Stateful sequence item and automatic UVM testbench generation

Jie Ding, Wen-hsing Chen

Qualcomm Inc.
Santa Clara, CA, USA

www.qualcomm.com

ABSTRACT

UVM is a very flexible technology used to construct testbenches and tests. It provides many constructs to adopters to facilitate fast and reusable testbench development.

However, these UVM constructs and syntaxes can be difficult to learn and apply for a new adopter. Different testbench construction styles are creating non-uniform testbench structures that can cause misunderstandings and errors when they are mixed together.

This paper discusses a stateful sequence item structure, that can help simplify UVM testbench construction and reuse. We present a new tool that can automatically generate reusable UVM testbenches.

Table of Contents

Table of Figures

1. Introduction

Bluetooth technology has been around for more than a decade[2]. The controller not only manages the radio but also facilitates the link layer protocols for Bluetooth operations. Over the years, the core technology went from Basic Rate, Enhance Data Rate, to Low Energy now. Numerous modes to support higher data rate, various profiles, lower power consumption and different usage scenarios enrich the technology along the way. At the same time, the richness in complexity presents a challenge in Design Verification to be adaptive to the changes and to efficiently validate the design.

Universal Verification Methodology (UVM) is a very flexible technology used to construct testbenches and tests. It provides Register Abstraction Layer, Phasing Extensions, Sequence Mechanism, TLM2.0 interfaces, Resource Database, End-of-test Mechanism and a command line processor [1]. All these new functions facilitate fast and reusable testbench development.

The core parts of UVM structure include TLM2.0 interfaces and Sequence Mechanism[3]. Because of their software origin, these UVM constructs and syntaxes can be hard to learn and apply for a new adopter. With added freedom in coding TLM2.0 interface and Sequence Mechanism, it is very easy to lose focus when creating a new testbench. Resources are spent on getting the testbench structure right, diverging from finding bugs. Still with all the efforts, the results can sometimes include non-uniform testbench structures with different testbench construction styles that can cause misunderstandings and errors when they are mixed together.

So, we started by examining the ever increasing need of quick testbench construction, focusing on bug finding. Then, we go deep into the very basics of testbench construction – returning data. There, we find a solution to this problem – Stateful Sequence Item. Then, we discuss the stateful sequence item structure and how it can help simplify UVM testbench construction and reuse. Finally, we will present a new tool that can automatically generate reusable UVM testbenches.

2. Design and Verification Complexity

The Bluetooth controller is in general split between the Link Controller and modem where the Link Controller communicates with the CPU/memory subsystem and modem interfaces with the radio. There are also many peripherals such as Crypto engine, audio codec, digital blocks in radio, FEM controls, and host interfaces in the SOC. As a wireless technology, coexistence with other radio technologies such as Wifi, LTE, GPS, FM, NFC becomes critical for performance. There are many interfaces to communicate with different blocks and devices to support the functionality aforementioned, such as AHB, APB, AXI, I2C, Slim bus, RFFE, UART, USB, and SDIO just to name a few. Numerous proprietary interfaces between blocks on the chip and across chips also exist to differentiate the products.

Not only are there many different blocks and interfaces to verify the operation, some may be required in certain configurations but not others or different configurations may be used between generations of chip and/or different variants in the same chip family. Similarly, in different phases of the development, for example the prototyping phase, using FPGA with a different generation of radio chips may also require different interface/blocks to emulate the Link Controller or modem individually. Design for debug also adds to the complexity, be it ARM core

architecture, proprietary test mux matrix, debug/probing or custom statistics where all the blocks and interfaces could be varying from project to project and require continuous support in validation.

All in all, the complexity of mixing and matching numerous blocks and interfaces imposes a great challenge for DV to verify them across projects. DV needs a flexible, scalable and efficient mechanism to create and link sequencers, different models, drivers, monitors and checkers dynamically depending on the requirements. The environments also need to be adaptive as new blocks and interfaces may be required in the future.

When constructing our testbench, we planned to use readily available Verification IP (VIP). For example, we used NVS AHB master, RGM register model, Crypto C library, GNU math library and several other Bus Functional Models (BFM). Some of these VIP are from external sources, some internal. Some are using UVM and others are not. They are of different levels of maturity and subject to change during our development. It is important to interface with these different VIPs so that our testbench can call the functions provided. Furthermore, it is more important to shield our testbench and test development from the changes of underlying VIPs. So, when the VIP makes a new release, or when we switch the vendor of the VIP, we will not affect internals of the testbench.

On the other hand, we need to support two connected but different targets. One target is to support Bluetooth IP development. We need to find bugs in design and provide regression to ensure quality and coverage. However, that is not enough for us. We also need to support SOC level testing and development by providing reusable building blocks for their testbench and even tests that can run at SOC level. Supporting SOC development can be especially challenging because we do not control SOC environment construction. In addition, we cannot anticipate future SOC environment. This is a common problem in Verification Intellectual Property(VIP) development[6].

It is clear to us that we are one link in the middle of the chain of testbench hierarchy. We are using building blocks from many different sources, and our Bluetooth testbench will also become a building block for other testbench environments. This situation posed several unique challenges to our testbench architecture.

3. Testbench Construction Challenges

Environment Hierarchy

We first start building our environment by providing agents to all DUT interfaces. Figure 1 shows the DUT and interfacing agents.

