ACE’ing the Verification of a Coherent System Using UVM

Romondy Luo
Synopsys
Zhaofeng Plaza,
Shanghai
+86.212.307.2379
rluo@synopsys.com

Ray Varghese, Parag Goel, Amit Sharma, Satyapriya Acharya
Synopsys RMZ Infinity, Bangalore,
0091.80.40180000
{rayrv, paragg, amits, acharyas}@synopsys.com

Peer Mohammed
MindSpeed Technologies
Hyderabad
+91 40 43402429
peer.mohammed@mindspeed.com

ABSTRACT

The need for higher processing capabilities while continuously optimizing on power consumption is imperative in most of the electronic appliances that we use today. This is fed by the requirement to run compute intensive applications such as video editing, image processing, image recognition, gaming engines and so on. It is well understood that multi-processing is a more efficient mechanism to achieve this instead of pushing the boundaries of single core processor. In addition, there is a need to move towards virtualization of the SoC’s, which increases the processing power efficiency, by enabling the ease of feature integration with minimum effort. However with multi-processor intensive SoC’s, there is a requirement to maintain coherency across the system (including peripherals and the main memory). It is critical to have multiple applications working together either simultaneously or in synchronism and the new AMBA 4 Coherency Extensions (ACE) helps in this scenario. ACE provides hardware-level cache coherency and is primarily aimed at multi-processor ARM Cortex-A15 designs for mobile devices.

Now, verifying and validating a coherent system can pose significantly complex problems for the verification engineer. This is where a robust verification methodology can come to the rescue. A powerful Verification Methodology empowers the verification engineer to help leverage the constrained random capabilities in the language to create configurable stimulus to help verify a wide range of functionalities. UVM-1.1 which was released recently provides a very flexible mechanism of achieving this. Thus, by using the case study of the Synopsys® Verification IP (VIP) for AMBA AXI, we will demonstrate how a set of sequences and a sequence library along with other UVM base class functionalities can be leveraged to meet the various challenges in the system-level validation of such cache coherent systems.

Categories and Subject Descriptors
Methodology – UVM, VIP, re-use philosophies

General Terms
Management, Performance, Verification, Design, Standardization

Keywords
AXI4, ACE, VIP, UVM, Layered Protocol, SystemVerilog

1. INTRODUCTION

The AMBA 4 specification now features AXI coherency extensions (ACE) in support of multi-core computing. The ACE specification enables system-level cache coherency across clusters of multi-core processors. When it comes down to the functional verification of such a system, these coherency extensions brings their own complex challenges. Some of these can be,

- *System-level cache coherency validation:* At any given time, whether the ACE Interconnect is able to maintain cache coherency across the different ACE Masters in the system.
- *Cache state transition validation:* Whether the ACE interconnect is able to handle all cache line state transitions in ACE Masters in the system

At a high level, this requires a high degree of configurability and responsiveness in the stimulus generation infrastructure. In terms of validating the System-level cache coherency, a robust checking mechanism also needs to be created. We show how the UVM configuration mechanism is leveraged to bring in the configurability of the sequences. This mechanism also enables the reactive sequences to create the right stimulus for the respective VIP (Master/Slave/Interconnect) components. Given that the coherency has to be maintained across multiple masters, this has to be enabled through the system-level components (and also sub-system). Using the UVM resource mechanism and ACE interconnect in different modes(Active/Passive), we demonstrate how to check the cache coherency using a combination of front-door and backdoor accesses to it. The UVM hierarchical phasing schemes and configurable sequences are also leveraged to model various transitions for the system which ensures complete verification closure. To handle such a complex system, an appropriate debug environment is also provided to the verification engineer to debug the environment at different levels of abstraction using the base UVM infrastructure.

2. OVERVIEW OF THE ACE PROTOCOL

Cache coherency refers to the consistency of data stored in the local caches of a shared resource. When clients in a system maintain caches of a common memory resource, problems might arise with inconsistent data among caches or main memory. This is particularly true for CPUs in a multiprocessing system. The cache coherence is intended to manage such conflicts and maintain consistency between cache and memory.

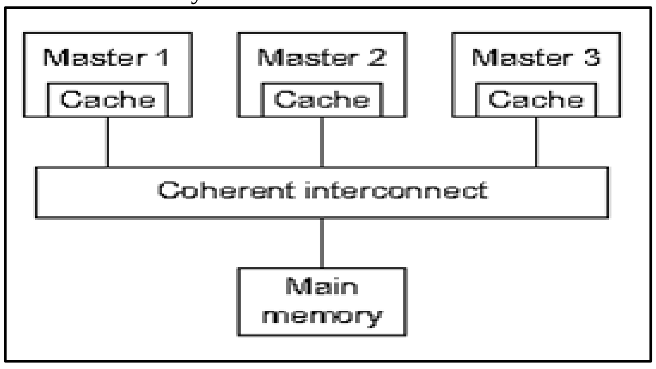


Figure 1: Cache Coherent Components

The ACE protocol extends the AXI protocol and provides support for hardware-coherent caches. The ACE protocol is implemented by using,

- A five state cache model to define the state of any cache line in the coherent system. The cache line state determines what actions are required during access to that cache line.
- Additional signaling on the existing AXI channels that enables new transactions and information to be conveyed to locations that require hardware coherency support.

