cādence®

Modeling and Verifying Cache-Coherent Protocols, VIP, and Designs

Cadence® Jasper® ActiveModel technology provides a novel method for modeling and verifying cache-coherent protocols, such as the ARM® AMBA® AXI Coherency Extensions (ACE) protocol. Using this technology, Jasper Design Automation (now a part of Cadence) and ARM successfully collaborated on a project that developed system- and interface-level verification IP (VIP). In the course of this collaborative project, the team devised a flow that included verifying RTL designs. In addition, it demonstrated how the technology is used to support in-depth exploration of the ACE protocol to aid comprehension and ensure protocol adherence.

Contents

Introduction................................1

Cache-Coherent Protocol Specifications..............................2

Cache-Coherent Protocol Verification 5

Cache-Coherent Protocol Verification of RTL..................... 11

Summary 13

Introduction

Cache-coherence management is about maintaining consistency between caches and memory. As illustrated in Figure 1, if agent A1 has a copy of data from memory location X and agent A2 modifies location X without notifying A1, the A1 cache will be stale.

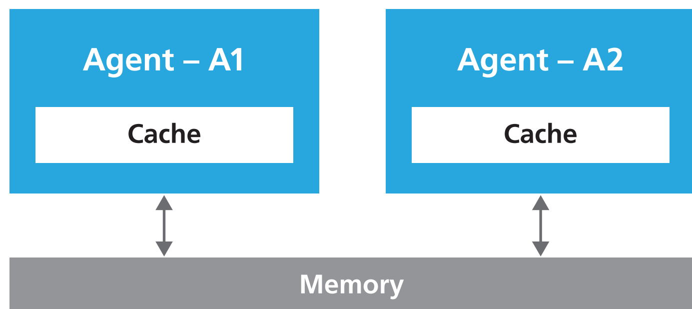

Figure 1: Cache coherence management

One way to manage cache coherence is to use software, but the resulting performance is typically inadequate for high-performance systems. This performance challenge becomes even greater as systems get larger. Hardware-based cache-coherence protocols provide superior performance and are common in many computing systems, such as servers. With the increasing complexity of today's SoCs and their performance requirements, hardware-based cache-coherence protocols have become necessary in that realm as well. The ACE protocol provides an excellent example of this type of protocol for SoCs and is referenced throughout this paper.

At its core, cache coherence means the following:

- A read to a coherent memory location reads the data that was last written to the same memory location. For example, a read to location X that follows a write to location X with value D must read the value D provided that no other writes to location X occur before the read.
- Writes to the same coherent memory location are observed in the same order by all components in the system. For example, if memory location X has been written to with values D1 and then D2 by one or more agents, then no agent must ever observe D2 and then D1.

This definition is somewhat ambiguous since reads and writes are not instantaneous in real computer systems. Therefore, cache-coherent protocol specifications define more precisely when writes must be visible.

Cache-Coherent Protocol Specifications

Cache-coherent protocol specifications generally define and describe several key elements of the underlying protocol, including system components, granularity of coherency (the size of a cache line), the access rights that agents have to cache lines (cache-line states), transactions that operate on cache lines, channels or networks the transactions travel on, the effect of the transactions on cache-line states, and a description of an illustrative subset of the transaction flows in the system.

System components

As distinct agents of a system, protocol components initiate or react to the protocol transactions and maintain the necessary state, for example, the cache-line states. In the ACE protocol, the main components are masters, slaves, and interconnects. The masters are the components that initiate requests and often contain a cache (also known as a cached master). The slaves are the components that respond to requests, for example, a memory. An interconnect connects one or more masters to one or more slaves. An interconnect also initiates transactions to take away cache-line access rights (also known as snoops).

Granularity of coherency

Protocols define the granularity of coherency to be some region of memory. This region of memory is called a cache line or simply just a line. For example, a cache line might be defined as 128 bytes of contiguous address space. Transactions most often reference the entire cache line. For example, for a system with 128-byte cache lines, a read transaction will retrieve all 128 bytes of data even though the internal operation that caused the read transaction may only need 8 bytes of that data.

Access rights

Protocols define access rights that a master will have on a cache line. For example, the access rights typically describe:

- Whether a component can modify the line
- Whether the line is in other caches
- Whether the line has to be written back to memory

The ACE protocol defines five states for each cache line in a cache:

- Invalid (I)
  - The cache line is not in this cache
- UniqueClean (UC)
  - The cache line is only in this cache
  - The cache line has not been modified with respect to memory
  - The cache line can be written without notifying other caches
- UniqueDirty (UD)
  - The cache line is only in this cache
  - The cache line has been modified with respect to memory and this component is responsible for eventually writing it back to memory
  - The cache line can be written without notifying other caches

- SharedClean (SC)
  - The cache line may be shared in other caches
  - The cache may or may not be modified with respect to memory, but this component has no write-back responsibility
  - The cache line cannot be written without notifying other caches
- SharedDirty (SD)
  - The cache line may be shared in other caches
  - The cache line has been modified with respect to memory, and this component is responsible for eventually writing it back to memory
  - The cache line cannot be written without notifying other caches

It should be noted that it is not essential for the caching components to support all cache-line states defined by the protocol. The determination of which cache-line states to support often depends on the capabilities of the component. For instance, in the ACE protocol, a component that will only read but never write a cache line might only support the SC and I states.

Protocol transactions

Protocols also define the transactions that operate on the cache lines. The set of transactions includes those initiated to gain access rights to a cache line, to give up access rights to a cache line, and to take away access rights to a cache line.

Gain Access Rights – When a master needs to perform a load operation, it must gain at least Shared access rights to a cache line. However, other caches can be allowed to retain a Shared copy of the line. In this case, a master may issue a ReadShared transaction (there are other possibilities too).

When a master needs to perform a store operation, it must gain Unique access rights to a cache line. Since the master is going to write the line, other caches are not allowed to retain a copy of the line. In this case, a master may issue a ReadUnique transaction (there are other possibilities too).

Give Up Access Rights – When a master needs to free up space in its cache and the cache line being flushed is not in a Dirty state, it can simply drop the cache line or it can issue an Evict transaction. When a master needs to free up space in its cache and the cache line being flushed is in a Dirty state, the master is responsible for updating memory with the dirty data. In this case, a master may issue a WriteBack transaction (there are other possibilities, too).

