Formal Analysis of the ACE Specification for Cache Coherent Systems-on-Chip

Abderahman Kriouile[1,2] and Wendelin Serwe[2]

[1] STMicroelectronics, 12, rue Jules Horowitz, BP 217, 38019 Grenoble, France
[2] Inria/LIG, 655, av. de l'Europe, Montbonnot, 38334 Saint Ismier, France

Abstract. System-on-Chip (SoC) architectures integrate now many different components, such as processors, accelerators, memory, and I/O blocks, some but not all of which may have caches. Because the validation effort with simulation-based validation techniques, as currently used in industry, grows exponentially with the complexity of the SoC, we investigate in this paper the use of formal verification techniques. More precisely, we use the CADP toolbox to develop and validate a generic formal model of an SoC compliant with the recent ACE specification proposed by ARM to implement system-level coherency.

1 Introduction

The integration of ever more functionalities in set-top boxes or mobile appliances such as smartphones increases the complexity of both the embedded software and the hardware architecture. The latter is usually a complex System-on-Chip (SoC), featuring a significant number of heterogeneous components. Indeed, a typical SoC includes nowadays not only processors and memory, but also dedicated hardware accelerators and (analog) I/O blocks. Integrating caches into some of these components (in particular, into processors and hardware accelerators) can increase performance and reduce power consumption, for instance by avoiding accesses to (possibly off-chip) memory.

In the past, prevalence of fast processors encouraged designers to manage cache coherency in software, taking advantage of the flexibility of software solutions. However, due to increased software complexity, a recent trend [14, 23] is to introduce hardware support for cache coherency to improve performance and to lower power consumption by lightening the load on the processors. Hence, ARM proposed ACE (AXI Coherency Extensions) [1], which is becoming a de facto industrial standard for system-level cache coherence in heterogeneous SoCs (ACE explicitly includes operations, called ACE-Lite operations, for components without cache). ACE is used in ARM's *big.LITTLE* framework, which takes advantage of two processors (i.e., a "big" and a "LITTLE" one) for low-power SoCs. Also, STMicroelectronics is about to integrate system level coherency (based on ACE) in its upcoming SoCs.

As cache coherence protocols are known to be complex and difficult to validate, assuring system-level cache coherency is one of the major challenges faced by architects of current SoC and NoC (Network-on-Chip) designs. Current industrial validation flows are based on simulation techniques. Because the related validation effort grows exponentially with the complexity of hardware architectures, we study the application of formal verification techniques, where the human modeling effort increases linearly with the complexity of architectures (each component is modeled by a process). Thus, the exponential complexity is supported by automated verification tools.

Concretely, we use the CADP toolbox [10] and its modeling language LNT [3] for the analysis of system-level cache coherency in a heterogeneous SoC. We focus on enumerative model checking methods to prove their feasibility on an industrial case study.

As a first step, we develop a generic formal LNT model of an SoC, including an ACE-compliant cache coherent interconnect and abstractions of master and slave components (e.g., processors and shared memory). The model is parametric and can be instantiated with different configurations (number of masters, number of cache lines, and number of memory lines) and different sets of supported ACE transactions. We use a constraint-oriented specification style to model the global requirements of the ACE specification, which must be guaranteed by any implementation. The LNT model enables STMicroelectronics architects to interactively simulate a coherent SoC at system level. We also express several correctness properties in the MCL language [15] and check them on the LNT model using the EVALUATOR 4.0 model checker. From the counterexamples generated by EVALUATOR 4.0, we extract interesting scenarios to be tested on any implementation of an ACE-compliant interconnect.

The rest of this paper is organized as follows. Section 2 presents the ACE specification. Section 3 describes our LNT model of an ACE-compliant SoC. Section 4 discusses the validation of correctness properties. Section 5 surveys related work. Section 6 gives concluding remarks and directions of future work.