- Additional channels that enable communication with a cached master when another master is accessing an address location that might be shared.[6]

2.1 COHERENCY MODEL

The ACE protocol ensures that all master components observe the correct data value at any given address location. The coherency protocol ensures that all masters observe the correct data value at any given address location by enforcing that only one copy exists whenever a store occurs to the location. The following is an example of an ACE based Coherent System:

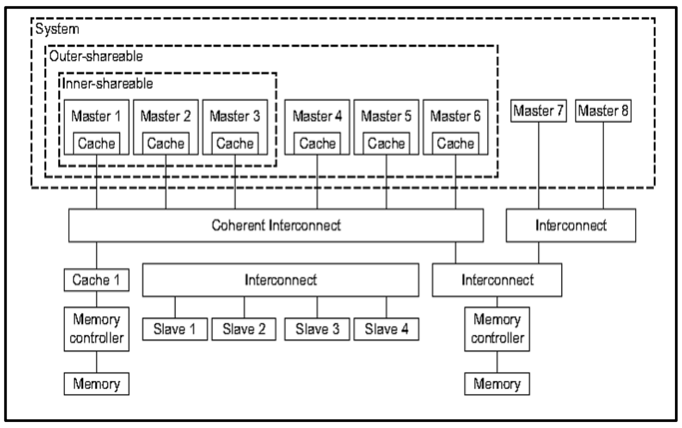


Figure 2: Cache Coherent System

The masters initiate requests and often contain a cache. An Interconnect connects one or more masters to one or more slaves. When a transaction requires coherency support, it is passed on to the coherency support logic within the Interconnect. After each store to a location, other masters can obtain a new copy of the data for their own local cache, allowing multiple copies to exist. The Interconnect can initiate "snoop" transactions to access cache lines in the master cache.. A cache line is defined as a cached copy of a number of sequentially byte addressed memory locations, with the first address being aligned to the total size of the cache line. There is no requirement to keep main memory up to date at all times. Main memory is only required to be updated before a copy of the memory location is no longer held in any shareable cache.

The ACE protocol enables master components to determine whether a cache line is the only copy of a particular memory location, or if there might be other copies of the same location, so that,

- If a cache line is the only copy, a master component can change the value of the cache line without notifying any other master components in the system

- If a cache line that is also be present in another cache, a master component must notify the other caches, by using an appropriate transaction.

Besides these, there are additional specifications centered on the granularity of coherency, access rights, cache line state updates, protocol transactions, protocol channels and transaction flows. [6]

3. CHALLENGES IN THE VERIFICATION OF A CACHE COHERENT SYSTEM

With coherency support now in the hardware with an associated protocol to support it, the complexity of the system and the underlying components has increased substantially. The verification of such systems needs to deal with several challenges.

3.1 COMPLEX STIMULUS REQUIREMENTS

An ACE system can have a variety of Masters and Slaves connected by a coherent interconnect. Individually each Master and Slave component can support complete ACE, ACE-Lite, AXI4 or AXI3 protocol and might work with different bus width or clock frequency. The different permutations involving the following parameters at large:

1. Cache states,
2. Transaction types,
3. Burst lengths, burst sizes,
4. Snoop mechanisms, snooped cache states, snoop responses,
5. Support for speculative fetches,
6. Support for snoop filtering, and
7. User-specified scheduling of interconnect.

All these cross combinations lead to a very large verification space. The challenges for generating stimulus mapped to all of these include:

- Ensuring each individual Master, Slave or Interconnect is fully compatible with the protocol it supports
- Ensuring all possible combinations of concurrent access among initiating masters, snooped masters and slave main memory are verified and are in compliance with the ACE specification
- Ensuring all user-specific features are covered and working as expected
- Providing a complete coverage model to ensure the completeness of verification.

3.2 SYSTEM LEVEL SELF-CHECKING

The ACE System can have a complex configuration of Masters, Slaves and Interconnect. Some of them can be RTL components with or without a local cache, while others can be VIPs or behavioral models. With such complex configurations, the system-level checks should be able to handle a lot of complexity. Some of these are:

- Most of the details of an AXI3/AXI4 transaction are presented on the bus. However, for an ACE system, many details of an ACE transaction are not presented on the bus. Some of these are not propagated as it gets routed through the Interconnect. Some of these attributes map to the *store/load/update/evict* of the local cache line by the ACE Master, or by Interconnect which initiates a *snoop*

- Multiple Masters would send out coherent transactions and at the same time receiving snoop transactions that might access the same location

- The ACE Master and Slave components support outstanding, interleaving and out-of-order transactions

All these make it very difficult to predict what the expected results should be. The more complicated checks would be,

- ***The snoop mechanism***: The system-level checks should be aware of which master should or should not be *snooped*. The information such as the cache line state in each master or the source master of a snoop transaction will not be present on the bus. There might be instances of a user-defined snoop scheduling. For example, the snoop can be done in a broadcast manner, which means all the snooped master are snooped at almost the same time with the same type of snoop transactions, or in a sequential manner, which means the snooped masters are snooped at different time with different type of snoop transactions.

- ***The snoop response***: Each ‘snooped’ master should respond properly based on the cache line state in its local cache. However, the cache line state of a ‘snooped’ master changes dynamically between the “*snoop start phase*” and “*snoop end phase*”. These states can encompass an implicit local store or invalidation sent by it, an explicit *writeback/writeclean/evict* transaction sent by it, or any unfinished outstanding snoop transactions on its channel sent by other Masters.

- ***The slave memory access:*** The system-level checker must check for the appropriate slave operation that must happen on the slave bus for an access. A slave access can be generated from a non-snoop transaction, a *speculative fetch*, cache *miss fetch* or a PassDirty propagating (Refer APPENDIX). It can also result from partial line merged to full line propagating; or direct coherent write.