Take Away Access Rights – Access rights may need to be taken away from one component when another component is trying to gain access rights. This process is often called snooping, and it is performed using snoop transactions. For example, if a master has a SharedClean copy of a line and another master makes a ReadUnique request, the SharedClean copy must go to Invalid. The interconnect in this case would react to the ReadUnique request and issue a ReadUnique snoop transaction to the master with the SharedClean copy. When the snoop is complete, the master would be invalidated and the interconnect could give Unique access to the requesting master.

Enough types of transactions need to be available for components that do not support all cache-line states. For example, in the ACE protocol, if a master that only supports the SC and I states wants to read a cache line, it must issue a read transaction indicating that it can never be given Dirty access rights to the cache line. This transaction is a ReadClean. Another master that supports all cache-line states could issue a less restrictive read allowing for receiving any of the cache-line states. In the ACE protocol, this transaction is a ReadShared.

In general, transactions go through several phases as described by the protocol. These phases are completed by components of the system communicating with each other through the channels or networks described in the protocol.

For example, in the ACE protocol, a ReadShared transaction is first issued on the read address channel (AR), and at this point, it is in the address phase. Then, the transaction is accepted by the interconnect, which may decide to issue snoop transactions on its behalf to other masters on the snoop address channels (AC). The ReadShared transaction is now waiting for the snoop transactions to complete, and this is called the snoop phase. The masters process and complete the snoop transactions on the snoop response channel (CR), and if data is found in the master's cache, the masters complete the transactions on the snoop data channel (CD). Once the interconnect has collected the responses, the original transaction moves on to the response phase. In the response phase, data is transferred to the original requesting master on the read data channel (R). After receiving all the data, the master sends a read acknowledge (RACK), which completes the transaction.

It should be noted that transactions often do not complete in the same order that they were initiated. A transaction can wait in a phase for an extended period of time. For example, it might wait in the snoop phase while snoops complete. Or it might wait in the response phase to receive data from memory. During these stalls, other transactions might have collected their necessary information to allow them to move on to the next phase.

Protocol channels

Protocols also define the channels or networks on which transactions travel. Distinct protocol channels are necessary for several reasons. They provide the signaling that is necessary to convey the information from one component to another. They can provide the distinct rules for transactions traveling on the channel (for example, whether a transaction can pass another transaction on the channel), and they can provide independent flow control for transactions traveling on the channel. The ACE protocol provides 10 channels:

- Read address channel (AR) – Carries read transactions from a master to an interconnect or an interconnect to a slave
- Read data channel (R) – Carries read data from an interconnect to a master or a slave to an interconnect
- Read acknowledge channel (RACK) – Carries a read acknowledge signal from a master to a interconnect
- Write address channel (AW) – Carries write transactions from a master to an interconnect or an interconnect to a slave
- Write data channel (W) – Carries write data from a master to an interconnect or an interconnect to a slave
- Write response channel (B) – Carries write responses from an interconnect to a master or a slave to an interconnect
- Write acknowledge channel (WACK) – Carries a write acknowledge signal from a master to an interconnect
- Snoop address channel (AC) – Carries snoop transactions from an interconnect to a master
- Snoop response channel (CR) – Carries snoop responses from a master to an interconnect
- Snoop data channel (CD) – Carries snoop data from a master to an interconnect

Cache-line state updates

One of the most critical elements of a protocol specification is a description of how transactions impact cache-line states. This information is often best conveyed in a table format. Figure 2 shows an example generic action table. The tables generally show the possible current states of a cache line along with the possible transactions operating on the cache lines (green). For each distinct combination of current state and operating transaction, next-state values and/or other actions are shown (yellow). Row 0 shows an RdC (read code) snoop transaction hitting an M (modified) cache-line state. The next state of the cache line goes to S (shared), and the snoop response is supposed to indicate that the cache will be keeping a copy of the line and that the response will include a data transfer.

	Snoop Txn	Current Cache State	Next Cache State	Cache Kept Copy	Rsp Has Data
0	RdC	M	S	1	1
1		E,S	S	1	0
2		I		0	0
3	RdD	M	I	0	1
4	Wr	M	I	0	1
5	RdD,Wr	E,S,I	I	0	0

Figure 2: Generic action table

Tables indicating the allowable transactions, cache-line state transitions, other transient states, and subsequent actions make up the heart of a cache-coherent protocol specification. And a single specification can warrant as few as 10 tables or in excess of 100 tables to adequately capture how transactions impact cache-line states. The tables are often the key pieces of reference material implementers and verifiers use to understand the protocol and how its components are intended to operate.

Transaction flows

Pieces of a protocol specification that are especially important to aid in understanding are the descriptions and drawings of some of the basic transaction flows in the protocol. Because of the large number of possibilities, only the most frequently used or most informative transaction flows are highlighted in these descriptions and drawings. Figure 3 shows one example of a transaction flow diagram.