2 System Level Cache Coherency with ACE

In general, a System-on-Chip (SoC) is composed of different hardware blocks like generic or specialized processors, memories, interconnects, dedicated Intellectual Properties (IPs), or input/output components. These heterogeneous components usually access a *shared memory* consisting of several *memory lines*. To increase data access performance, some components may use a *cache*, containing local copies of memory lines. An SoC is called *cache coherent* if write operations to the same memory line by two components are observable in the same order by all components of the system. One may distinguish *sharable* and *non-sharable* memory lines. For example, the graphics memory of an SoC might be dedicated to image processing and exclusively used by the Graphics Processing Unit (GPU), whereas the remaining memory might be used by either the generic processors (Central Processing Units, CPUs) and the GPU: in this case, the graphics memory is non-sharable, and the remaining memory is sharable.

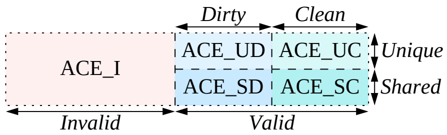

Fig. 1. ACE states of a cache line

The components of an SoC can be grouped into *master* components (such as CPUs) and *slave* components (such as memories). Components communicate via an interconnection medium, called the *interconnect*. In the case of a cache coherent system, the interconnect is also called a Cache Coherent Interconnect (CCI). Each component communicates with the interconnect by a communication port, each of which may consist of several channels. Operations performed on ports are called *transactions*.

2.1 ACE

To support system-level coherency, ARM has recently proposed the ACE (AXI Coherency Extensions) protocol specification [1, 22], which extends the AMBA (Advanced Microcontroller Bus Architecture)/AXI (Advanced eXtensible Interface) specification. ACE is designed to maintain coherence when sharing data across caches of an SoC, to enable interaction between heterogeneous components, and to ensure maximal reuse of cached data. ACE also supports a flexible framework for system level coherency: the system designer can determine the ranges of memory lines that are coherent, the system components that implement the coherency extensions, and the communication policies.

The ACE specification defines the hardware interface protocol (between components and the interconnect), the expected behavior of the components, and the responsibilities of the interconnect. ACE admits different cache coherence policies, known as directory based, snoop filter, or no snoop filter models.

2.2 ACE States

ACE distinguishes five states (shown in Figure 1) of a cache line.

A cache line is *invalid* if it does not contain a copy of any memory line. A cache line is *unique* if all other copies of the same memory line are invalid. A cache line is *shared* if all other copies of the same memory line are shared or invalid. A cache line is *dirty* (respectively *clean*) if the master is responsible (respectively not responsible) of writing the data back to the shared memory.

2.3 ACE Ports and Channels

The ACE specification distinguishes three kinds of ports to connect a component to an interconnect. An *ACE port* is used for components having a cache memory.

An *ACE-Lite port* is used for components without a cache. An *AXI port* is used for components that do not use coherency.

Each port consists of several channels. ACE distinguishes three types of channels: *read channels*, *write channels*, and *snoop channels*. Read (respectively, write) channels are used to read (respectively, write) data; these channels extend AMBA AXI channels with coherency related parameters. Read channels are the *address read* channel (AR, to send read requests) and the *data read* channel (R, to send the data back). Write channels are the *address write* channel (AW, to send write requests), the *data write* channel (W, to send the data to be written), and the *write response* channel (B, to signal completion of a write).

Snoop channels are used for snoop requests issued by the interconnect to masters with a cache. Snoop channels are the *address coherency* channel (AC, to send snoop requests), the *coherency response* channel (CR, to answer snoop requests, indicating whether a data transfer will follow), and the *coherency data* channel (CD, to send data to the interconnect).

2.4 ACE Transactions

The ACE specification defines several types of transactions. In the sequel, we focus on a significant subset of the transactions related to cache coherency. For each transaction, we present the expected order of operations on the channels. A master initiating a transaction is called *initiator*. A master with a cache receiving a snoop from the CCI is called a *snooped master*.