- ***Data integrity check***: Different transactions can have different actual (data in the cache or in the main memory)/expected (local data queues) data sources. Even for a single transaction the data sources can be varied. The system-level checker must monitor the whole ACE system to decide where the expected/actual data should come from. For example, a *ReadOnce* transaction from ACE master can have the expected data source on any of these: a local cache copy before the transaction is sent (*speculative read*); the local cache copy after a coherent response (*allocate after read*), a snooped master’s cache (*cache hit*), a slave memory (*cache miss*) and an ACE bus (does not allocate after read). The data checking becomes all the more challenging because as interconnect can modify a transaction issued from coherent master. So it is not easy to map a slave memory access to a coherent transaction initiated by a master.

- ***Cache coherency check***: The system-level checker must monitor the whole ACE system to decide when and what type of operation might cause a cache line change. Whenever there is a cache line change, it needs to check if there is any loss of coherency. The checks include “cache vs. cache” comparison and “cache vs. slave memory” comparison. The two important cacheline checks are as follows:

  1. A cacheline can be held in a dirty state only in one master's cache
  2. A cacheline can be held in a unique state only in one master's cache

- ***User-specified features***: Such might have impact on the prediction of the expected behavior. For example, interconnect scheduling/priority can affect the behavior of snoop and slave accesses. Support for unaligned/cross line access can affect how many times the snoop/slave access will happen. Support for different bus width can affect the burst type and burst length difference between masters and slaves.

3.3 REUSE REQUIREMENTS

Horizontal and Vertical reusing of block-level environments has its own set of unique challenges. In the context of vertical reuse, some of the ACE components can be replaced with actual RTL models.
Thus, the testbench or the verification components should provide the infrastructure to be able to factor in the behavior of the RTL components at various protocol phases, for example, during the initiation state of an ACE DUT master, the end state of a snoop transaction, the cache state transition after a local store/invalidation and so on. Horizontal reuse or reuse across the projects can be more complicated.

The different projects can have different numbers of Master and Slave components complying with a different subset of the complete protocol. The large number of ACE masters with different levels of protocol compliance complicates the expected result of a coherent transaction. The increasing number of caches leads,

1. To more concurrent overlapping accesses to the same location,
2. To more complex snoop scheduling and responses

This makes it more difficult to predict the snoop hit or miss. To support the maximum reuse, configurability of different verification components is vital.

4. UVM STIMULUS GENERATION INFRASTRUCTURE OVERVIEW

It is quite apparent that the stimulus generation schemes have to be sophisticated and configurable to meet the complex verification requirements of the ACE cache coherence capabilities. The UVM base class library provides significant functionalities in string together a robust and configurable stimulus generation infrastructure.

uvm_sequence is the basic building block to help model complex verification scenarios. The figure below illustrates the basic UVM stimulus generation mechanism.

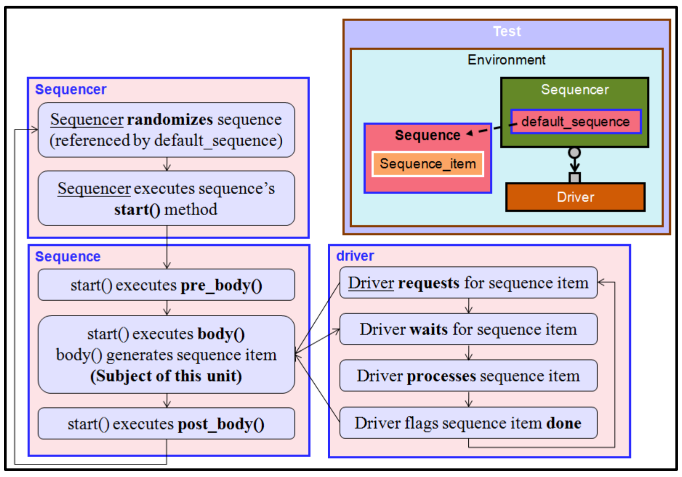
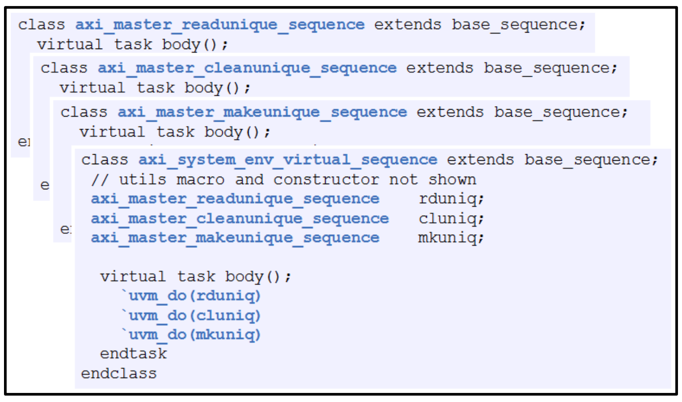

Test
Environment
Sequencer
default_sequence
Sequence
Sequence_item
Driver
Sequencer
Sequencer randomizes sequence (referenced by default_sequence)
Sequencer executes sequence’s start() method
Sequence
start() executes pre_body()
start() executes body() body() generates sequence item (Subject of this unit)
start() executes post_body()
driver
Driver requests for sequence item
Driver waits for sequence item
Driver processes sequence item
Driver flags sequence item done

Figure 3: Stimulus Communication with Testbench Components

Typically, one would start with some atomic sequences and move towards creating complicated ones as the basic functionality is verified. Instead of creating a flattened logic in the sequence body, hierarchical or nested sequences can be created which leverage the basic sequences which have already been created.



```
class axi_master_readunique_sequence extends base_sequence;
  virtual task body();
  class axi_master_cleanunique_sequence extends base_sequence;
    virtual task body();
    class axi_master_makeunique_sequence extends base_sequence;
      virtual task body();
      class axi_system_env_virtual_sequence extends base_sequence;
       // utils macro and constructor not shown
       axi_master_readunique_sequence    rduniq;
       axi_master_cleanunique_sequence   cluniq;
       axi_master_makeunique_sequence    mkuniq;

        virtual task body();
          `uvm_do(rduniq)
          `uvm_do(cluniq)
          `uvm_do(mkuniq)
        endtask
      endclass
```

Figure 4: Nested Sequences

This can go upto multiple levels of hierarchy and thus it becomes possible to converge towards meeting the requirements of the most complex scenarios.