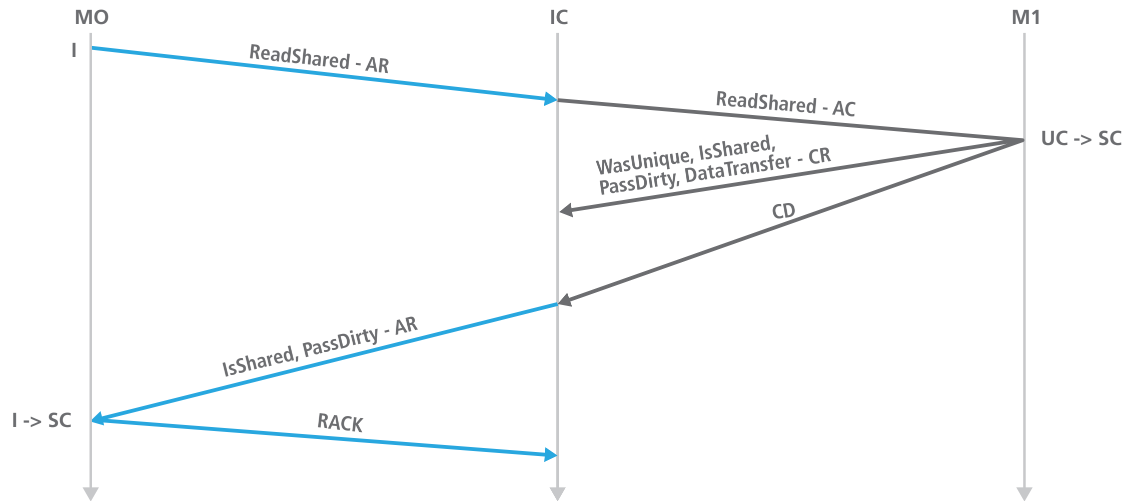

Figure 3: Transaction flow diagram

In the flow above, a master, M0, issues a ReadShared transaction on the AR channel because it does not have the cache line (Invalid state). The interconnect, IC, is connected to one other master, M1, and issues a ReadShared snoop transaction on the AC channel. M1 had the cache line in the UC (UniqueClean) state, so it moves to the SC (SharedClean) state and responds on the CR channel indicating that it was in a Unique state, is still in a shared state, is not passing dirty data, but is passing data. The master also transfers the associated data on the CD channel. Once the IC collects the snoop response, it issues a read data response to M0 on the R channel. It indicates that another master still may have the cache line in a shared state and it is not passing dirty data. Also on the R channel is the associated data. Once the response is received, M0 moves to the SC state. Also, when the data is received, M0 acknowledges the response with a RACK.

One thing that becomes very clear from transaction flow drawings like Figure 3 is that there are potentially many what-if questions that could be asked and one-off flows that would be interesting to view. For example, imagine if both masters were requesting the same line at the same time or if the CD snoop data arrived at the IC prior to the CR snoop response. Also, what would happen if the line was UD in M1 or if the original transaction was a ReadClean instead of a ReadShared? In addition, it is unclear whether other cache-line state transitions are possible.

Cache-Coherent Protocol Verification

After the specification of the protocol is created, an equally challenging task is to sufficiently verify the protocol. Often, considerable energy goes into the verification of cache-coherent protocols with several forms of verification.

Documenting the protocol is the first form of verification and is effective because it forces the specification writer to carefully consider different aspects of the protocol. Contemplating the flows often reveals things that need to be handled differently by the protocol. Also, creating action tables can highlight holes in the protocol and uncover ambiguities.

Peer reviews are also very common and naturally occur since there are so many consumers of the specification. The peer review process often leads to interesting whiteboard discussions that promote clarification from the architect. The architect then incorporates the necessary updates into the specification to remove ambiguities.

The final level of verification is to build a model of the protocol, create high-level properties describing cache coherence, and either run simulations against the properties or attempt to prove the properties with a formal verification tool. There are many challenges inherent in this level of verification, including creating a machine-readable specification, ensuring protocol model soundness, debugging the protocol model, verifying the protocol model, using the model for protocol comprehension, and creating a reusable protocol model for RTL verification.

Creating a machine-readable specification

A machine-readable, cache-coherent specification can be created by choosing a modeling language (for example, Murphi, TLA, Verilog), and then going through the specification, creating the stated rules, capturing the necessary states, and coding the necessary state transitions at the specified moments. However, this process has its shortcomings for several reasons.

The language chosen is generally unlike anything in the specification; for example, the specification is not likely to define variables or represent the rules in explicit if-then-else statements. Instead, there are words, tables, and drawings. Thus, the translation itself will likely lead to some errors. Also, because of the necessary translation, it may be difficult to adapt the model when the specification is changed or enhanced. Further, modeling languages generally do not have good methods for capturing the clarifying comments that a specification might have, nor are they able to attach the comments to the rules or actions of the protocol.

Instead, as has been discussed, specifications are heavily table based. These tables concisely describe the legal input activity when the protocol is in a particular state and then describe the actions to take. A table-based specification also tends to more clearly give a big-picture view of the operations. Therefore, to ease the burden of modeling, a table-based input format that is machine readable is often desired.

A model for the ACE protocol was created using the ActiveModel table-based entry format. As seen in Figure 4, the format is similar to the entry format used by many cache-coherent protocol specifications where the current state and input activity are shown in columns on the left side of the table (green) and the next-state transitions and other activities are shown in columns on the right side of the table (yellow). The rows of the tables represent the various possible combinations of input activity and current state followed by the appropriate actions for each combination. This format is not only similar to the tables that architects write, it can be the actual tables the architects write, which eliminates one potential source of errors.

	A	B	C	D	E	F	G	H	I	J
228										
229		TABLE=IC_CR		PRLL	FULL	COVER (ICCOVER)				
230		LABEL	IN	IN	NEXT	IN [MS]	IN [MS]	IN [MS]	WIRE [MS]	WIRE [MS] [CSW]
231		FOR ms[MS]	ic_crvalid [ms]	snp_ac_type [ms] [ic_cr_tid[ms]]	dirs [ms] [snp_ac_addr [ms] [ic_cr_tid[ms]]]	ic_cr_PassDirty [ms]	ic_cr_IsShared [ms]	ic_cr_DataTransfer [ms]	ic_cr_dirs_ue [ms]	ic_cr_dirs [ms]
232		0	0	-	-	-	-	-		
233		1	1	-	I	-	-	-		
234		2		ReadOnce	UC	-	0	-	1	I
235		3				0	1	-	1	UC
236		4				1	1	-	1	SC
237		5			UD	1	0	-	1	I
238		6				0	1	-	1	UD
239		7				1	1	-	1	SC
240		8			SC	-	0	-	1	I
241		9				-	1	-	1	SC
242		10			SD	1	0	-	1	I
243		11				1	1	-	1	SC
244		[illegible]				0	1	-	1	SD
245		13		ReadClean, adNotSharedDirty, ReadShared	UC	-	0	-	1	I
246		14				1	1	-	1	SC
247		15				0	1	-	1	SC
248	#	16						1	1	SD
249		16			UD	1	0	-	1	I
250		17					1	-	1	SC
251		18				0	1	-	1	SD
252		19			SC	-	0	-	1	I
253		20				-	1	-	1	SC
254		21			SD	1	0	-	1	I
255		22					1	-	1	SC
256		23				0	1	-	1	SD
257		24		CleanShared	UC	-	0	-	1	I
258		25				1	1	-	1	SC
259		26				0	1	-	1	UC
260		27			UD,SC,SD	-	0	-	1	I
261		28				-	1	-	1	SC
262		29		ReadUnique, CleanInvalid, MakeInvalid	UD,UC,SD,SC	-	-	-	1	I
263										