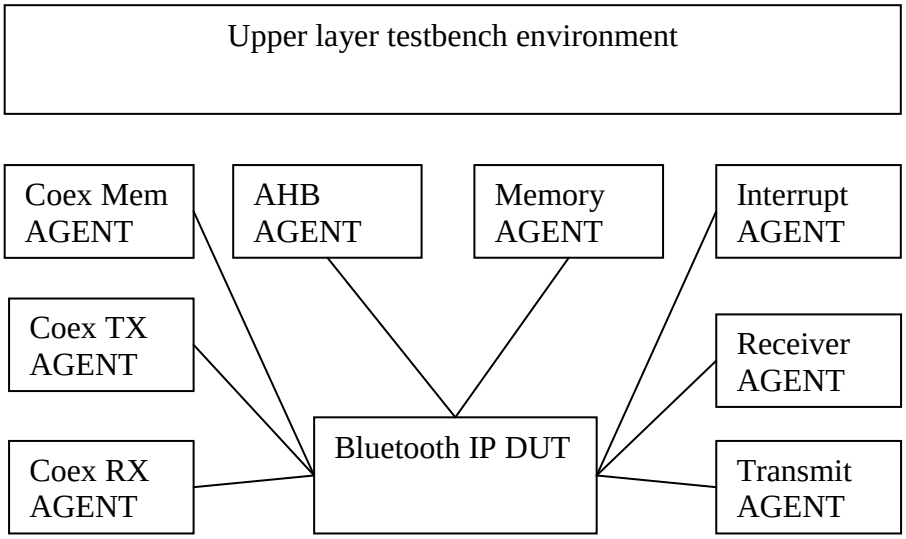

Figure 1 Testbench with one agent on each interface.

In our design, there are 8 interfaces, including AHB register bus, memory interface, CPU interrupt, coexistence transmit, coexistence receive, Bluetooth transmit, Bluetooth receiver and coexistence memory interface. We built and attached a UVM agent to each and every interface on DUT.

We built our virtual sequencer and virtual sequences on top of these agents. However, many of these agents only have limited functionality requiring us to build a substantial amount of testbench into upper layers of test environment. While the agents can handle the basics like reading or writing of registers, the upper layer environment will perform more complex operations like configuring a DUT mode, synchronizing packet transmission with DUT states, handling interrupts during receiving, and so forth.

After evaluating testbench complexity, it was obvious to us that a flat structure was not ideal for distribution of work, reuse of code and scalabilities. We needed a layered structure for our upper layer environment. We had two choices in constructing a layered upper level test environment. One option was to use virtual sequencers at the bottom and built a hierarchy of virtual sequences as shown in Figure 2.

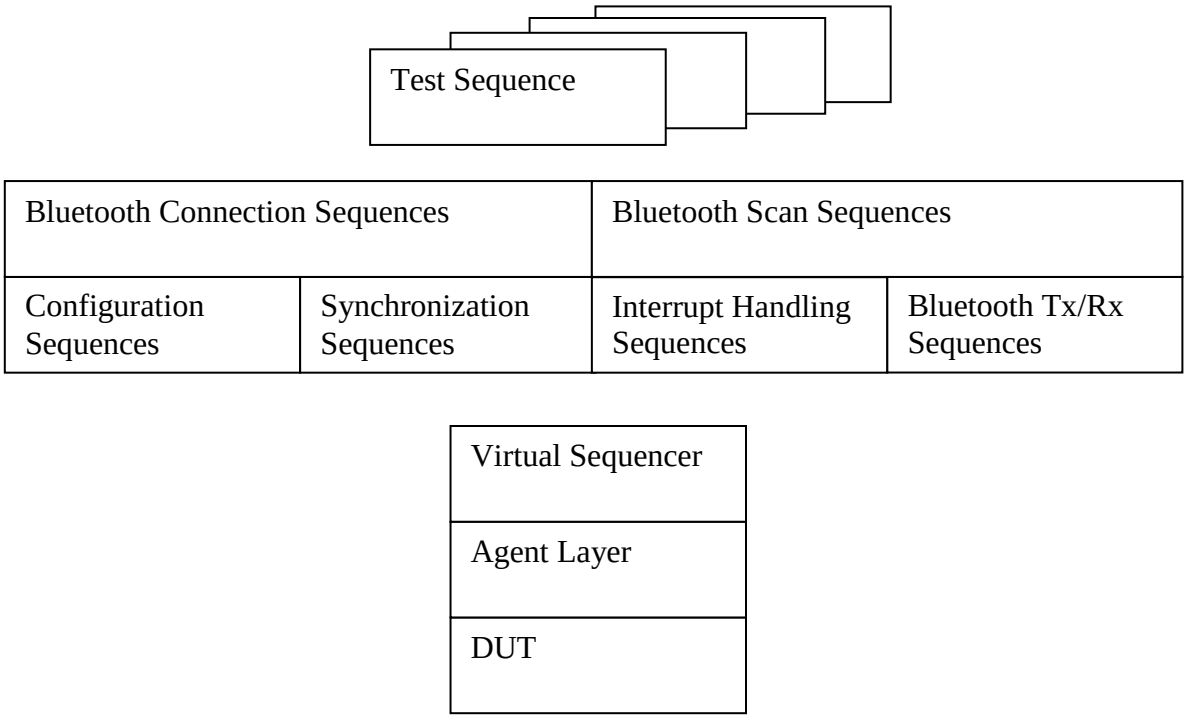

Figure 2 Option 1: Test Environment using Sequence Hierarchy

For Option 1, we would have built a virtual sequencer that has a reference to all agents. Then we would build configuration layer sequences on top of all agents. Above that, we would build process layer sequences to handle connection processes. Top-level test sequences call the services provided by this hierarchy.

