[1:0], respectively). When the values do not match a DUT error is issued.

```
extend glbt_sect_reg_kind_t: [WDR];

extend AG_1 glbt_sect_ref_model_s {
    wdr: WDR glbt_sect_register_s;
};

extend AG_1 WDR glbt_sect_register_s {
    keep name == "Wrapper Defined Register";
    keep size == 4;
};

extend AG_1 WDR glbt_sect_register_cell_s {
    -- Constrain attributes appropriately here..
};
```

Figure 7. Definition of user-defined register (WDR)

Wrapper internal/external isolation mode checking – When wrapper is in internal (e.g. W_CORE_TEST_WS is loaded) or external (e.g. W_EX_TEST_S is loaded) isolation mode, values stored inside WBR cells are compared against values on their parallel output signals (i.e. c_i[2:0] and f_o[1:0), upon each P1500 event to verify isolation of the wrapper. If the values on the parallel output signals do not match the expected ones when the check is done, a DUT error is issued. Internal values of WBR cells are provided by the Reference Model.

Update checking – Provided that WBR has an update stage, we check that its parallel output signals (i.e. c_i[2:0] and f_o[1:0]) get updated with its cells internal values when UPDATE_DR is issued and the instruction loaded permits so. If the values on the parallel output signals do not match the expected ones when the check is done, a DUT error is issued. Internal values of WBR cells are provided by the Reference Model again. This functionality is inherent on all register types, including user-defined registers, sub-typed from the basic register type the ***e***VC provides. Should the verification engineer desire to modify or add to the inherited checking functionality upon UPDATE_DR events, they can do so by overriding the hook method provided.

Shift checking – If WBR does not have an update stage, we check that its parallel output signals (i.e. c_i[2:0] and f_o[1:0]) get the values of its cells internal values when SHIFT_DR is issued. If the values on the parallel output signals do not match the expected ones when the check is done, a DUT error is issued. Internal values of WBR cells are provided by the Reference Model of course. Similarly to the previous check functionality, the verification engineer may modify or add to the hook method provided to implement their own checks upon SHIFT_DR.

6. JTAG TAP Support at the SoC Level

The IEEE 1500 standard proposal defines the test infrastructure at the embedded core level. However, the proposal does not dictate the way to connect IEEE P1500 wrapped cores at the SoC level. A typical way of connecting wrapped cores to primary terminals at the SoC level is to use an IEEE Std. 1149.1 (JTAG) Test Access Port (TAP) and Controller. Such an example is shown in Figure 8. The JTAG TAP Controller simply translates the JTAG serial interface to the IEEE P1500 serial interface.

IEEE 1149.1 standard defines a State Diagram resulting in strict sequences of register accesses. Briefly, a data register access must start with a capture, followed by zero or more shifts, followed by an update. The same sequence holds for instruction register accesses. Hence, there are two possible P1500 event sequences only, coming out of the JTAG TAP Controller:

- CAPTURE_DR, *SHIFT_DR, UPDATE_DR
- CAPTURE_WIR, *SHIFT_WIR, UPDATE_WIR

Let us refer to the first sequence as JTAG_SCAN_DR and to the second one as JTAG_SCAN_WIR. Now, we consider JTAG_SCAN_DR and JTAG_SCAN_WIR as basic JTAG events. One can give any JTAG event to the JTAG TAP Controller by driving its TMS input appropriately, as defined in the JTAG State Diagram.

The ***e***VC supports the usage of a JTAG TAP Controller, by just configuring the generation and driving layers to generate and drive JTAG events as ***e***RM items. The change affects the two active units: Sequence Driver and BFM.

In the Sequence Driver level, the basic ***e***RM item type has been extended to include the two JTAG events. When a JTAG TAP controller is present, the item is constrained to be of JTAG event type only. Else, the item is constrained to be of P1500 event type only (the default case). Also, not all sequence sub-types defined in the sequence library are valid when JTAG TAP Controller is present. For example, a sequence sub-type that issues a CAPTURE_DR followed by a SHIFT_WIR is not valid in JTAG. Such sequence sub-types just type a warning message at JTAG's presence. On the other hand, any sequence sub-type that results in one or more repetitions of JTAG_SCAN_DR or JTAG_SCAN_WIR, is valid for JTAG. These sub-types result in generating JTAG events as ***e***RM items.