As the complexity increases across multiple ACE components, there might be a requirement to coordinate the sequences across multiple sequencers and drivers. This can be handled through virtual sequences and sequencers

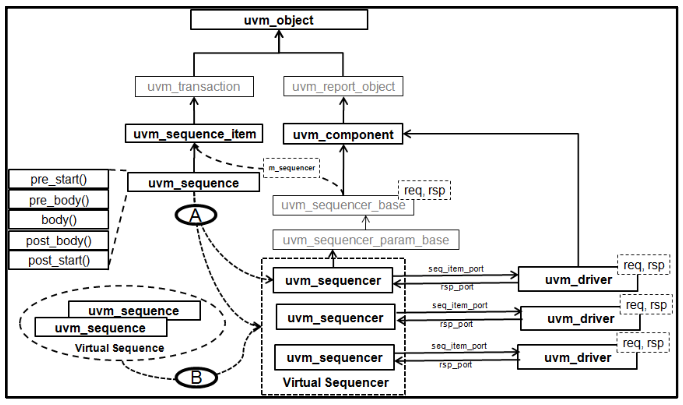
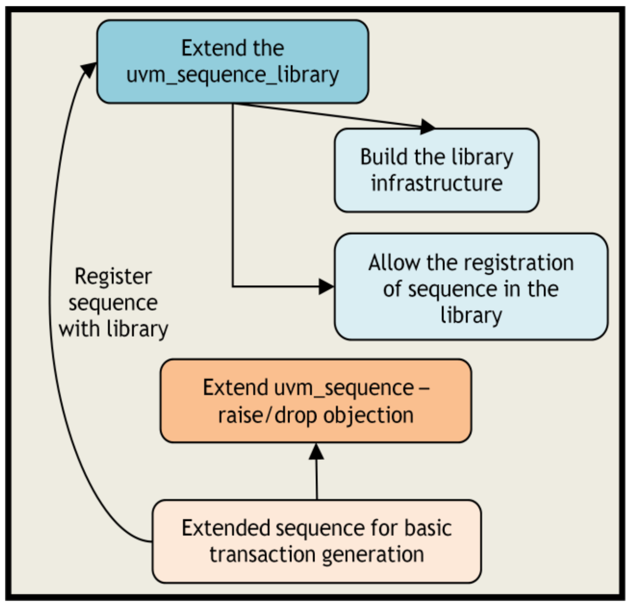


Figure 5: UVM BCL Interaction

Now each 'sequence' extending from *uvm_sequence* has a reference (*m_sequencer)* to the sequencer on which it is supposed to execute. So, whenever a sequence is executed on a sequencer either directly through a sequence.start() or by setting it as the *default_sequence* of a sequencer in a specific phase, the '*m_sequencer'* variable is appropriately set. A virtual sequence which basically orchestrates multiple sequences can execute only on the virtual sequencer. It is ensured that the individual sequences within are appropriately executed on the desired sequencer by mapping these to the real sequencers instantiated within the virtual sequencers (route 'B' in figure 4). This infrastructure can also be leveraged to enable the sequencer composition to create a layered protocol implementation

The other important functionality from a stimulus generation perspective is the grouping of sequences and the creation of hierarchical sequences. In UVM, it is possible to group similar sequences together into a sequence library.


Extend the uvm_sequence_library
Build the library infrastructure
Allow the registration of sequence in the library
Register sequence with library
Extend uvm_sequence – raise/drop objection
Extended sequence for basic transaction generation
Figure 6: UVM Sequence Library Package

The *'uvm_sequence_library'* is used to create a sequence library. The *`uvm_sequence_library_utils*(class_name) would build the library infrastructure. Any sequence can be registered to the sequence library through the *add_typewide_sequence()* method of the library.

Once the library is registered to be the *default_sequence* of any sequence, the default functionality causes a random number of sequences to be picked up and executed. Now, the default mode of sequence library can be modified by changing the parameters of the *uvm_sequence_library_cfg* class. The user can cause specific number of sequences to be picked up, enable random cyclic sequences and can also program a user defined sequence execution. Hence, without having to write multiple tests, the user can create user defined sequence execution across multiple sequence libraries across different interfaces through a virtual sequencer to create a stimulus management setup which would help meet all the stimulus generation requirements much faster.

5. OVERVIEW OF THE UVM Based ACE VERIFICATION VIP

The Synopsys Verification IP (VIP) for AMBA AXI is a suite of SV UVM-based verification components that provide a complete UVM-based verification solution for ACE protocol. The AXI ACE VIP provides a System Environment component with a configurable number of ACE Master and Slave agents, a System monitor and an Interconnect component. The VIP also has a purely SV architecture that eliminates the need for wrappers giving much improved performance and complete alignment with standard SystemVerilog methodology. The architecture without any gaskets enables users to leverage the underlying SV UVM methodology to the fullest extent. This also brings in inherent visibility and control. As the entire VIP runs natively in the simulator, there are no layers to slow it down. New features can easily be added to help in protocol centric debug and in providing a simple configuration interface to the VIP. This also ensures portability across all simulators. The VIP leverages most of the functionality mentioned in the previous sections and the UVM Resource Mechanism to give the configurability and the sophisticated stimulus generation requirements in the ACE context.

