MULTIPLE INTERFACE CROSSCHECKING IN A SINGLE *e*VC ARCHITECTURE

by
Aggelos Ioannou, Thanasis Oikonomou, Stylianos Diamantidis
Globetech Solutions

ABSTRACT: *An eVC™ architecture is commonly used to verify a single interface of a device and to confirm that it complies with a particular set of protocol rules. However, architecting eVCs solely for single-interface verification can be limiting for certain types of applications and devices. Instead, a number of interesting benefits can be realized when deploying eVCs for verification across multiple interfaces, e.g. egress and ingress port of a design. By architecting an eVC for multiple interface cross-checking, one can harness a powerful verification environment that encapsulates not only the efforts of testing each interface separately, but also a better representation of data flow and overall functionality. This article offers solutions to the challenges that arise in multiple interface crosschecking and presents useful practices for this type of eVC-based verification architecture. Furthermore, the discussion is supported by an example of a commercially available eVC which implements cross-checking between two distinct interfaces.*

INTRODUCTION

A variety of *e* Verification Components exists today catering to a multitude of industry standard protocols and interfaces. As verification requirements grow more and more demanding, *e*VCs offer a great means of encapsulating verification experience, hence reducing effort and time spent while increasing quality and completeness. Such *e*VCs typically focus around single interfaces which provide a certain functionality or connectivity to a device or system under test (DUT). Modules that comprise the *e*VC environment are then used to inject input stimuli (drivers) or observe and check traffic (monitors). In this configuration, an *e*VC checks that all traffic adheres to the underlying protocol as well as that all data goes through checkpoints correctly. DUTs can this way effectively be verified against interface behavior. But how about the DUT itself?

Although using *e*VCs for interface (or bus) compliance can provide us with a lot of information about the proper workings of a device that encompasses such an interface, there is still a lot of valuable information that we could gather were we to take a step back and quantify interface-to-interface relationships across the device. There are many devices that are built on specifications that have more than one well defined interface or even interfaces which, although not exhaustively defined themselves, still need to adhere to some higher level protocol that can be quantified. For example, take a device with two interfaces[2], say a network device with an ingress and an egress port; such a device can have both its interfaces built on some well defined protocol specifications. For such devices there are many benefits if an *e*VC was to exercise the functionality on multiple sides. Implementing this type of operation in an *e*VC would not only give us the ability to check each side separately, but would also provide us with the power to incorporate checking of the device's functionality itself, drastically expanding our verification coverage.

This paper presents several benefits that result from deploying such an *e*VC architecture for device-level verification. In addition we discuss some of the challenges that arise in this type of architecture and suggest some solutions. The discussion is supported by brief examples taken from an already commercially available *e*VC that incorporates multiple interface support and cross-checking.

[1] *e* and *e*VC are trademarks of Verisity Inc.
[2] Hereon, the number two will be used for the ease of description when describing methodologies for crosschecking. Scaling to more than two interfaces can be understood intuitively.

BENEFITS OF MULTIPLE I/F CROSSCHECKING

A number of interesting benefits can be realized when incorporating multiple interface cross-checking in an *e*VC.

Interface-to-Interface Data Interdependencies

The first advantage lies in the fact that the user is allowed to capture interface-to-interface data interdependencies. Such interdependencies can be quantified in the *e*VC environment and create a basis for several comprehensive checks. As device protocol is being followed, the *e*VC, which has access to the interfaces, can check data flow even in such cases where the data are not just steered unmodified to the output. Furthermore, mechanisms can be deployed that check the data flow itself through the device to be tested.

Consider an example of a network interface subsystem transferring data between an embedded processor and a network link. The checking capabilities of the verification environment of a DUT like this can be greatly enhanced if the *e*VC can observe both the processor bus and the network link. One can design such an *e*VC to be able to observe arbitrary data flows through the device, whether generated by an *e* stimuli driver, or another component including a processor DUT.

Device-Level Data Flow Independent of Stimulus Driving

Being able to describe device-level data flow completely can allow relieving stimuli drivers from supporting data checking. In a verification environment, data checking would mainly be based on a scoreboard which would reside on the input drivers and output collectors (i.e. the so called stubs). However, in an *e*VC that can monitor device-level data flow, the user should be able to drive data in a variety of ways while simultaneously supporting all of the checking functionality. Multiple interface *e*VC architecture achieves this by collecting data flow information on the interface level while at the same time being independent of the drivers. The design-level data flow then becomes a feature that can easily be used when deploying such an *e*VC.