Snoop transactions are initiated by the interconnect while handling coherent transactions and cache maintenance transactions (see below). The interconnect initiates a snoop request on the AC channel. The snooped master responds on the CR channel indicating if a data transfer is needed. If so, the data is transferred on the CD channel indicating also whether the data is shared and whether the snooped master keeps the responsibility to write the data to memory.

Coherent transactions are used to access sharable memory lines, which might be in the caches of other components. We focus on four coherent transactions, all of which are initiated by a master through a request on the AR channel. The interconnect initiates corresponding snoop transactions to all other masters with a cache and, if necessary, reads the data from the sharable memory. Finally, the interconnect sends a reply transaction to the initiator on the R channel, indicating whether the data is shared and whether the responsibility to write the data to memory is passed to the initiator.

- A *ReadShared* transaction obtains a copy of the memory line without any constraint on the resulting state of the cache line.
- A *ReadUnique* transaction obtains a copy of the memory line and ensures that the copy is unique (i.e., no other copies exist).
- A *MakeUnique* transaction invalidates all other copies of the memory line.
- A *ReadOnce* transaction obtains the current contents of a memory line, which may not be copied into the cache.

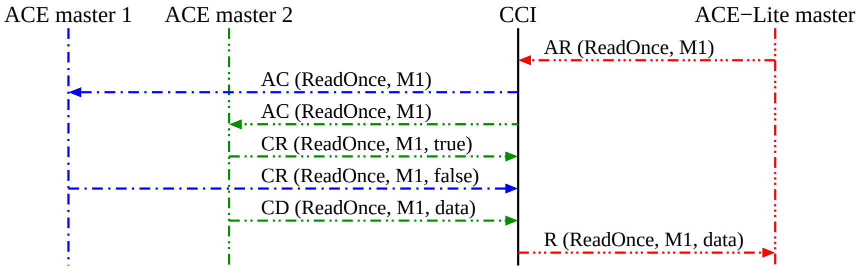

Fig. 2. Execution scenario of a *ReadOnce* transaction

Example 1. Consider an SoC with two ACE masters, a CCI, and an ACE-Lite master. Figure 2 shows the execution of a *ReadOnce* transaction (for memory line M1) initiated by the ACE-Lite master. The CCI snoops both ACE masters, which answer with a Boolean indicating whether the data is in their cache. The cache of ACE master 2 contains the data, hence this master also sends the data, which the CCI forwards to the ACE-Lite master to complete the transaction.

Non-snooping transactions are used to access non-shareable memory lines which must not be in the caches of other master components. We consider two non-snooping transactions: *ReadNoSnoop* and *WriteNoSnoop*.[3]

Memory update transactions are used to update shared memory. These transactions (e.g., *WriteBack*) are initiated by a master on the AW channel; the data to write is sent by the master on the W channel. The interconnect writes the data to the memory and returns an acknowledgement on the B channel.

Cache maintenance transactions are used by master components to access and impact the caches of other components. In particular, cache maintenance transactions enable a master to observe the effect of load and store operations on system caches (which cannot otherwise be accessed). The ACE specification distinguishes three cache maintenance transactions: *CleanShared*, *CleanInvalid*, and *MakeInvalid*. These transactions are initiated by sending a request on the AR channel. The interconnect initiates corresponding snoop transactions to all other masters with a cache. For a *CleanShared* transaction, a snooped master may retain its local copy of the memory line, but for a *CleanInvalid* or *MakeInvalid* transaction, a snooped master must invalidate its local copy. For a *CleanShared* or *CleanInvalid* transaction, a snooped master must also provide the data if the corresponding cache line is dirty. After all snooped masters have answered, the interconnect returns an acknowledgement to the initiator, on the R channel.

ACE-Lite transactions are a subset of ACE transactions, namely: *ReadNoSnoop*, *ReadOnce*, *CleanShared*, *CleanInvalid*, and *MakeInvalid*.

[3] Those transactions are equivalent to the AXI Read and AXI Write transactions.