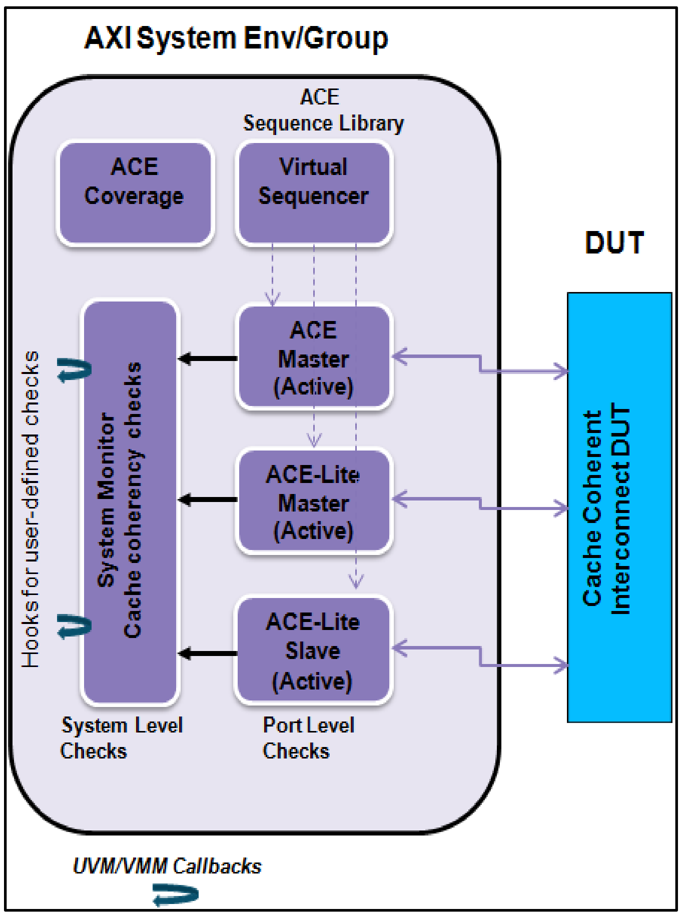


Figure 7: AXI ACE System Environment

The Master agent generates constrained random ACE coherent transactions, and responds to the ACE Snoop transactions concurrently. It also allocates cache lines and performs cache state transitions to the various cache states based on the transactions it sends and receives by using a built-in cache model. The user has back-door access (through API's in the cache model) to the cache model to allocate, de-allocate or query the cache lines. The Slave agent responds to read/write requests and models the memory for the system. It also supports ACE-Lite requirement through simple configuration parameters. The Interconnect Environment component receives coherent transactions from the initiating master, and generates appropriate snoop transactions to the other masters based on domain information. It then responds to the coherent transactions based on responses received through snoop transactions.

The Master and Slave agent instantiate the Port Monitor which continues to be available when the agents are configured in the passive mode. These monitors perform port-level transaction checks, signal stability checks and sequencing between ACE coherent and snoop transaction checks. Another key component of the ACE solution is the System Monitor. The System Monitor performs system-level checks, coherency checks and data integrity checks. As certain checks are dependent on design behavior, the system monitor also provides hooks to implement design-specific checks. The built-in coverage supports ACE coherent and snoop transaction coverage. The cache state transition coverage helps to validate whether the master’s cache has transitioned through all the legal cache states. The coverage can be used in conjunction with ACE Verification Planner to track the verification progress.

6. SOLVING CACHE COHERENCY VERIFICATION CHALLENGES

The verification strategy for a cache coherency system can be broken down into different stages. They would be :

- *Integration testing* : This would involve all the different VIP and RTL components are appropriately hooked up. Pre-defined sanity tests shipped with the VIP and the sequences shipped with the UVM register package can help in this regard.

- *Basic Testing:* Here stimulus generation is directed towards single interfaces. All different transaction types and the combination of all valid cache line states should be validated in this stage

- *Intermediate Testing:* This would involve specific scenarios involving multi-master communication. Some example sequences could be those involving overlapping ‘writes’ and ‘snoops’ during memory updates

- *Advanced Testing:* In this stage, system level stimulus mapped to different traffic profiles would be generated and all the checks enabled to verify the correct operation.

The following subsections capture how different challenges are addressed in the context of stimulus generation, system level checks, hierarchical phasing, coverage tracking and management and debug.

6.1 STIMULUS GENERATION

The VIP components are complemented by a library of configurable ACE sequences. These sequences weaved together form virtual sequences and sequencers further aids in scenario creation at the block, cluster or system level across various masters and the Interconnect. Additionally, the UVM sequence library enables the user to control the different permutations by which atomic and hierarchical sequences can be stitched together to create the complex scenarios depicted above.

Creating custom rules for the sequence library would not only help to streamline multiple sequences in different simulations but also to avoid redundancy and move progressively towards convergence of all interesting system-level scenarios. Again, in such scenarios, the sequences have to be aware of the functional configuration so as to be able to reconfigure itself based on the system-level requirements.