For some devices, such as data-steering DUTs which could also perform on-the-fly operations to data such as encapsulation or modulation, data-flow checking alone can be very powerful. Even errors that do not directly have to do with the data flow can cause data dissimilarities and hence can be revealed. Of course, there are other more focused checks that can provide the exact origin of the error for each specific case.

Better Protocol Checking

A multiple interface coverage approach can result in more complete protocol checks. In many cases protocol rules are not only related to separate interfaces, but also include other information such as timing, etc., that depend on the combined behavior of the device. With such rules specified, one can incorporate checks that have to do with this combined behavior. In addition, note that in most cases, combining these types of checks reveals the most important and complex functionality of a given device. Of course, single interface rules that have to do with verifying each of the interfaces separately will also be incorporated so that the offered environment can provide the complete range of error checking necessary.

Consider an example of a device that produces interrupts through one interface as a result of traffic on another interface. The entire interrupt activation protocol can be highly complex due to the fact that it represents a significant part of the device's functionality and thus deserves much attention. This type of testing is only possible in an *e*VC environment which provides access to both interfaces simultaneously.

Time, Effort and Quality Gains

Time savings and improvements in quality are additional advantages to designing *e*VCs based on multiple interface cross-checking as opposed to using a separate verification environment for each interface.

- By using a single *e*VC, the user will typically minimize the effort of having to set up multiple environments for each of the interfaces, including additional glue logic and other overhead.
- Being able to translate device-level transactions in the form of checks allows users to better translate the specification of the device into checks, hence creating more effective verification test cases.
- The incorporation of cross-checking results in a more complete verification environment by adding functionality that would otherwise be omitted and almost impossible to be later introduced and covered.
- Abstracting data flow checking out of the driving logic makes it easy to use the *e*VC's monitoring capabilities in different verification scenarios. For example, a data driver can be replaced by another DUT in an SoC environment which would provide data stimuli to the DUT covered by the *e*VC. This would still offer the same data-flow checking coverage, much the same as in the standalone DUT verification case.

CHALLENGES & SOLUTIONS

When designing an *e*VC for multiple interface coverage, a number of interesting challenges arise. In this section, we describe some of these challenges and present suggested solutions. We also illustrate examples based on the UART *e*VC; this is a commercially available *e*VC that incorporates two interfaces, specifically a generic processor side interface and a serial network side interface, and hence deploys many of the techniques offered in this discussion.

Multiple Interface Protocol Rule Checking

Checking multiple interfaces is the first challenge encountered when designing a multiple interface *e*VC. In order to accommodate the significant increase in complexity, checks should encompass more complicated functioning procedures which can include signals from more than one interface in order to be described and verified. Such checks can use signals from a certain interface, extract the control stimuli injected, and then check that another interface responds in the expected way. Consider an example in the context of the UART *e*VC:

```
expect {@net_packet_received;
        true (received_items >= trigger_level() and
          (trigger_interrupt == ENABLED));} =>
       {[..2]; @intr_assr;} @rclk_clk
else
       dut_error("Intr was not activated although Received ",
                 "Data are available and the corresponding ",
                 "interrupt is activated";
```

In this example, when a packet is received from the network interface and the FIFO trigger level is reached, the result should be the activation of a specific kind of interrupt to the processor. This check incorporates events that are associated with signals both from the network interface (packet received) and the processor interface (interrupt), as well as some internal protocol requirements.

Writing these types of checks can be very complicated and therefore, *e*VC designers should pay special attention to the checks needed for the protocol of the device being verified. Designers should make sure that the protocol rules translated into *e*-language checks are described exactly and exhaustively so that they do not collide with other protocol aspects. Otherwise, the result could be the signaling of DUT errors by the *e*VC that do not actually occur or, even worse, the omission of significant protocol errors.

Timing of Checks

Another challenge of multiple interface cross-checking relates to timing; one of the most important items to be verified. An *e*VC that incorporates multiple interfaces can work with *cross-timing* checks, that is, checks that refer to timing constraints among different interfaces. The rules that check these constraints can be very complicated as they are based on the protocol followed internally by the device, as well as the protocol of each interface.