Read*Shared/Clean snoops UC and goes to SC.

interface / aw / w / b / wack / ar / r / rack / ac / crcd / misc / cchs_dirs / barriers / DVM / properties / deadlock

Cell A245 commented by Ross Weber

NUM

Figure 4: ActiveModel table-based entry format

Ensuring protocol model soundness

It is imperative to question whether the protocol model you created is sound. For example, if the properties you are verifying are not failing, how do you know that the model can get into all the possible states that should be explored? In a typical coverage-driven verification flow of an RTL design, coverage points ensure that the input stimulus can get the design into all the interesting states. Similarly, for a protocol model, the model itself should be instrumented with coverage points to ensure that interesting states are being explored.

Recall that the rows of a generic action table represent the various possible combinations of input activity and current state followed by the appropriate actions for each combination. ActiveModel technology, in combination with these tables, removes the burden of manually creating cover properties by automatically generating covers for every row of every table. These row covers can then be used to determine whether the specified row can actually be reached. If a row cover cannot be reached, the protocol may be over-specifying some cases that are not possible. Removing these over-specified cases adds clarity to the specification.

ActiveModel technology also automatically generates a full check. The full check ensures that at least one row of the table is always being referenced. This check identifies cases where the specification does not address the actions necessary for all the possible states the protocol can get into. These problems are considered holes in the specification and indicate that the specification must be re-examined and modified accordingly.

ActiveModel tables can also automatically generate parallel checks. The parallel checks ensure that each row pair combination in the table does not overlap. An overlap is typically just a coding problem with the table, but eliminating the overlaps is essential for creating an unambiguous specification.

Debugging the protocol model

Debugging a cache-coherent protocol model typically is not much different than debugging any RTL-based design. A checker fails, a waveform showing the failure is produced, and a normal flow of tracing back from the failure ensues and continues until the bug is found. This process is usually adequate, but it has room for improvement. For example, to understand a failure trace from a protocol model, it is often beneficial to have a high-level view of what is occurring. In this situation, a link back from the property failure to the relevant tables in the specification is helpful. ActiveModel technology performs this link from a failing property to the appropriate rows in the table (Figure 5). With the table in view, a better, high-level grasp of what is occurring (or what is not occurring) can be achieved.

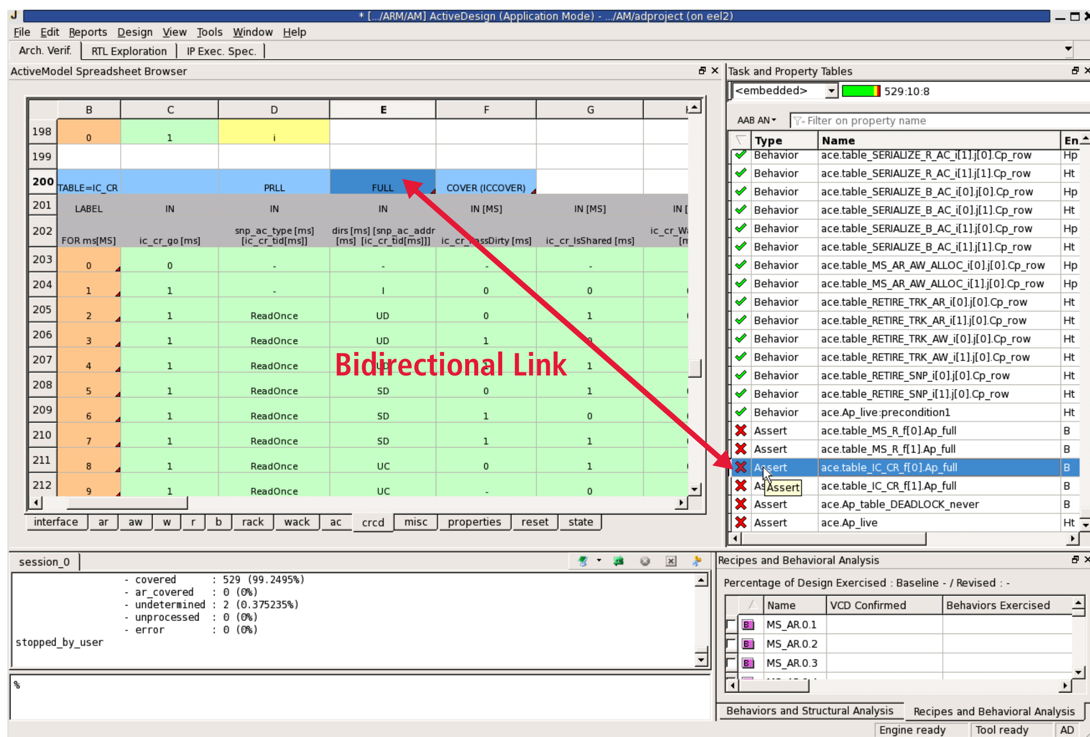

Figure 5: Linked views for debugging

Another debugging difficulty with protocol model failures is the fact that often many transactions are occurring in parallel in order to create the appropriate sequence that violates the check. Ideally, if the unimportant transactions can be eliminated from the waveform showing the violation, the violation can be debugged more quickly. This technology, appropriately called Jasper QuietTrace™ technology, is available when using ActiveModel technology.

Figure 6 shows two waveforms, each showing all the row covers that were active in the violation trace. The first is without the waveform being quieted, and shows 50 firings of the row covers. The second is the equivalent trace obtained by QuietTrace technology. It shows only 13 firings of the covers, all of which are likely contributing to the violation, thus drastically improving debug efficiency.