6.1.1 CREATING CONFIGURABLE SEQUENCES

There might be specific requirements when the sequences’ constraints or properties depend on the values in the configuration object. The UVM Resource mechanism is used in the AC sequences to bring in the configurability as shown below:

```
status = uvm_config_db#(bit)::get(m_sequencer, “” ,
                        "force_addr_gen", force_addr_gen);

`uvm_info("body", $sformatf("force_addr_gen is %0d as a
result of %0s.", force_addr_gen, status ? "config DB" :
"randomization"), UVM_LOW)

if (!status) begin
 if (my_basic_sequence != null)
    force_addr_gen = my_basic_sequence.force_addr_gen;
 end
…
 if(force_addr_gen) `uvm_do_on(…)
```

Figure 8: Configurable Sequence

Though the hierarchical UVM configuration mechanism is designed around components, the non-component object can access the configuration field through the component handle. In case of sequences, ‘m_sequencer’ is the handle to the sequencer that executes the sequence. It is a built-in member of the uvm_sequence class. The configuration parameter can be accessed in a hierarchical context through the ‘m_sequencer’ handle as shown below.

uvm_config_db#(int)::get(m_sequencer,“”, "item_count",item_count);

The ‘set’ of the parameter is as follows:

uvm_config_db#(int)::set(this"env.agent.seqr", "item_count", 20);

Therefore, when parameters change in a dynamic environment, the ACE sequences can reconfigure themselves to meet the generation requirements at that point in time. Thus, for different Master and Slave component which might support a subset or full ACE, ACE-Lite, AXI4 or AXI3 protocol and might work with different bus width or clock frequency, the sequences can be reconfigured to work with each of their associated sequencers.

6.1.2 HIERARCHICAL SEQUENCE STITCHING AND SEQUENCE LIBRARIES

The range of functionalities supported by the protocol ranges from ones that can be mapped to atomic transactions to ones which would run into hundreds of lines of testbench code. The sequence collection has a rich set of functionality. There are sequences to initiate all the possible coherent transactions, Sequences which do not cause a snoop of any cached masters, which must cause a snoop of the cached masters that can hold a copy of the cache line, which must cause a snoop of any of the cached masters that can hold a copy of the cache line and more.
Hence, given the functionality that UVM provides, it is much more convenient to stitch together complex scenarios from low level ones which has been proven or validated. This is how the ACE higher level and virtual sequences are built up and here we can see how custom user scenarios can be built using the sequence collection.

Let us take the case where we need to verify that all the cacheline states associated with a *Readclean* transaction needs to be tested. This would require a cacheline initialization followed by cacheline invalidation then a basic Readclean. A cacheline initialization sequence initializes the cacheline states of a master's cache and its peer's caches to a set of random but valid states. This ensures that all the different cacheline state transitions for a coherent transaction initiated by a master are verified. A cacheline invalidation sequence invalidates cachelines of a master. This may be required for load type of transactions which are not speculative. A basic *Readclean* sequences initiates a *Readclean* transactions over a given set of addresses. The basic steps for the same are:

1. Choose the set of addresses on which to test the sequence (user configurable)
2. Cacheline initialization - brings cachelines states to random, but valid states for all masters.
3. Cacheline invalidation - Load transactions may need to invalidate its cache before initiating transactions, unless they are speculative.
4. Basic *ReadClean* - Initiate a particular transaction type from one master.

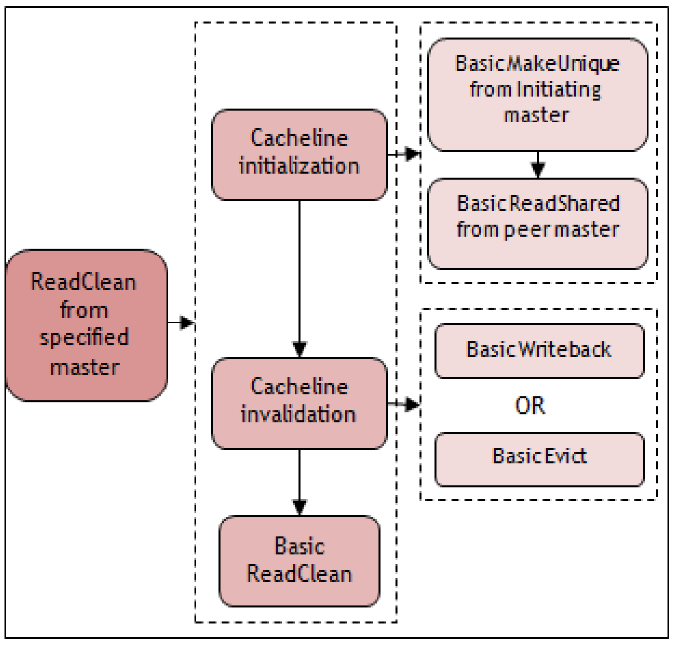

ReadClean from specified master
Cacheline initialization
Cacheline invalidation
Basic ReadClean
BasicMakeUnique from Initiating master
BasicReadShared from peer master
BasicWriteback
OR
BasicEvict

Figure 9: ReadClean Coherent Command – A basic flow

A complete verification scenario (like, shown in Fig. 9) can be mimicked using the nested sequences as explained in Figure 4. With the hierarchical approach, it becomes relatively easy to model any scenario generation requirements regardless of how complicated they are.
The same approach when combined with the virtual sequences helps to bring this functionality across multiple interfaces and is very relevant in the system context. For example, there are multiple virtual sequences that are part of the library and perform a combination of different sequential coherent transactions from different masters to the same slave.

```
// Write into M0's local cache. Data is
now dirty in local cache
M0 initiating MAKEUNIQUE to addr1

// Write data into memory. Data is now
clean in local cache. Data in cache
matches data in memory
M1 initiating WRITECLEAN to addr1

// Read data into M1's local cache. Gets
clean data from M0
M1 initiating READSHARED to addr1
```

Through the above sequence we test that the interconnect can do the following:

1. Initiate snoop transactions correctly
2. Fetch data from snooped masters and provide it to another master.
3. Interact with main memory correctly.