The problem with this option is that we have asynchronous logics inside Bluetooth. If we build everything using sequence hierarchy, we will have to do a lot of fork/join operations inside the sequences. We would also have to do many synchronization operations the forked threads. Synchronization across hierarchical layers can be quite difficult to handle.

The other option we had was to build a hierarchy of agents. Instead of using virtual sequencers at the bottom of the hierarchy, we will keep it at the top of the test environment where parallel thread is not pervasive and synchronization among threads is minimal. Figure 3 shows our option 2.

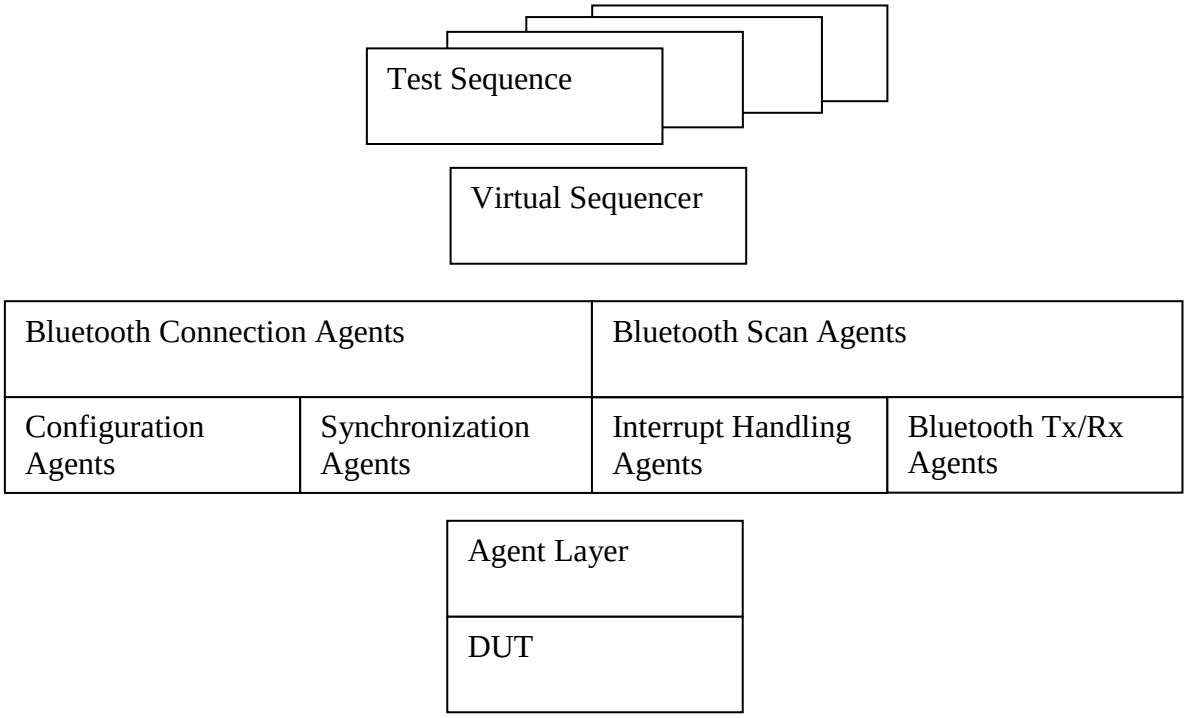

Figure 3 Option 2 Hierarchy of Agent with virtual sequencer at the top of the hierarchy

Note that in this option we built all supporting functionality into a hierarchy of agents. Thus, we could use Transaction Level Modelling (TLM) to handle inter-thread synchronization. We use sequences only on the very high level of the test environment where events happen sequentially or where only limited level of parallelism is present.

Return Data from UVM Driver

The biggest challenge we had when constructing our hierarchical environment was to find a scalable way to return the data response back to the caller. UVM provides a request channel for sending data to the UVM driver and a response channel for returning response and data back to the caller[4][5]. However, this method is hardly scalable. The following example shows the scalability problem when using request and response channels.