The BFM needs to implement a different interface protocol in the case of JTAG. Its main TCM executes a different thread in this case, implementing the JTAG State Diagram and driving the JTAG interface signals. To achieve this, the signal map of the ACTIVE agent also needs to be extended with the signal names of the JTAG interface.

Units and structs in both Sequence Driver and BFM level are notified on the presence of JTAG TAP Controller by a global environment variable, which is constrained by the verification engineer at the configuration file.

With the above mentioned approach, test level remains unchanged. The same tests can be run either with or without a JTAG TAP Controller. In the case of JTAG, if test uses a sequence that does not comply with the standard, that sequence will just type a warning message and the test will continue.

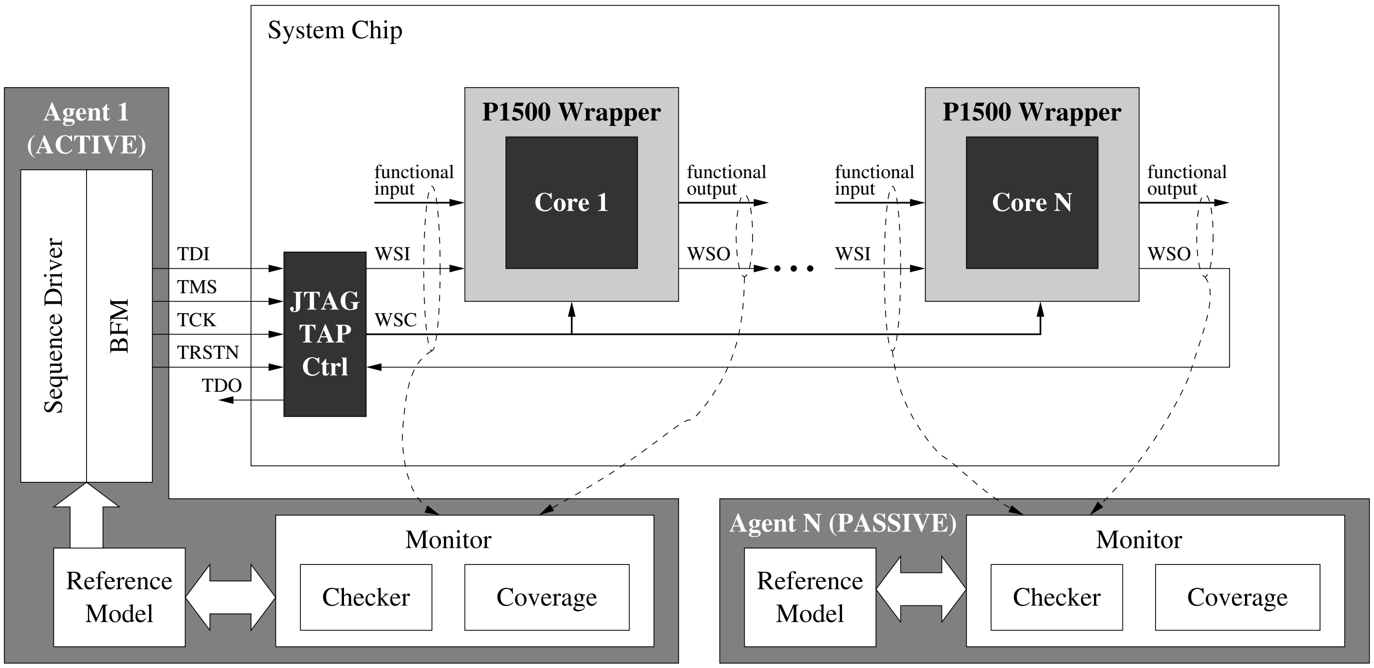

Figure 8. *e*VC operation at SoC level with JTAG TAP

Concerning the passive elements of the ***e***VC, these are still associated to IEEE P1500 wrappers by monitoring the IEEE P1500 serial interfaces, as Figure 8 depicts. This means that monitoring, checking and coverage collection is done on the IEEE P1500 serial interfaces, as if JTAG does not exist. One detail that the verification engineer can notice if JTAG is present concerns the ***P1500 event transition*** coverage item (derived with transitioning ***P1500 events*** coverage item, presented in Section 4). In this case, only transitions derived from JTAG_SCAN_DR and JTAG_SCAN_WIR can be seen. In the coverage results presented, every other transition will always be uncovered. The verification engineer is given the choice to either see these transitions or hide them from the coverage results as illegal.

7. Conclusions

In this paper, we presented a highly configurable, ***e***VC architecture for thoroughly verifying P1500-compliant test infrastructures. The ***e***VC addresses all challenges posed by the imminent IEEE 1500 standard having been designed with flexibility and extensibility in mind through all its levels and modules. Features include constrained-random vector generation, automated checking and coverage metrics definition, making the environment suitable for fully verifying