The presence of multiple clocks is one main issue that can lead to added complexity. Each interface can use a different clock, and furthermore, these clocks may not be synchronized. It is also possible that their relative frequencies are not constant, changing between different simulation runs or even during the same run. The following is an example of a check that employs events from two clock domains. It originates from the UART *e*VC where an interrupt must be asserted by the DUT to the processor side immediately after the last datum is transmitted by the DUT to the network side (indicating empty space inside the DUT's FIFO).

```
expect {@last_THR_xmitted and
       true(THR_interrupt == ENABLED);} =>
       {[..8]*not @new_THR; @intr_assr;} @baud_clk
else
       dut_error("Interrupt was not asserted, although ",
                 "THR became empty and THRE interrupt ",
                 "was enabled");
```

Highlighted event, **`last_THR_xmitted`**, is emitted whenever the last datum leaves the DUT for the network side. This event is synchronized to `baud_clk`. However, the processor writes data to the DUT's `Tx FIFO` using a different clock, `xin_clk`. `baud_clk`'s period is a multiple of `xin_clk`'s period. The multiplier is configurable through a DUT register and can change at any time, thus configuring DUT's transmission speed. Highlighted event, **`new_THR`**, is synchronized to `xin_clk` and is emitted every time a new datum is written to DUT's Transmitter Holding Register (`THR` - actually DUT's FIFO) by the processor.

Another issue having to do with timing is that some aspects of the internal device protocol are important and should be included in the *e*VC's checks. However, their timing is a superset of a set of simpler timings. The *e*VC should in such cases rely on a set of *cross delays*. Such delays can be used to provide the needed checking capability in a hierarchical manner and allow the user more flexibility in setting their boundaries. To illustrate a cross delay we provide another check from the UART *e*VC example. As a hint, note that when a processor writes a specific register to the UART with a character, the device starts transmitting this character *after a while*.

```
expect @new_THR => {[..13*16]; @start_bit_xmitted} @baud_clk
else dut_error("THR was written by the processor but no ",
               "transmission followed");
```

Event **`new_THR`** has been introduced in the previous example. Event **`start_bit_xmitted`** is emitted whenever the first bit of a character is transmitted. As you can see, we have delay tolerance of 0 to 13x16 `baud_clk` cycles between the time `THR` is written and the actual transmission takes place to account for intra-device delays.

Implementation Dependencies

When designing a general application *e*VC, especially one that encapsulates device functionality across multiple interfaces, the designer must always make sure that the final solution is DUT implementation-independent. Such dependencies can arise when checking timing aspects of the internal device protocol where the timing is not tightly restrained. Techniques should then be applied that relieve any possible implementation differentiations.

One technique that can be deployed is the use of *time slacks*. The *e*VC can have some time periods defined within which an event is valid. In order to treat different DUT implementations properly, time slacks can be incorporated into the checks that concern complex protocol aspects. These definitions also need to be external and provide users the capability of fine-tuning their environment by constraining time slacks in a manner that best matches their DUT implementation. Below is an example that shows this technique, again from the UART *e*VC domain. To better understand the check, note that a UART DUT should issue an interrupt to the processor when there has been an error in the network reception line and this interrupt source is enabled. LSR (Line Status Register) is the register that holds reception errors and IER is the Interrupt Enable Register.

```
expect {@LSR_copy_ch;
        true( (LSR_copy[4:1] != 4'b0000) and
              (IER_copy[2:2] == 1'b1) and
              (stable_state == TRUE) ); [10]; } =>
        detach( {@intr_assr; ~[2..20]} ) @rclk_clk
else
       dut_error("LSR indicated an error, the corresponding ",
                 "interrupt is enabled, but no interrupt was ",
                 "issued");
```

Event LSR_copy_ch is emitted whenever the *e*VC internal copy of LSR changes. Checking whether there has been a reception error is done by checking LSR internal copy's bits [4:1]. However, there is no way to be certain that the value of LSR[4:1] inside the DUT is the same as the *e*VC's internal copy at every rclk_clk cycle because the cycles at which the DUT changes the LSR are implementation-dependent. However, there are cycles at which LSR[4:1] is stable and known. We refer to these cycles as ***stable states.*** A good example of a stable state is the time margin between a "few" cycles after reception of a network datum has finished and the cycle at which a new reception starts. The monitor inspects the network interface and decides on the stable state validity. The amount "few" can be set to a number large enough to account for every possible implementation.