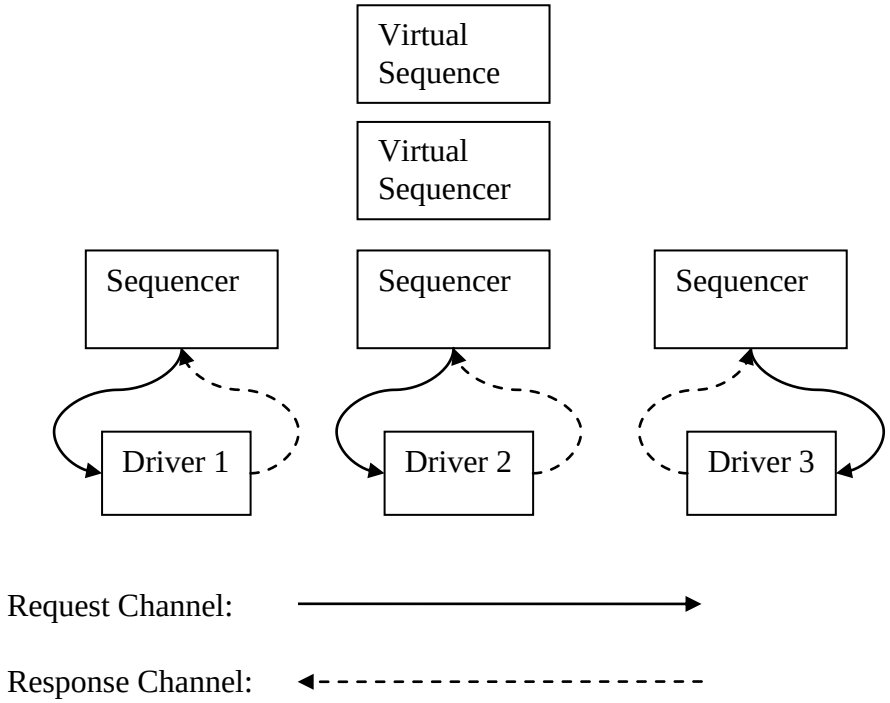

Figure 4 UVM TLM request and response model

In the figure, in order for the virtual sequence to communicate with three underline drivers, we have to make 6 connections, two connections for each driver. Also, we need to maintain three different sets of request ID inside the virtual sequence. If the upper layer is not a virtual sequence but is a UVM agent, then we have to make 6 connections for each agent supported. If three such agents exist at higher level, we will need to make 18 connections and maintain 3 request IDs in each agent. This can be very challenging when building and connecting TLM channels.

We developed a new method of communication between UVM components. Instead of creating two channels, one for request and one for response, we create only one channel. This channel is for request only. Instead of using request ID to match response with request, we use the request itself to identify the response. The procedure is shown in Figure 5.

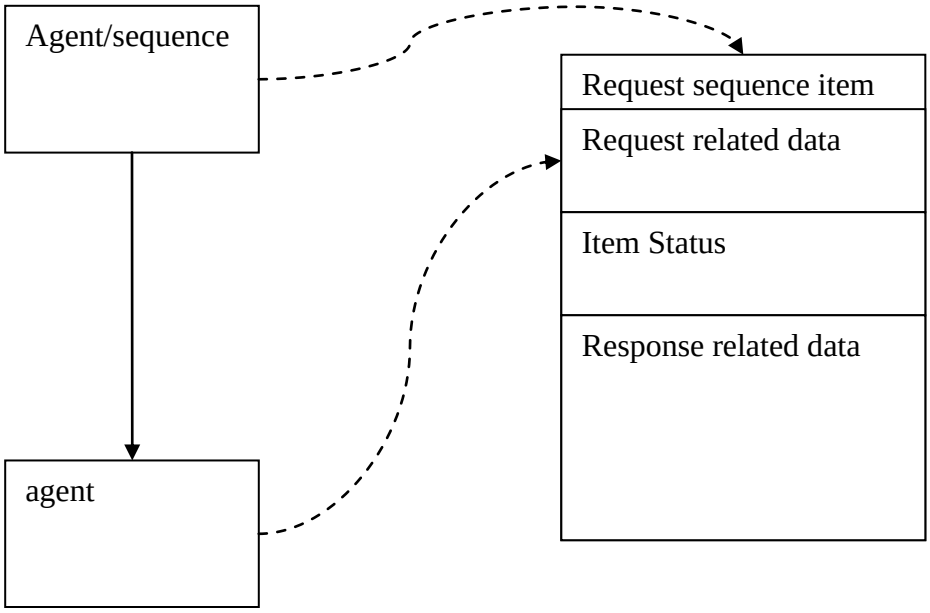

Figure 5 Stateful Sequence Item and request/response procedure

First, the upper layer sequence or agent creates a request sequence item. This sequence item contains three fields; request related data, response related data and item status. It will initialize the item status to IDLE, fill out the request related data then pass the sequence item to the lower layer agent. In UVM, when a sequence item is created, a handle is returned by the "new" function. The handle is then passed down the channel into the agent. So, now both upper layer sequence and lower layer agent each have a pointer to the same request sequence item.

A lower layer agent will then perform the request and update the response related data section with the response from DUT, then update the item status with RESPONSE_AVAILABLE. Meanwhile, an upper layer agent will perform a wait on the status change of the request item. When the status is updated by a lower layer agent, this action will un-block the wait in sequence. The sequence can then retrieve the response data from the request sequence item it created. Using this mechanism, the sequence or upper layer agent can exchange data using only one communication channel.

The key concept in this data exchange mechanism is that, as the creator of the sequence item, the sequence or agent has a handle to the request item. It shares the handle of the request item through the channel with lower level agents. After that, the executor of the request item also has the same handle to the same request item. Using this handle, both creator and executor can exchange request data and response data. Usually, this data share is unidirectional from requestor to executor. However this does not necessarily have to be the case. Provided that we can find a suitable way to synchronize the data sharing, we can not only pass the request data from requestor to executor, but response data from executor back to the requestor. One of such synchronization methods is Sequence Item State (SIS).

4. Stateful Sequence Item

Sequence Items State (SIS) is a single variable inside a sequence item. It contains a unique field to identify the state of the sequence item. All the UVM entities that hold a handle to the sequence item are in the *Access Circle* of this sequence item. Sequence Item Status is updated by the UVM components in the Access Circle. Access Circle entities will update the state of sequence item, after they perform actions related to that sequence item. Entities in Access Circle should monitor the SIS changes and react accordingly.

Any Access Circle Entity that needs to exchange data with other entities in the same circle should first update the data then change the Sequence Item Status to avoid a race condition. We recommend placing data update and status update in the same atomic code block.