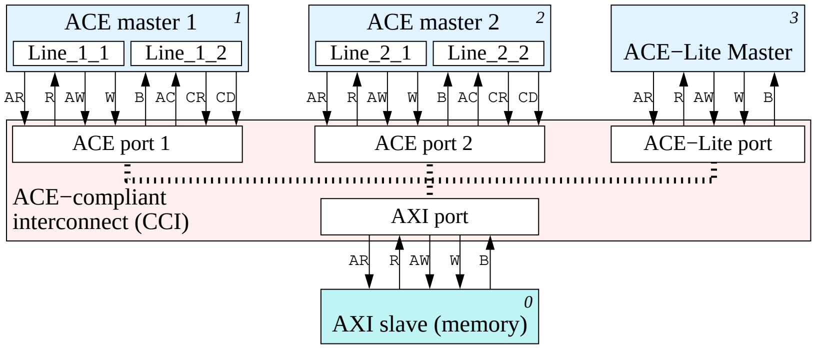

Fig. 3. Model architecture

3 Formally Modeling an ACE-compliant SoC in CADP

We developed a formal model of an ACE-compliant SoC, consisting of a CCI, masters, and slaves, using the LNT (also called LOTOS NT) language [3], supported by the CADP toolbox [10]. LNT combines the best features of process-algebraic and imperative programming languages. The semantics of LNT model is defined as a *Labeled Transition System* (LTS), following a black box view of the system. The LNT.OPEN tool translates an LNT model into an LTS suitable for (on-the-fly) verification.

Our formal model (about 3200 lines of LNT code) represents the behavior of the system focusing on the interactions between components. It is parametric and can be instantiated with different configurations (number of masters, number of cache lines for each master, number of memory lines per slave, etc.). The model is generic in the sense that it includes all the behaviors permitted by the ACE specification for any correct implementation. The masters are non-deterministic agents, which may initiate all the transactions described in Section 2.

We opted for modeling a *fully connected snoop* topology, i.e., all coherent transactions lead to snoop transactions for all masters with cache. Note that the first industrial implementation [2] of the ACE protocol also has a fully connected snoop topology.

Each operation on a channel is modeled by an LNT rendezvous[4] on a gate of the same name as the channel.

Example 2. Figure 3 shows the model of an SoC consisting of a CCI, two ACE masters, an ACE-Lite master, and a shared memory (consisting of three memory lines). Each ACE master contains two cache lines. The component index is 0 for

[4] The semantics of an LNT rendezvous avoids the need to model the acknowledgement *signals* at the level of channel transmission. However, the acknowledgement *operation* for a non-atomic transaction (e.g., the operation on the B channel for Write transactions) is represented by an independent LNT rendezvous (on gate `B`).

the shared memory, 1 (respectively, 2) for the ACE masters, and 3 for the ACE-Lite master. Notice that this configuration shares most characteristics with the big.LITTLE architecture.

3.1 Types and Data Structures

Each memory line is characterized by two parameters: an index (of range type `Index_Mem`, where `N` is the number of memory lines) and a data (of type `Nat`). Hence the shared memory can be represented by an array of values of type `Nat`, indexed by the range of `Index_Mem`. ACE states are represented by an enumerated type called `ACE_state`.

```
type Index_Mem is range 1 .. N of Nat end type
type Cache_Line is
   LINE_C (indC: Index_Cache, S: ACE_state, indM: Index_Mem, data: Nat)
end type
type Mem_Lines is array [1 .. N] of Nat end type
type ACE_state is ACE_I, ACE_UC, ACE_UD, ACE_SC, ACE_SD end type
```

Similarly, we define an index for system components (`Index_Component`) and an index for cache lines of a master with cache (`Index_Cache`). ACE transactions are modeled by an enumerated type `ACE_Trans`.

We introduce an abstract transaction $\mathcal{A}$ that simulates any ACE transaction by executing all the phases of an ACE transaction without changing the ACE state of cache lines.