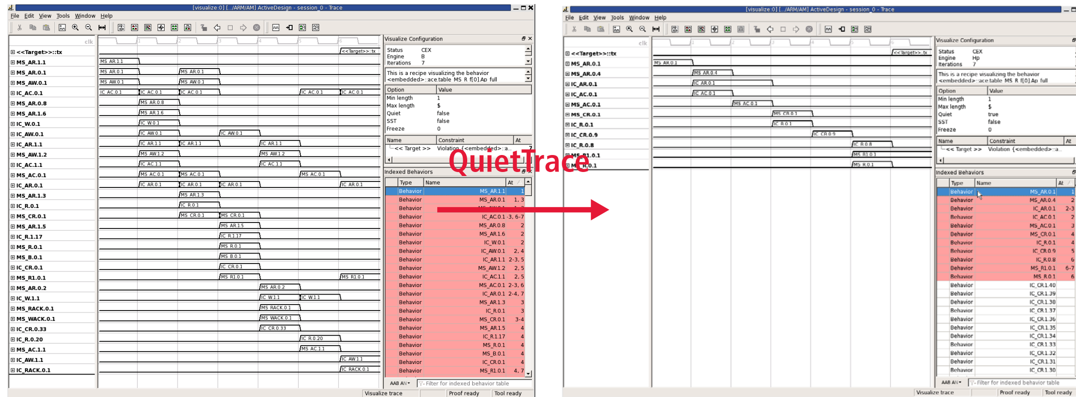

Figure 6: QuietTrace technology

Verifying the protocol model

A cache-coherent protocol specification describes the rules that, when followed, keep the caches in the system coherent. These rules are the assumptions made about the operations of the components of the system. To confirm that adhering to these rules does indeed keep the caches coherent, you need high-level properties that define coherency. Recall that high-level cache coherence means a read to a coherent memory location reads the data that was last written to the same memory location, provided no other writes occur. Also, all agents must observe writes to a coherent memory location in the same order. These statements can be constructed as high-level properties (assertions) and then verified. In addition, there are common invariants that can be expressed about the caches in a system. For example, according to the ACE protocol, two caches cannot have the same cache line in a Dirty state, and two caches cannot have the same cache line in a Unique state.

A protocol model verified using simulation alone can only find bugs. A protocol model verified using formal techniques can find all those same bugs and prove the high-level properties. Protocol models are generally smaller than most RTL designs being formally verified and they are usually better understood (since there is a specification). These factors make protocol models an ideal fit for formal verification.

The ACE/ActiveModel-based protocol model was formally verified. The formal verification process proved to be effective at both clarifying the specification and ensuring the correctness of the protocol model, which was later used to verify RTL. Consumers of the protocol can now use the model to aid in protocol comprehension.

Using the model for protocol comprehension

Another goal when creating an executable protocol model is to use the model to help consumers of the protocol, such as RTL designers and verifiers, quickly comprehend the protocol. The protocol specification, although rich with valuable information, is a static document. The transaction flows described or drawn in this document represent a small subset of the allowable transaction flows in a system. Consumers of the specification regularly wonder about some variations of the described transaction flows or what-if scenarios when viewing transaction flow drawings. The answers are probably somewhere in the words of the specification; whereas, the waveforms, drawings, and transaction flow descriptions are concise representations that allow for quick comprehension, but lack the depth of the specification's text. The protocol model offers a way to arrive at the answers in a meaningful way that does not require the user to mine the specification's text to find the relevant nuggets.

To generate the desired transaction flow, a user typically has a target in mind. For example, a user trying to understand the ACE protocol may want to see a case where a ReadShared snoop hits a UniqueClean line. In a simulation model, this flow may be difficult to generate. The user would need to know the sequence of inputs needed to hit

the condition or the user could write a coverage point and try to let the coverage point be hit through a random simulation. Both of these options are too time consuming to be practical. Thus, simulation models are rarely used for protocol comprehension.

On the other hand, this use model (visualizing cases to build comprehension) is a perfect application for the ACE/ActiveModel technology. A user can simply go to the ActiveModel table showing a master reacting to snoops and ask the tool to show a scenario for a row where a ReadShared snoop hits a UniqueClean line. In Figure 7 below, we choose to visualize a row indicating that a ReadShared snoop is hitting a UC line. The tool quickly generates a nine-cycle waveform that shows the desired flow (Figure 8).