After adding SIS to the sequence item, we define the sequence item as a *Stateful Sequence Item* (SSI). Compared with a stateless sequence item, a stateful sequence item has many benefits.

Isolation of network topology and changes

Stateful Sequence Item and Access Circle provide a way to isolate network topology changes from Access Circle Entities (ACE). Figure 6 shows an example of such isolation.

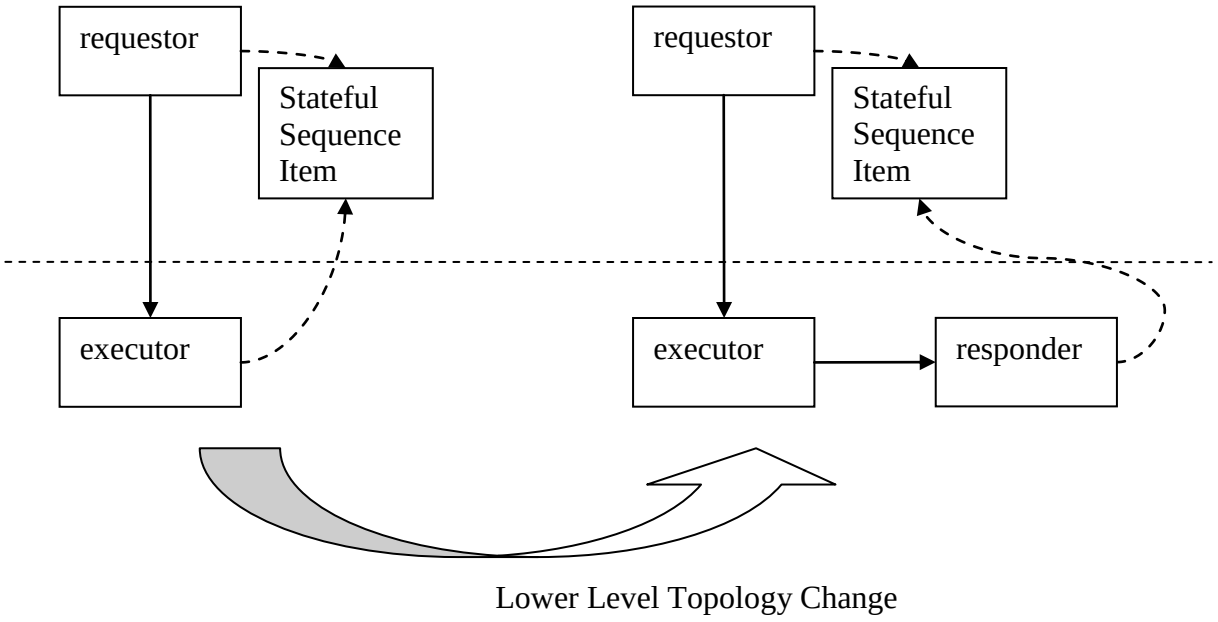

Figure 6 Topology changes at lower level of hierarchy does not affect upperlayer implementation

Figure 6 shows an Access Circle of two entities. One is the requestor and the other is the executor. The requestor has one channel to the executor and relies on SSI to get a response. If for some reason, network topology has changed, then a response has to come from a responder in the lower level hierarchy. We can simply build an Access Circle of requestor, executor and responder. We will let the responder to change the item status. In this case, the requestor does not have to make any changes to accommodate the lower layer changes. This demonstrates requestor implementation is isolated from the changes around it. This isolation greatly improves the portability and reusability of UVM components.

Out-of order data returning

Another benefit of using SSI is easy out-of-order data return. When the UVM agent returns data back to the sequence, we use ***request_id*** to handle out-of-order problems. We would set the request_id on each response back to sequence, so that the sequence can use request_id to match response with request.

This issue does not exist in SSI Access Circle. It is because all the responses to a specific request are placed directly in the request sequence item itself. Therefore, the requestor can search for the response in the very request it sends out. Response and request are match automatically.

Multi-phase out of order hand-shaking protocol

Another interesting usage of SSI is to perform multi-phase out of order hand shaking. In our Bluetooth testing, software can schedule multiple transmission/receiving with different start times and different execution delays resulting in different end times. This is an inherently out-of-order process. Furthermore, the requestor needs to perform reactive driving during different execution phases as shown in Figure 7.