3.2 Channels

Each ACE channel is modeled by a typed LNT gate. The types of LNT gates (called LNT channels) specify the number and types of the parameters (called *offers*), i.e., the values exchanged during a rendezvous. All gates have an offer to represent the ongoing ACE transaction, an offer to represent the initiator of the current transaction, and an offer to designate the concerned memory line. Snooping gates (`AC` and `CR`) have also an offer to represent the snooped master. Gates which transfer data (`R`, `W`, and `CD`) have also an offer for the data. The gates `R` (read data channel) and `CD` (snoop data channel) have also three Boolean offers. *DataStatus* indicates whether the data is valid, *PassDirty* indicates whether the responsibility of writing data to memory is passed, and *IsShared* indicates whether the data is shared. The gate `CR` has a Boolean offer *DataTransfer* to indicate if a data transfer will be follow on the `CD` gate. The gate `B` has a Boolean offer indicating if the write has completed correctly.

For verification purposes, we add an offer representing the ACE state of the cache line to all gates going out from an ACE master (i.e., `AR`, `AW`, and `CR`). Similarly, the gates between the CCI and a slave have an additional offer corresponding to the initiator.

```
process memory [AR: CHANNEL_AXI_AR, R: CHANNEL_AXI_R,
                AW: CHANNEL_AXI_AW, W: CHANNEL_AXI_W, B: CHANNEL_AXI_B]
               (idMEM: Index_Component)
is
 var LINES: Mem_Lines, pending_read: Bool, transR, transW: ACE_Trans,
     ind_R, ind_W: Index_Mem, CPU_R, CPU_W: Index_Component, data: Nat
 in
   -- initializations (not included)
   loop select
      when pending_read == false then
         AR (?transR, idMEM, ?indM_R, ?CPU_R);
         pending_read := true
      end when
   []
      when pending_read == true then
         R (transR, idMEM, LINES[Nat(indM_R)], CPU_R);
         pending_read := false
      end when
   []
      AW (?transW, idMEM, ?ind_W, ?CPU_W);
      W (transW, idMEM, ind_W, ?data, CPU_W);
      LINES[Nat(ind_W)] := data;
      B (transW, idMEM, indM_W, true, CPU_W)
   end select end loop
end var end process
```

Fig. 4. LNT process representing the shared memory

3.3 ACE Slave: Shared Memory

The shared memory is modeled by the LNT process shown in Figure 4. The five gates `AR`, `R`, `AW`, `W`, and `B` correspond to the AXI channels. We use the Boolean `pending_read` to indicate if a read operation is in progress. The behavior of the memory process is a non-terminating `loop`, the body of which is a non-deterministic choice (`select`[5]) between three possibilities:[6]

- Receiving a read request on the `AR` gate, which is only possible if no read operation is in progress (`pending_read == false`),
- Sending back a read data on the `R` gate, which is only possible if a previous read request was received (`pending_read == true`),
- Receiving a write request.

In our model, a write operation cannot be interrupted by a read operation.

[5] The LNT construction "`select` A `[]` B `[]` C `end select`" presents a non-deterministic choice between A, B, and C.

[6] In an LNT rendezvous, an offer "`?x`" accepts any value of the same type as variable `x`, and the received value is stored in variable `x`.

3.4 ACE Masters

The cache lines of a master are essentially independent from each other, i.e., transactions on different cache lines can freely interleave.[7] Hence we choose to model each master by a parallel composition of cache lines. Each cache line is modeled by five mutually recursive LNT processes: (1) process *cpu* initializes the cache line; (2) process *cpu_ready* represents a cache line that is ready to initiate an ACE transaction or to receive a snoop request from the CCI; (3) process *cpu_reply* represents a cache line that has previously initiated an ACE transaction and waits for the reply from the CCI (the cache line is also ready to receive any snoop request from the CCI); (4) process *cpu_snoop* represents a cache line that has previously received a snoop request from the CCI and can either reply to this request or initiate a new ACE transaction; (5) process *cpu_reply_snoop* represents a cache line that has previously initiated an ACE transaction and has also received a snoop request from CCI: thus, it is both waiting for the reply of the ACE transaction and ready to reply to the snoop request. Each of these processes behaves as a large non-deterministic choice between all possible rendezvous. Each branch consists of a guard, a rendezvous with parameters to handle the ongoing transaction, and a recursive call corresponding to the new state of the cache line.