Apart from building the explicit virtual sequences, we can use the uvm_sequence_library to achieve the same. Here we can add the sequences registered with the sequence library on the per requirement basis for a specified instance of the sequencer. Thus, sequences modeling functionalities such as overlapping store operations to verify the interconnects behavior for concurrent transactions or those exercising multiple initiating masters attempt simultaneous shareable store operations to the same cache line can easily be made part of the sequence library or collection which can readily be leveraged by the end user.

6.2 USING THE AXI INTERCONNECT AND SYSTEM LEVEL CHECKS

The system monitor observes transactions across the ports of a single interconnect and performs checks between the transactions of these ports. It does not perform port level checks which are done by the checkers of each master/slave agent connected to a port. In ACE, the system monitor correlates coherent transactions and the corresponding snoop transactions to perform checks. The checks in the system monitor are geared towards checking the proper working of an interconnect DUT.

The system monitor requires transaction-level inputs from the master and slave ports that are connected to interconnect. By transaction level inputs, we mean transactions created by port level monitors as a result of signal level activity. The system monitor does not require signal level inputs. Transaction level inputs are provided by port monitors. In order to provide transaction level inputs, the system monitor could in turn instantiate port level monitors. UVM provides us with the capabilities to easily connect various components. All transactions from the port level monitors of each of the agents can easily be provided to the system monitor via TLM connections, thereby eliminating the need for instantiating these port level monitors in the system monitor.
Two examples for system level checks are,

Check	*Specification Ref.*	*Description*
coherent_resp_isshared_check	C6.4 Transaction responses from the interconnect 10. If WasUnique was not asserted for any snoop response received by the interconnect then, - If any snoop responses had IsShared asserted then, IsShared must be asserted in the transaction response to the initiating master	Checks that the IsShared response to initiating master is correct
snoop_resp_wasunique_check	C1.2.3 Cache State Rules: A line in a Unique State must be only in one cache	Checks that no two responses to a snoop transaction have the WasUnique(CRRESP[4]) bit asserted.

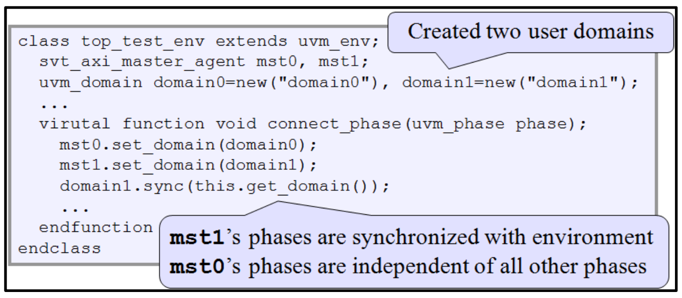
Figure 9: System Checks

Thus again by leveraging the UVM capabilities coupled with knowledge on the coherency the system check provides robustness to the DUT verification.

6.3 DISTRIBUTED PHASING

Finally, given the usage of such coherent systems in all handheld devices, it is imperative to come up with a mechanism to have a power aware verification setup. Also, as mentioned earlier, different components might support a different subset of the protocol. Some of the components might be power aware and would be modeling components

in power domains. Such components would need the phase aware sequences to be executing in user defined phases. Some of these might go to a powered down phase in the middle of simulation and on 'waking up' would have to catch up the other phases. Again, the UVM hierarchical phasing schemes and configurable sequences can be leveraged to help the user to model the different power state transitions for the system.
UVM allows new *domain's* to be created and components can be grouped into different domains which have executed their phases independent of each other. The default domain name is the 'uvm' domain which contains the default runtime phases.

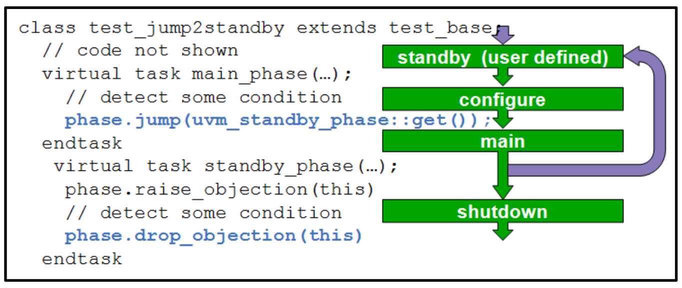


```
class top_test_env extends uvm_env;
  svt_axi_master_agent mst0, mst1;
  uvm_domain domain0=new("domain0"), domain1=new("domain1");
  ...
  virutal function void connect_phase(uvm_phase phase);
    mst0.set_domain(domain0);
    mst1.set_domain(domain1);
    domain1.sync(this.get_domain());
    ...
  endfunction
endclass
```

Created two user domains
mst1's phases are synchronized with environment
mst0's phases are independent of all other phases

Figure 9: Distributed phase synchronization

New phases can be inserted to the domains created. The components in a specific user-defined domain can be made to come in sync with the other domain at the end of run_phase. So, even if an ACE component is powered down, it alone can be made to rewind back to an earlier phase, wake-up and then get in phase with the other components running the default runtime phases.



```
class test_jump2standby extends test_base;
  // code not shown
  virtual task main_phase(…);
    // detect some condition
    phase.jump(uvm_standby_phase::get());
  endtask
   virtual task standby_phase(…);
    phase.raise_objection(this)
    // detect some condition
    phase.drop_objection(this)
  endtask
```