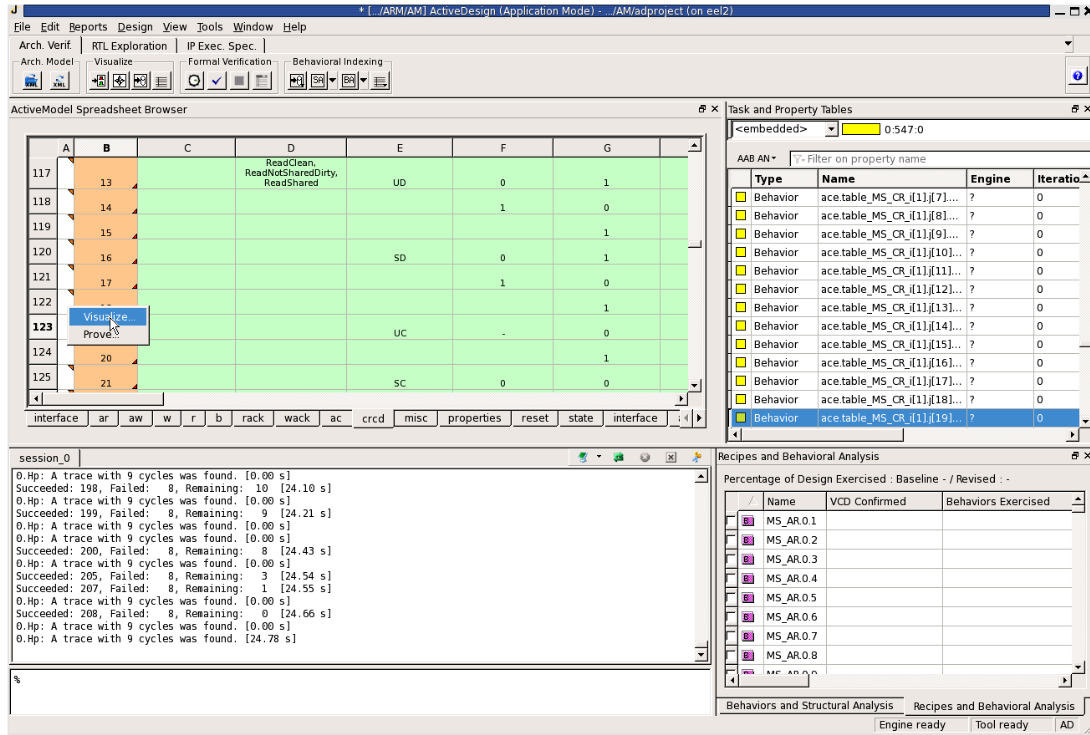

Figure 7: Visualizing the protocol

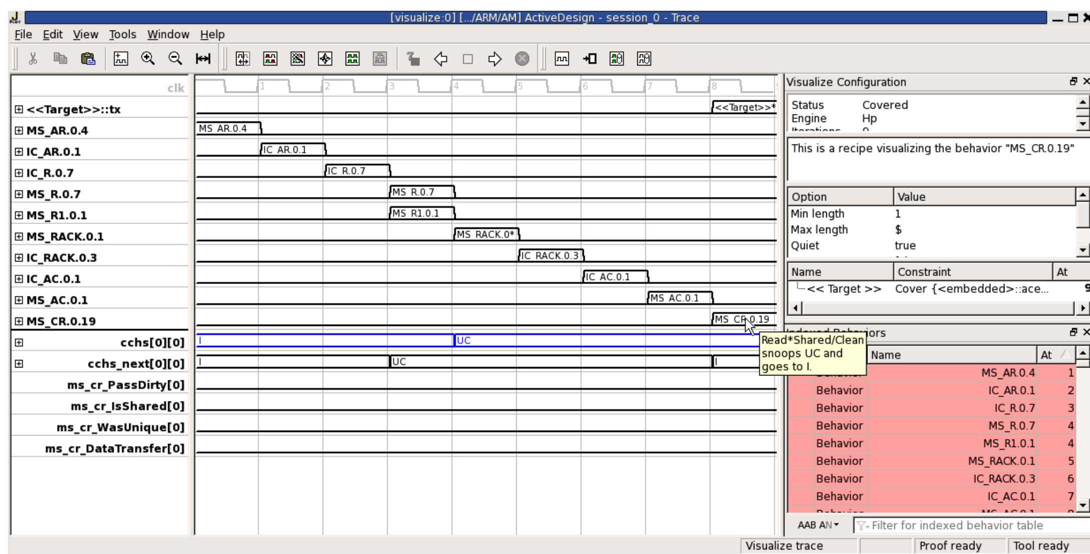

Figure 8: Desired flow waveform trace

Notice that in Figure 8 the row covers hit along the way are displayed in the waveform. In addition, a tool tip appears when we use the mouse to hover over any of the row covers on the cycle it was hit. The automatically generated tool tip text gives a short description of what is occurring at that time. This description came directly from the comments entered into the spreadsheet next to each row.

The initial waveform produced is often just a starting point. Once the user understands the waveform, further questions may arise. For example, in the above scenario it might be interesting to know whether WasUnique and DataTransfer can be active on the snoop response. To ask this question, a user simply adds another constraint to the waveform by 1) selecting the necessary signals, 2) clicking an “Add Constraint” button, 3) modifying the expression to indicate the desired values, and 4) replotting the waveform. Figure 9 shows two steps of this process. The upper waveform shows the “Add Constraint” dialog where the default expression can be modified and the lower waveform shows the results after replotting the waveform. The replotted waveform confirms that WasUnique and DataTransfer can be active on the snoop response.