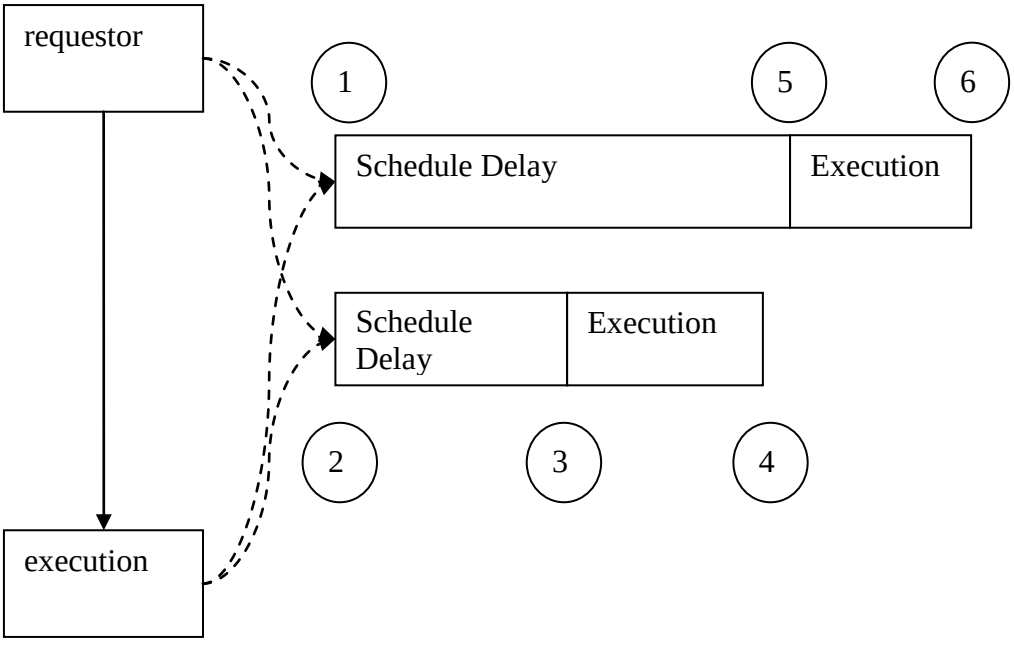

Figure 7 Multi-phase out of order hand-shaking protocol implemented by SSI

In Figure 7, the requestor will send two request items to execution. The first one has a longer schedule delay and the second one shorter delay. The request has two execution phases; delay and execution. At the transition of these two phases, data exchanges are performed between requestor and executor.

As before, executor will perform the operation and update SIS on each SSI. The requestor will monitor the progress and trigger on state change at time 3 and 5. Since each SSI status is updated separately, there is no mix-up of data exchanges due to out of order execution. Also, because the item status can be updated and triggered more than one time, execution phases are easily modelled using SSI.

A more complex example may involve multi-entities in Access Circle.

Multi-entity multi-phase hand-shaking protocol

When we implement the execution unit in an Access Circle, many times we would have more than one execution unit. The sequence item could also have more than one execution phase. Finally, the sequence items could complete out of order. Using SSI, this scenario can be easily implemented per the example in Figure 8.

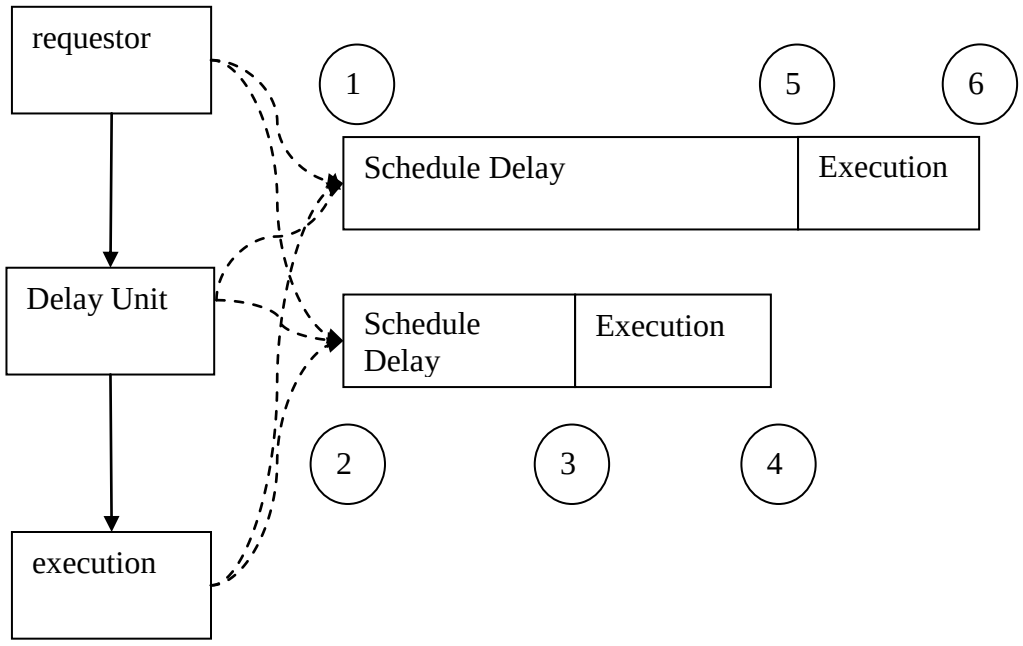

Figure 8 Multi-entity multi-phase out of order Access Circle

In the above Access Circle, delay and execution are separated into two units. The requestor will first send a request to a delay unit. The delay unit will forward the request to execution unit as well. Now for both sequence items shown in the figure, there are three entities in their respective

access circles. They will all have access to the sequence item data and status. Each will perform its own action and update the status accordingly. So we can easily support this model with SSI.

Stateful Sequence Item Implementation

We implemented SSI as a uvm_object. To make setting/getting easy, status is modeled as a string. Status change is a event that indicate a status change. The following code shows the core implementation of Sequence Item State.

```
class sis_base extends uvm_object;
…
local string status;
event       status_change;
…
task tb_status::set_as(string status);
      process_sem.get(); // Don't remove this, this is needed to resolve race
                         // condition
#0;                      // Wait a delta time for all other wait to get triggered.
      this.status = status;
      -> status_change;
      process_sem.put();
endtask

task tb_status::wait_for(string status, time timeout = 0);
      while(this.status != status) begin
            @(status_change);
      end
endtask
…
endclass
```

In the code above, we also have to protect the status change with a semaphore so that the set_as function will not be entered by more than one thread. To prevent a race condition, we will perform a delta cycle wait before we change the status. This delta cycle is crucial to the system event scheduler because it will allow the event change to be “seen“ by all other waiting statements before a new change of status is allowed. Finally, we will change the status, trigger status_change event and exit the protection of the semaphore.