In order to generate a smaller LTS in the debug phase of the model, we can deactivate the non-deterministic choice of cache lines relative to the state changes permitted by the ACE specification.

3.5 ACE-Lite Masters

The LNT model of an ACE-Lite master is obtained from the model of an ACE master by removing the handling of snoop requests. Thus, an ACE-Lite master is modeled by three mutually recursive LNT processes: a process to initialize the ACE-Lite master, a process *lite_ready*, which can initiate any of the ACE-Lite transactions presented in Section 2.4, and a process *lite_reply*, which waits for a reply from the CCI.

3.6 Cache Coherent Interconnect (CCI)

To ease the modeling of all interleavings between the ports of the CCI, we employ two techniques. First, we model the CCI by a parallel composition of as many processes as there are ports; each port is always ready to receive a request from both the corresponding component and other ports. Second, all received requests are stored in a set and are handled in any order.[8]

The ports of the CCI communicate internally via dedicated gates, which are not part of the ACE specification and can be hidden (in the LTS and in counterexamples), but are useful in the debug phase of the model.

---

[7] Actually, the only constraint is to store the same memory line in at most one cache line of a same master.

[8] Because the numbers of CPUs and cache lines are fixed, the number of requests in a set is bounded by construction.

Example 3. The CCI of Figure 3 contains four ports: two ACE ports, each communicating with an ACE master, one ACE-Lite port communicating with an ACE-Lite master, and one AXI port communicating with the shared memory.

3.7 Requirements on the Global Ordering of Transaction

The ACE specification includes some global requirements concerning system-level coherency for the implementation of any ACE-compliant interconnect. Following a constraint-oriented specification style, our LNT model integrates these global requirements as dedicated processes (one process per requirement and memory line), composed in parallel with the remainder of the model. Hence, those processes monitor the system and have a global view of all transactions. There are two kinds of global requirements:

– Coherency between caches (called *horizontal coherency*) [1, section C4.10]: When two masters attempt to write to the same memory line simultaneously (i.e., the second transaction begins before the end of the first transaction), then the interconnect must ensure a strict order of the transactions. Concretely, while handling a snoop transaction, the constraint process ensures that a subset of snoop transactions (relative to the same memory line) are not handled before the end of the first transaction.
– Coherency between the memory and caches (called *vertical coherency*) [1, section C6.5.3]: Data received from caches must be written to the memory in the correct order. The constraint process monitors write transactions, prohibiting that an old data overwrites a more recent one.

3.8 State Space Generation

For our analysis, we consider several SoC configurations, each consisting of a shared memory, one ACE-Lite master, and two ACE masters, with two cache lines each. To focus on coherency issues, the first cache lines of each ACE master execute transactions concerning the same memory line (this is suitable according to [8]). Each master initiates at most one transaction (chosen from a set of allowed transactions); thus, the second cache lines of each ACE master never initiate a transaction (but answer snoop requests). We selected subsets of transaction that could create problems for properties to verify.

For each considered configuration, Table 1 gives the size of the corresponding LTS. Columns one, two, and three give the set of transactions that master 1 (respectively, master 2, or the ACE-Lite master) are allowed to initiate. Column four tells whether the model includes the processes enforcing the global ordering requirements; we generate LTSs for models without the corresponding constraint processes to study their impact on the properties of the system. An LTS of the model with global constraints is included in the one without global constraints with respect to strong bisimulation (i.e., the constraints only removed behaviors), but the state space may be larger because a state now also integrates the current state of the control process.