Another technique that can be deployed is *queuing of events*. Some events should be expected in relation to other events. However, if the timing of such events is based on loose protocol aspects, the *e*VC monitor can queue them and compare them when a secure checkpoint is reached. This queuing can happen either for events received by the DUT or even for events that the monitor produces which will be used for comparison with those of the device.

In Figure 1 below, the idea of event queuing is graphically illustrated. The monitor expects some events to happen on interface A, while in parallel, it creates the expected events with its own logic for comparison. These events are produced according to the stimuli of protocol B and the related DUT protocol. However, instead of advancing in a lock-step fashion, the events are now kept in FIFOs. In this way, if a series of events are delayed due to implementation differences, the FIFO absorbs these delays leading to correct checking information for the user.

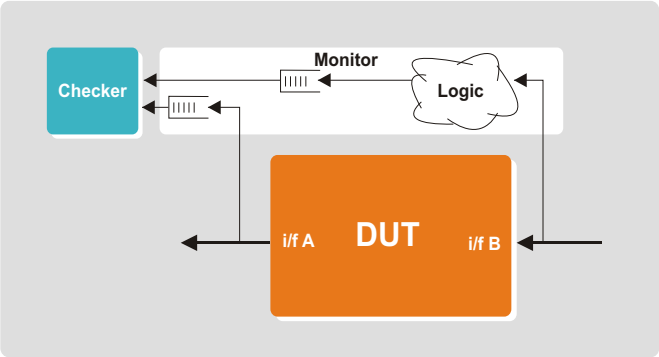


Figure1: Checkpointing events from separate interfaces inside the monitor

In the context of the UART *e*VC, this queuing of events can be implemented as follows:

```
on PROC_LSR_READ {
  if(stable_state == TRUE) {
    if (Monitor_Received_Data_Ready == TRUE) {
      if(Dut_Received_Data_Ready == TRUE)
        check that DUT_LSR[2:2] == MONITOR_LSR[2:2]
      else
        dut_error("The Overrun Error signaled ",
                  "in LSR is not correct");
    };
  };
};

if(DUT_LSR[0:0] == 1'b1 and MONITOR_LSR[0:0] == 1'b1){
    // if DUT has not received data_ready,
    // the MONITOR_LSR[4:1] will not be reset,
    // and so the flags will be used on the next
    // LSR read. Another benefit of this method
    // is that even slight timing differences
    // for LSR[4:1] reset, between the DUT and
    // monitor, won't be able to result to error.
  MONITOR_LSR[4:1] = 4'b0000;
};
```

In the above fragment of code, the monitor checks that the LSR bit 2 (parity error) is correctly produced by the DUT. The check is performed only when both the DUT and the monitor are synchronized. The reset method for the LSR that follows maintains the same principle; resetting the bits only when in synchronization, otherwise keeping them unchanged for the next check.

Data Flow Checking

Another challenge of constructing an *e*VC for a design that incorporates several interfaces is the difficulty in dealing with data flow checking. A common practice when verifying a design is the use of scoreboarding inside the drivers/collectors (stubs). However, in a multiple interface *e*VC, it is crucial to maintain the independence and scalability of the stimuli drivers. This allows for user extensions or even complete replacement without degrading or even removing the data and protocol checking capabilities of the environment.

A solution that offers more flexibility is a scoreboarding scheme that resides in the monitor. In this way, the *e*VC environment is able to gather the data flow on the interface level and thus be completely independent of the drivers. This can be achieved because a single *e*VC deals with all interfaces and so has access to all the information necessary to perform data flow checking. However, constructing such an unusual scoreboarding scheme can be quite complicated due to the large range of requirements already set forth in our discussion.

The following figures depict two options for scoreboard placement with respect to the UART *e*VC. In Figure 2, the conventional way is reflected with the scoreboard being updated immediately by the drivers. The items generated are added to the scoreboard and once received, a check is performed. However, in order to avoid driver-dependency, the UART *e*VC deploys the second alternative as depicted in Figure 3. In this solution, the monitor that stands on the interface gathers the items transmitted at each side and examines each transaction independently. The scoreboard can then remain active even in the absence of the drivers or when modifications are made inside the drivers by the UART user.