Figure 10: UVM phasing – jump-back

6.4. Coverage Tracking and Management

It is important to track the diverse permutation and combinations of verification scenarios for a Cache Coherent System. An executable verification plan can help significantly in this regard. This verification plan has to be hierarchical with sub-plans based on the different ACE components. UVM Resource Mechanism can be used to ensure that the functional coverage model can be configured based on the system configuration. Coverage Model is thus 'configuration aware' which means bins are ignored if they are not applicable to the VIP configuration. Users can extend the built in coverage to add their own bins based on built in VIP sampling events and groups or create their own groups with any sampling event or data. The configurability in transaction, configuration and scenario coverage would ensure that wasteful debug cycles are not spent trying to analyze coverage holes which do not have relevance in a specific context.. When the VIP provided executable plan is used, functional coverage results can be back-annotated to the plan demonstrating coverage that was achieved as mapped to the protocol specification and for a specific Cache Coherent System configuration. The executable plan provided through the Verification Planner has the skip' feature to customize the VIP Verification Plan or simply not import a subplan that isn't needed based on the UVM Configuration.

6.5. UVM ENABLED DEBUG OF A CACHE COHERENT ARCHITECTURE

The complexity of modern protocols also creates a significant challenge for many "traditional" debug methodologies. Some methodologies utilize a "bottom-up" approach, extracting information from simulation log or waveform dump files and then attempting to transform this low-level data into more easily analyzed higher-level representations. However, this approach assumes that all of the required high-level information can be inferred from the low-level data. Top-down methodologies which focuses on higher layer transactions rarely offer adequate insight into the underlying simulation details, which are often required to uncover the ultimate cause of unexpected behavior. This illustrates an important requirement for a protocol-oriented analysis tool: specifically, the ability to represent and view protocol data at multiple levels of abstraction.

How can an UVM based environment help here? The requirements are to provide a protocol-oriented analysis environment which should provide visualization and analysis of *"protocol objects"*. A "protocol object" would be any description of data that is found in a protocol specification. To ensure that there are enough configurable hooks in the VIP, the component hierarchy is interspersed with UVM callbacks at all interesting execution points. These callbacks can be leveraged to dump an XML trace of the simulation. As these points are all 'protocol aware' and the UVM testbench is aware of the levels of abstraction at these different points, the appropriate information can be dumped into the XML traced as mentioned earlier. Thus a tool like the Protocol Analyzer can efficiently analyze this information and provide an Interactive frontend to debug protocol behavior The three different displays that are available for visualizing protocol objects each feed on the underlying methodology layer to provide a unique set of analysis capabilities . For example, the Object Timeline Display emphasizes the temporal relationship between the protocol objects in the view, the Object Tree/Table Display emphasizes the hierarchical relationship between objects and the Object Tree. Display focuses on the field attributes and values of the protocol objects.

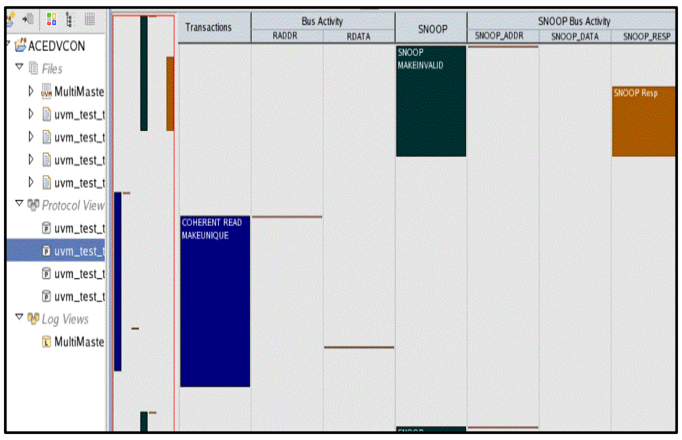


Figure 11: GUI plot of AXI-ACE transactions and VIP log messages

Additionally, the underlying UVM base classes can be easily leveraged to dump all the transactions and the transformations and therefore debugging them with the signals from the HDL testbench.

7. CONCLUSION

As the complexity of protocols continues to increase and evolve, the infrastructure required for the verification of the same needs to scale up in sophistication as well. Methodologies such as, UVM and VMM has been undergoing continuous evolution to keep up with the many complex requirements. The new AMBA 4 Coherency

Extensions (ACE) comes in at this critical juncture to help meet the need for higher processing capabilities under optimal power consumption modes. Verification of basic cache coherent systems is in itself challenging. The additional complexity which the ACE protocol brings demands a lot from the verification methodology used. The recent updates to the UVM library with respect to sequence generation, distributed phasing, configuration management has come as a boon to meet all these requirements.

8. APPENDIX

"Speculative fetch" - The interconnect initiates a transaction to the slave memory even before it gets a response to snoop transaction it initiated to the masters. This is done to improve the latency for fetching data.
"Cache miss fetch" - None of the masters' cache has data and so the interconnect needs to fetch from the memory.
"Pass Dirty propagating" - Certain coherent transactions cannot accept a dirty response. Snooped masters can however pass dirty data. The interconnect might have to write this dirty data to the memory before passing a clean response to the master that initiated the transaction.

9. ACKNOWLEGEMENT

We would like to extend our sincere thanks to Abhijeet Khopkar, Manager R&D, Verification Group, Synopsys, for reviewing the paper and providing his valuable insights.

10. REFERENCES

[1] UVM User Guide
[2] UVM Reference Guide
[3] Synopsys® Verification IP (VIP) for AMBA AXI User Guide
[4] Synopsys Protocol Analyzer Getting Started Guide
[5] Synopsys UVM CES Training
[6] AMBA AXI and ACE Protocol Specification - AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite (Non-Confidential - Draft – Beta) Document Number: ARM IHI 0022D-2c ID060311