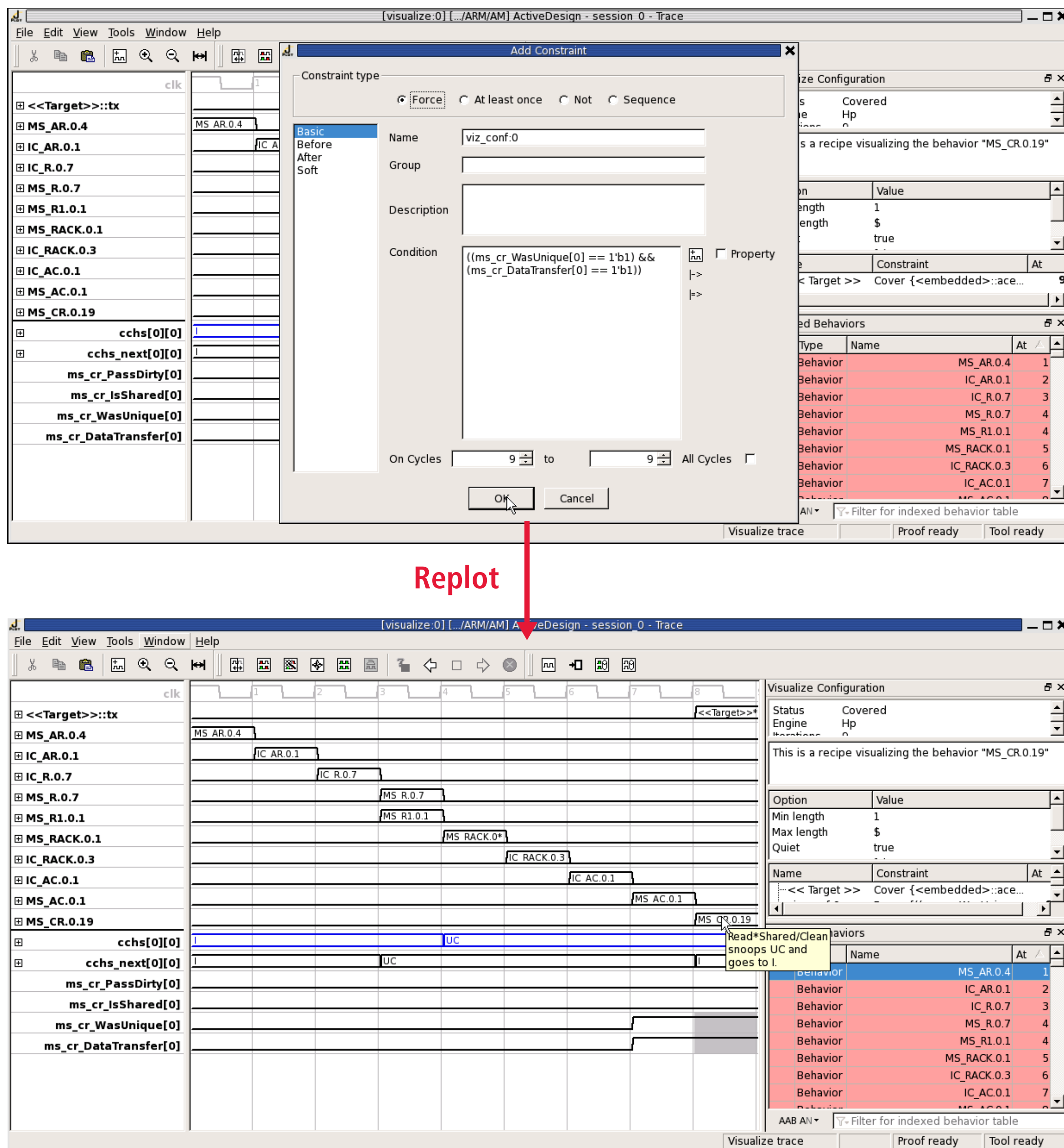

Figure 9: Adding a constraint

Also, as mentioned in the section on debugging a protocol model, the ActiveModel technology provides the QuietTrace feature. This feature modifies the generated waveform to reduce the irrelevant activity and make it easier to comprehend the target activity.

The added value of a protocol model that is not only verifiable but can be used to aid in comprehension of the protocol itself is enormous since there are so many consumers of the protocol specification. In addition, having a tool that provides the advanced capabilities to interact with the model speeds comprehension considerably.

Creating a reusable protocol model for RTL verification

As previously shown, a protocol model can be used to verify the protocol and to aid in protocol comprehension, and therefore can be used to verify RTL. The challenge is in trying to map the abstract protocol model states and transactions to the corresponding RTL states and transactions. Not only is the mapping difficult, the RTL is constantly changing and, thus, potentially breaking the map.

An alternative is to create a protocol model where all the protocol rules are written at the interface level. Then, the RTL can connect directly to the protocol model without any mapping needed. However, there are drawbacks to this method. For one, the model is more difficult to write because it needs to be less abstract and must deal with some of the bit-level activity at the interfaces. Also, an interface-based model is not making its relevant state transitions at the same time as the RTL. The state transitions within the RTL model are happening based on the ordering points internal to the design, for example, at the end of the L2 cache pipeline. Therefore, the interface-based model's states have to be slightly abstracted versions of the actual cache states in order to handle cases where the interface-based model cannot determine which exact ordering a design will choose to handle transactions.

The ACE/ActiveModel technology is interface based, and thus is capable of being used to verify RTL by connecting to the ACE interfaces of a system. The rules of the ACE protocol were used as assumptions when the protocol model was verified against high-level properties about cache coherence, and these protocol rules are used as assertions/checks when the model is attached to RTL, thereby implying that the assertions/checks are verifying that the RTL is cache coherent. In addition, the row covers generated by the ActiveModel technology are a high-quality set of properties that can be used to check the quality of the verification environment.

Cache-Coherent Protocol Verification of RTL

When verifying RTL designs whose protocol does not support cache coherence, it is sufficient to create VIP that monitors only one protocol interface and to verify the design against the VIP. The single-interface VIP tracks enough states to ensure that transactions in flight are operating correctly. For example, a single-interface VIP tracks all the reads that are outstanding and ensures that all read responses are only for outstanding reads. The single-interface VIP also monitors, in detail, all the bit-level activity occurring on the interface and ensures that it adheres to the specification.

However, when verifying designs whose protocol supports cache coherence, not only does the VIP need to continue to monitor the single-interface activity, it must also monitor system-level activity. For example, if two agents in a system both request access rights to a cache line that will allow the agents to modify the data, then the ordering agent in the system must not give both agents access rights at the same time. In order for the VIP to check this requirement, it must track the access rights given to both agents. Thus, for completeness, the VIP can no longer just monitor a single interface in the system. A system-level piece of VIP is needed.

As has been discussed, the components of the protocol may only support a subset of the available cache-line states. Thus, there are a variety of transactions that different components might issue for what is essentially the same internal operation. An earlier example mentioned that, in the ACE protocol, a component that supports all the cache-line states might issue a ReadShared when the internal operation is a load; whereas, a component that supports a smaller set of the cache-line states might issue a ReadClean when the internal operation is a load. The challenge for the VIP developer is that the model must be able to represent both types of components.

A similar challenge exists when considering all the possible responses a component can make to a transaction. The protocol specification will often indicate all the allowable responses and then make a recommendation on which response to use. In the ACE protocol, when a ReadShared snoop hits on a cache line in the SD state in a master's cache, there are several responses allowed by the protocol. Here again, we see that the challenge for the VIP developer is to generate all the allowable responses. The master being snooped could:

- Transition to the I state, assert PassDirty, and deassert IsShared
- Transition to the SC state, assert PassDirty, and assert IsShared

- Stay in the SD state, deassert PassDirty, and assert IsShared

Some of the most complex pieces of RTL designs are the pieces of logic handling the conflict cases when there are multiple requestors of the same cache line. Therefore, it is important for VIP to be able to generate scenarios that will get the design into cases where these conflicts occur. A classic conflict example is when an agent is snooped for a cache line when it already has a write back transaction outstanding for that line. Generating these types of cache-line conflict scenarios is essential for effective verification.

For simulation-based verification, the checkers and the generators are distinct VIP components. For example, a set of properties is needed to check an interface that a design drives, and a model is needed to randomly generate traffic on an interface that is an input to the design. For formal-based verification, the VIP components are one and the same. In formal verification, the properties of the protocol are used as checks (assertions) when a design is driving an interface, and they are used as generators (constraints) when the interface is an input to the design. To clarify, a formal verification tool is not actually generating transactions, it is considering all possible transactions based on the properties written for the rules for the protocol.

Three elements must be verified for successful cache-coherent protocol verification: the RTL design that implements a protocol component, the VIP that checks a protocol component, and the VIP that generates transactions into a protocol component. Often, these elements are verified simultaneously by connecting them to each other. For example, in a simulation environment, the checker VIP can be connected to the generator VIP. A comparable formal environment would be one where two instances of the VIP are connected back to back with one using the properties as assertions (checkers) and the other as constraints (generators).