infrastructures under virtually any configuration. Examples of ***e*** code have also been demonstrated in order to give a better insight of the ***e***VC operation and configuration parameters. Finally, we illustrated how the ***e***VC can support verification at the SoC level, for systems deploying the IEEE 1149.1 (JTAG) for chip terminal access.

References

[1] Y. Zorian and E. J. Marinissen, "Testing Embedded-Core-Based System Chips," *IEEE Computer*, vol. 32, pp. 52-60, 1999.

[2] E. J. Marinissen, R. Kapur, M. Lousberg, T. McLaurin, M. Ricchetti, and Y. Zorian, "On IEEE P1500's Standard for Embedded Core Test," *Journal of Electronic Testing*, vol. 18, pp. 365-383, 2002.

[3] F. DaSilva, Y. Zorian, L. Whetsel, K. Arabi, and R. Kapur, "Overview of the IEEE P1500 Standard," in the proceedings of *International Test Conference*, Charlotte Convention Center, Charlotte, NC, USA, Sep 30 - Oct 2, 2003, pp. 988-997.

[4] T. McLaurin and S. Ghosh, "ETM10 Incorporates Hardware Segment of IEEE P1500," *Design & Test of Computers, IEEE*, vol. 19, pp. 8-13, 2002.

[5] S. Picchiotino, M. Diaz-Nava, B. Foret, S. Engels, and R. Wilson, "Platform to Validate SoC Designs and Methodologies Targeting Nanometer CMOS Technologies," in the proceedings of *IP-SOC*, Espace Congres du World Trade Center, Grenoble, France, Dec 8-9, 2004, pp. 39-44.

[6] D. Appello, F. Corno, M. Giovinetto, M. Rebaudengo, and M. Sonza Reorda, "A P1500 compliant BIST-based approach to embedded RAM Diagnosis," in the proceedings of *10th Asian Test Symposium*, Kyoto, Japan, Nov. 19-21, 2001, pp. 97-102.

[7] IEEE Computer Society, "IEEE Standard Test Access Port and Boundary-Scan Architecture - IEEE Std. 1149.1-2001", New York: IEEE, 2001.

[8] K. Melocco, H. Arora, P. Setlak, G. Kunselman, and S. Mardhani, "A Comprehensive Approach to Assessing and Analyzing 1149.1 Test Logic," in the proceedings of *International Test Conference*, Charlotte Convention Center,Charlotee, NC, USA, Sep 30 - Oct 2, 2003, pp. 358-367.

[9] "***e*** Reuse Methodology (***e***RM) Developer Manual", Verisity Design, Inc, 2003.

[10] I. Diamantidis, T. Oikonomou, and S. Diamantidis. "Towards an IEEE P1500 Verification Infrastructure: A Comprehensive Approach", Globetech Solutions, 2005.