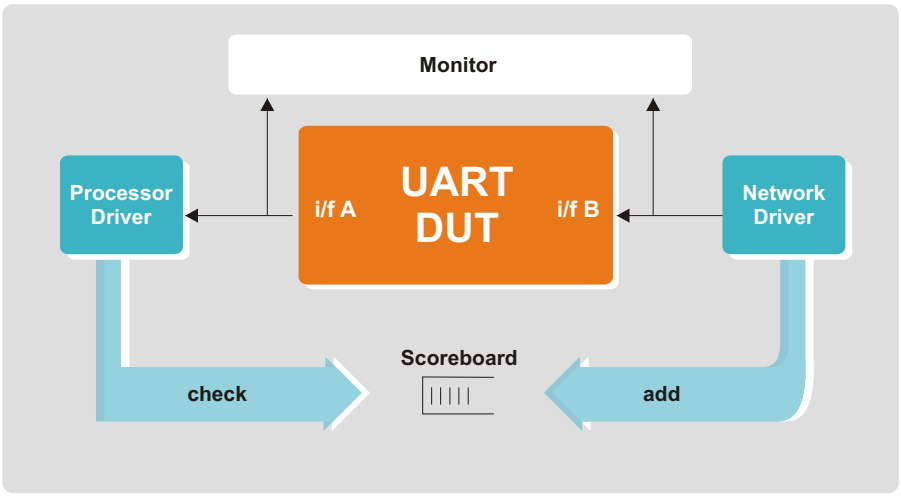


Figure 2: Implementing a device-level integrated scoreboard; data is added/checked by the stimuli drivers

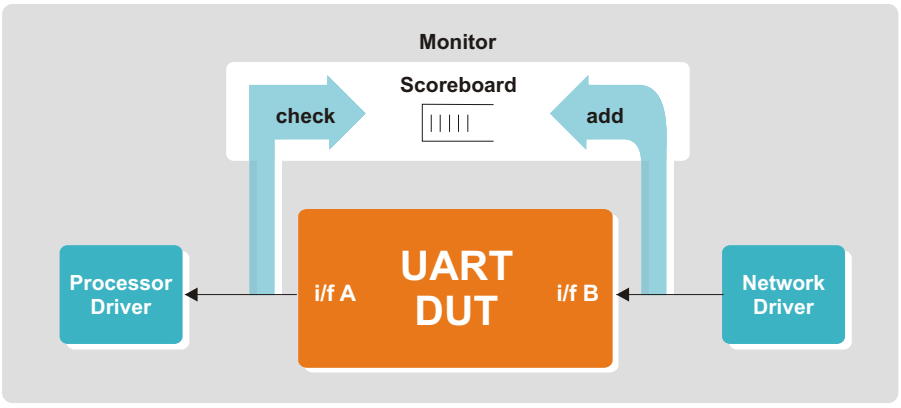


Figure 3: Implementing a device-level independent scoreboard; data is added and checked by the monitor

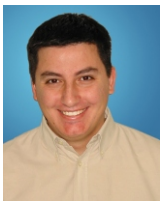
CONCLUSION

This paper presented a variety of uses for an *e*VC architecture that goes beyond verifying a single interface's protocol. When multiple interfaces are incorporated in a single *e*VC, the designer benefits from an increased range of valuable verification checks resulting in higher quality verification and better time-effort performance. This paper also presented several issues arising from this implementation method. The UART *e*VC was used as an example to describe how to overcome some of the obstacles such as cross-timing constraints, data flow checking and implementation independence.

BIOGRAPHIES

Thanasis Oikonomou is a digital systems designer at Globetech Solutions. His interests are in computer architecture, high speed network systems, FPGA/VLSI design, verification and testing. He received an MS and a BS in Computer Science from the University of Crete, Greece.

Aggelos Ioannou received his BS and MS from the University of Crete, Greece. His interests include switch architectures, micro-processor design and high performance networks. He is currently a digital systems designer at Globetech Solutions, working on verification of high-speed ASICs.



Stylianos Diamantidis is a founding partner of Globetech Solutions. His interests lie in the areas of verification platforms and IP, design for testability of complex systems, and distributed systems. He holds a BEng from the University of Kent, UK, and an MS in Electrical Engineering from Stanford University.

FURTHER INFORMATION

For further information, visit www.globetechsolutions.com
In addition, you can email us at info@globetechsolutions.com
or call ++1 650 988 6900 (US)
or ++30 23 10 31 35 53 (Europe)

Globetech Solutions is a member of Verisity Inc.'s Verification Alliance. Please contact us for information about *e*VCs and other exciting products and services.