Another, more thorough, verification option is to independently develop two similar pieces of VIP and then formally verify them against each other. Essentially, this is an equivalence check. For example, ARM engineers, and Jasper engineers independently developed single-interface VIP for the ACE protocol (and for the AXI protocol). The ARM VIP was targeted for simulation environments while the Jasper VIP was targeted to have maximal performance during formal analysis. The two independently developed pieces of VIP were then formally verified in several back-to-back configurations where the properties in each VIP were used as both constraints (generators) and assertions (checkers) (Figure 10). This verification process identifies holes in the property set (missing checks) as well as properties that are too strong. The latter presents the potential for over-constraint situations in formal analysis.

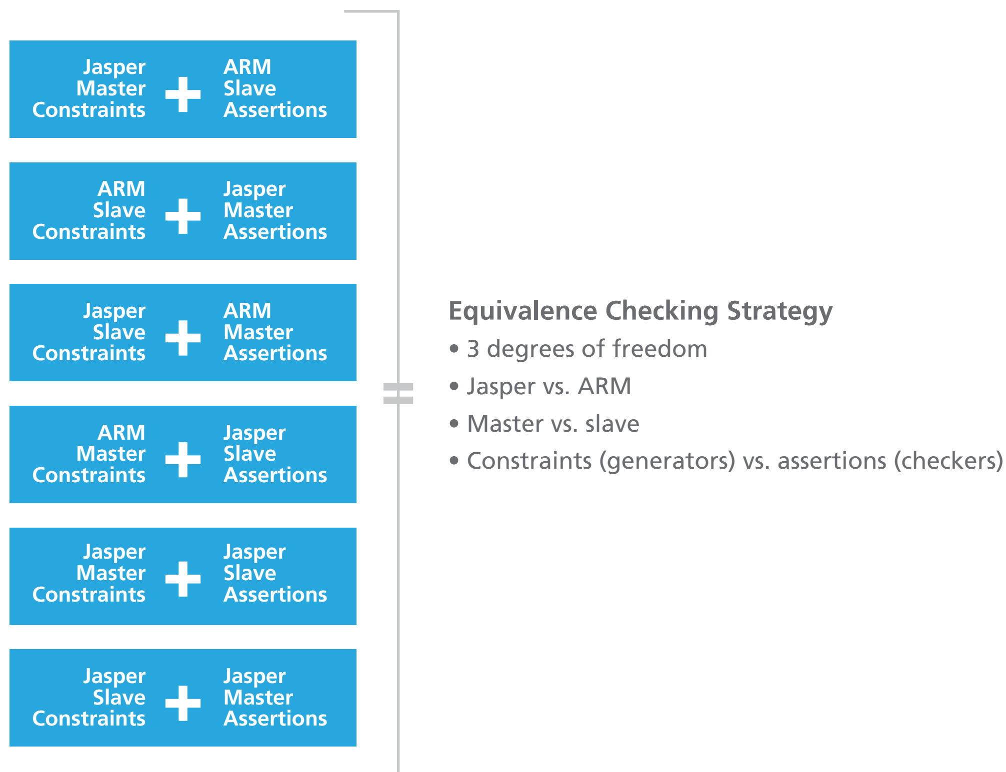

Figure 10: Checking the VIPs back-to-back

The end goal is to ensure that RTL designs are built correctly and maintain the cache-coherency attributes of the system. To that end, verification environments are built with a combination of VIP components and RTL components. The verification target in these environments is typically the RTL. However, VIPs—especially new VIPs—are also tested for correctness. In Figure 11, the cache-coherent interconnect RTL component has two ACE interfaces for master components to connect to and three AXI interfaces for slave components to connect to. In the verification environment, the VIP models take the place of actual RTL components, and an ACE system-level VIP is connected to both ACE interfaces to ensure proper system-level behavior. Jasper and ARM collaborated to perform formal verification in environments like this on both an ARM cache-coherent interconnect and a cache-coherent processor.

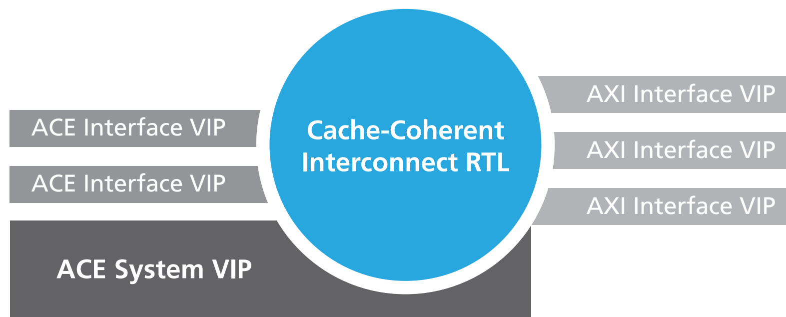

Figure 11: Checking an ACE system

Summary

Cache-coherent protocols and the RTL implementations of the components of the protocols provide unique verification challenges. Initially, the protocol must be modeled and verified to demonstrate that it adheres to high-level, cache-coherence rules. The protocol must also be understood by a large audience, such as RTL designers and verifiers. To enable high-quality verification of RTL protocol components, a sound methodology is imperative when developing VIP for the protocol. This paper described the collaboration of Jasper and ARM to address these challenges.

The Cadence Jasper ActiveModel technology provided the means to effectively model the ARM ACE protocol. The resulting model was used to not only verify the protocol; it also became a platform for querying the protocol to understand transaction flows and freedoms in the protocol. Finally, the ACE/ActiveModel technology became the system-level VIP used to verify cache-coherent ACE components alongside the ACE interface-level VIP.

cadence®

Cadence Design Systems enables global electronic design innovation and plays an essential role in the creation of today's electronics. Customers use Cadence software, hardware, IP, and expertise to design and verify today's mobile, cloud and connectivity applications. www.cadence.com

© 2016 Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, Jasper, and the Cadence logo are registered trademarks and QuietTrace is a trademark of Cadence Design Systems, Inc. in the United States and other countries. ARM and AMBA are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other trademarks are the property of their respective owners. .
4417 01/16 SA/DM/PDF