In the wait_for task, we will first check if current status is triggered. If not, we will keep checking all status changes, until the specified status is triggered.

This simple yet powerful code allows us to model and construct our Bluetooth test environment in a manner that is versatile yet scalable. There are however, several issues that we should address regarding the use of Stateful Sequence Item.

The issue of global data

Is item status global data? Stateful sequence items are pointed to and manipulated freely among the uvm_components in the Access Circle, so item status appears to be global data. But on closer look, it is very different from global data. First, global data is created at the start of the program and stays in memory until the program terminates. SSI is, however, created by the requestor and shared in Access Circle. It is destroyed when all actions are executed and all handlers to the object are freed. Then, it will be destroyed by garbage collection.

Global data has no access control and is shared by all code blocks, regardless of relevence. So, some totally irrelevent code could accidentally access the data, making it very hard to debug and maintain.

Stateful sequence items will always remain inside the access circle where all the entities are relevent to the sequence item. When a UVM component completes its work with this sequence item, it can drop the access to the sequence item by dropping the handle to the sequence item.

A stateful sequence item has non-global access control, does not exist in memory from beginning to end and is therefore safe to use.

Separate response/request data

Although Stateful Sequence Item is safe to use in sharing data between UVM components, we have to take precautions to avoid unnecessary debugging difficulties. A stateful sequence item contains all data from requestor to execution and back. The data must be kept separate between different flow directions. We separate the data into two or more groups so that requestors only access request data and responder only access response data. For example, in AHB agent, we kept read_data and write_data separate. Read and write data are not shared. When read_data changes, we know it is the responder who changed it. And when the write_data changes, we know it is changed by the requestor.

The greatest benefit of using SSI is that it helps us build our automatic UVM TLM framework generator.

Efficiency

When we add response data into request, the request object becomes bigger and takes more memory space. Some may question the efficiency of this approach when we have to move a big object inside a UVM network. However, instead of hurting the efficiency, this is going to help.

The key concept in understanding efficiency is to understand the difference between object and object handle. UVM objects contain user data and take memory space when we add more fields. Object handles only point to the location of the object, thus the size of handle remains constant. When we move the UVM sequence item around inside the testbench, the object is not copied or moved inside the UVM component network. Only the object handle is moved. The object handle is copied and moved around inside the network with a constant cost regardless of object size. So, even if we increase the size of the object, we will not increase communication cost.

Creating or destroying object and handling traffic may use additional processing power and cause efficiency reduction. Because we add response and state in the sequence item, we no longer have to create separate response object nor handling additional response traffic. We also eliminate the need to match the response with request using ID. This improvement not only makes the testbench easy to understand, but greatly improves testbench efficiency.

5. Network description and Automatic UVM testbench generation

UVM TLM and sequences are two very useful facets of the tool set. TLM can better model the parallel events because all uvm_components are parallel threads. Data exchange and synchronization are also provided by different channels and ports. Sequences better model the sequential events. They are normally executed sequentially and are very easy to track.

When doing the Bluetooth testbench, we needed a quick and automatic way to construct UVM the testbench including both TLM modelling and sequences. We looked at a few readily available options and eventually created our own Network Description Language and UVM generator script.

Network Model and Description Language

We combine sequencer and tlm_fifo_channel into one access port type. This access port type is used to access UVM agent. Then, we combine uvm_driver and uvm_monitor inside the agent into one entity. That gives us *Transactor*.

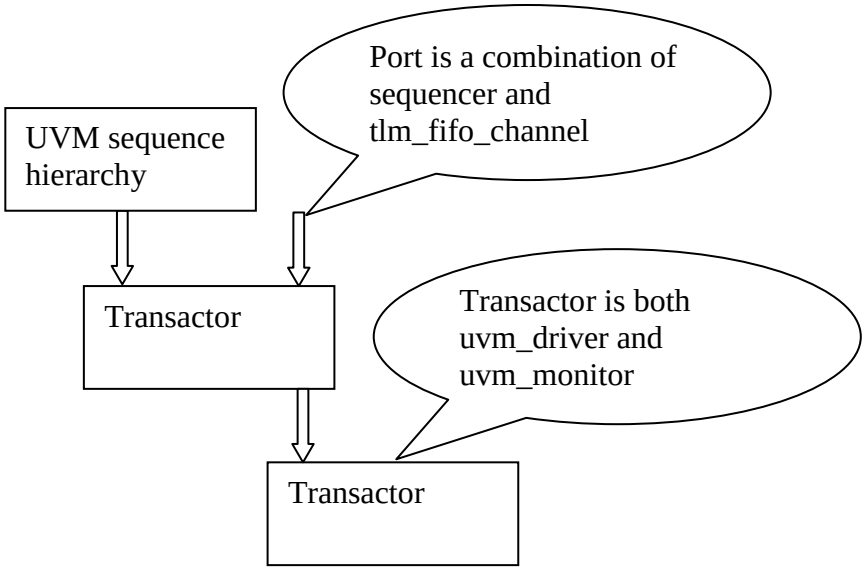

Figure 9 Network model for quick uvm code generation

In Figure 9, each transactor is a combination of driver and monitor. It can have multiple input and output access ports. Access Port is used to access the functionalities provided by drivers and monitors inside the transactor. Access Port combines tlm_fifo and sequencer, so that both sequence and uvm_component can send Stateful Sequence Item into drivers and monitors in the transactor. Since this network model is very regular and simple, we designed a language to

describe this model and automatically generate UVM code. Here is what we have for the transactor:

```
xactor my_xactor( <input_port_type iport>  ->  <output_port_type oport>)
{
        [code “my_xactor.sv”];
}

xactor my_env_xactor( <input_port_type iport>  ->  <output_port_type oport>)
{
        my_xactor   a, b;
        iport -> a.iport;
        iport -> b.iport;
        a.oport -> oport;
        b.oport -> oport;
        [code “my_xactor.sv”];
}
```

The code above describes following network.

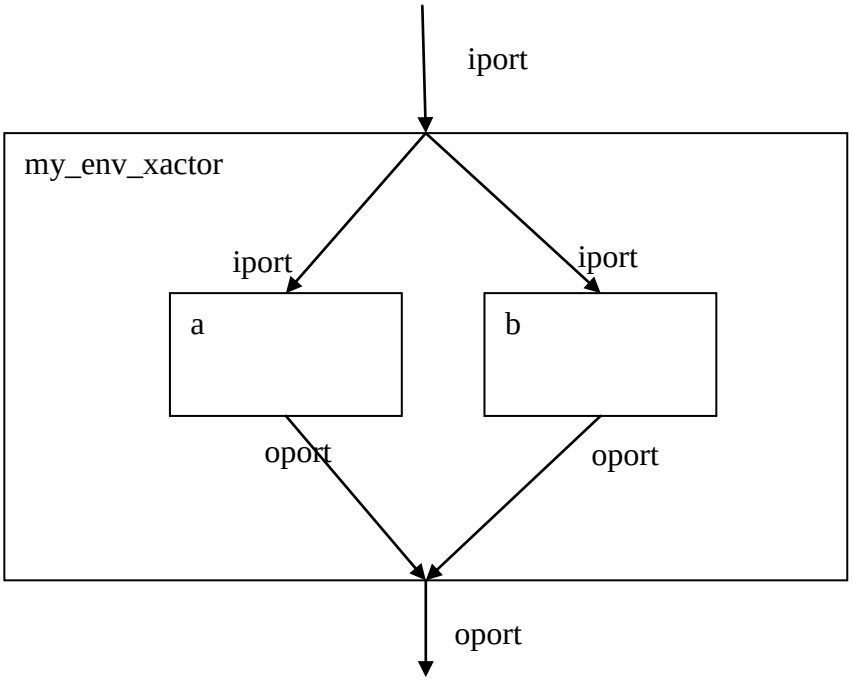

Figure 10 Network Model described in the code above

Note that each *input_port_type* or *output_port_type* is derived from Stateful Sequence Item base. Therefore they are all Stateful Sequence Item. Generator will automatically include a *sis_base* object inside.

Our compiler will automatically inject a state variable into every each sequence item it creates, so the user can take advantage of synchronization using state variables.

Finally, we run our generator script and proper code is generated for us automatically.

6. Results

The flexible UVM based environment provides Design Verification in Bluetooth Controllers in a scalable testbench. With a base construct to support Link Controller and modem simulation together, simulation can be executed at block level or with Link Controller standalone with a CPU subsystem. Design teams in Link Controller and modem blocks can work independently. On the other hand, both design teams can cooperate together in the combined testbench to co-simulate the behaviour between them.

The same UVM methodology is extended beyond the Bluetooth Controller. The testbench has been extended to include the radio modelling to complete the end-to-end verification of Bluetooth operations. Leveraging from end-to-end Bluetooth testbench, verification of coexistence with other wireless technologies like Wifi and LTE are developed efficiently by sharing and reusing components.

We implemented Stateful Sequence Item and the automatic UVM testbench generation tool to help us generate testbenches quickly and efficiently. We were able to support both IP level verification and SOC level verification with this method. We were also able to bring new engineers up to speed with testbench development easily.

Compared with traditional inter-agent communication methods, the Stateful Sequence Item approach reduced communication overhead, cleaned up request-response matching processes and improved performance by reducing object creation/destruction frequency and reducing traffic inside the UVM component network. Overall, we are able to focus on DUT testing instead of testbench developing and debugging.

7. Conclusions

The method created a status class inside each sequence item, so it helps to return data from uvm_driver back to uvm_sequence without the need of response channel and set_id_info / get_id_info.

Based on this method, we have a network description language to describe the UVM testbench hierarchy. We call it “xf” language. Using this simple and intuitive language, we can quick specify the components and connectivity of the testbench. Then we create a compiler to compile the network described in xf language and generate UVM code automatically. The compiler can handle the generation of all UVM components, connectivity, overrides and derivation.

Combining stateful sequence_item and network description language, we can quickly construct a complex testbench from scratch. We can also reduce human error in coding UVM testbench and connectivity. Finally, the method is simple, intuitive and easy to maintain.

8. References

[1] Universal Verification Methodology (UVM) User’s Guide. Accellera www.accellera.org

[2] The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology, Ray Salemi, Oct 23, 2013

[3] Doulos UVM Golden Reference Guide, John Aynsley, David Long, Doug Smith and Mike Smith, Dec 22, 2013

[4] A Practical Guide to Adopting the Universal Verification Methodology(UVM), Sharon Rosenberg and Kathleen Meade, Aug 18, 2010

[5] Verification Intellectual Property(VIP) Recommended Practices, Accellera, Aug 25, 2009