Table 1. Experimental results: state space generation and verification

allowed transactions			global	LTS size		properties				
m1	m2	lite	constraints	states	transitions	φ_1	φ_2	φ_3	φ_4	φ_5
S_0	$\{\mathcal{A}\}$	S_0	yes	93,481,270	308,087,560	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_0	$\{\mathcal{A}\}$	S_0	no	105,376,971	351,344,207	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
S_0	$\emptyset$	S_0	yes	7,518,552	21,227,610	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_1	$\emptyset$	S_1	yes	3,685,311	10,649,422	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_1	$\emptyset$	S_1	no	3,127,707	9,121,134	$\surd$	$\surd$	$\times$	$\times$	$\times$
S_2	S_2	$\emptyset$	yes	3,545,801	11,122,536	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_2	S_2	$\emptyset$	no	2,819,505	9,095,620	$\surd$	$\surd$	$\times$	$\times$	$\surd$
S_3	$\emptyset$	S_3'	yes	1,834,195	5,170,829	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_3	$\emptyset$	S_3'	no	1,437,412	4,547,398	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
S_4	S_4	$\emptyset$	yes	560,299	1,669,886	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_4	S_4	$\emptyset$	no	599,971	1,780,634	$\surd$	$\surd$	$\times$	$\times$	$\times$
S_5	S_5	$\emptyset$	yes	40,983	63,922	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_5	S_5	$\emptyset$	no	55,439	98,688	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

In the table above, we use those sets of allowed transactions:
$S_0 =$ set of all ACE (respectively ACE-Lite) transactions
$S_1 = \{\mathit{MakeUnique}, \mathit{ReadOnce}, \mathit{ReadUnique}, \mathit{WriteBack}\}$
$S_2 = \{\mathit{MakeInvalid}, \mathit{MakeUnique}, \mathit{ReadShared}, \mathit{ReadUnique}, \mathit{WriteBack}\}$
$S_3 = \{\mathit{MakeUnique}, \mathit{WriteBack}\}$, $S_3' = \{\mathit{ReadOnce}\}$
$S_4 = \{\mathit{CleanInvalid}, \mathit{CleanShared}, \mathit{ReadUnique}, \mathit{WriteBack}\}$
$S_5 = \{\mathit{MakeInvalid}, \mathit{MakeUnique}, \mathit{WriteBack}\}$

4 Validation

We verify several system-level properties on our model of the ACE-compliant SoC presented in Example 2. We start by validating the complete and correct execution of separate transactions, then we verify the coherency of the cache states, and finally we check data integrity in the system.

We express all these properties in *Model Checking Language* (MCL) [15], an extension of the modal μ-calculus with high-level operators aimed at improving expressiveness and conciseness of formulæ. The main ingredients of MCL are parametrized fixed points, action patterns enabling to extract data values from LTS transition labels, modalities on transition sequences described using extended regular expressions and programming language constructs, and an infinite looping operator specifying fairness. The EVALUATOR 4.0 model checker of CADP can verify MCL properties on the fly, based on the local resolution of Boolean equation systems, which has a linear-time complexity for (data-less) alternation-free and fairness formulæ. We wrote also several macros to simplify writing properties for industrial users.

4.1 Complete Execution of Transactions

To verify that every transaction inevitably finishes, we use the following two liveness formulæ (φ_1 and φ_2):

```
[ true * . { AR ?op:String ?n:Nat ?l:Nat ... } ] inev ( { R !op !n !l } )
[ true * . { AW ?op:String ?n:Nat ?l:Nat ... } ] inev ( { B !op !n !l } )
```

These formulæ use the macro `inev (L)`, which expresses that a transition labeled with `L` will eventually occur. This macro can be defined as follows:

```
macro inev (L) = mu X . ( < true > true and [ not L ] X ) end_macro
```

The first (respectively second) formula requires that each action `AR` (respectively `AW`) is eventually followed by an action `R` (respectively `B`). Note the capture of the exchanged values into the variables `op`, `n`, and `l` in the first action predicate (using the LNT-like syntax "`?`*variable*`:` *Type*", where *Type* is one of the predefined types of MCL) and the use of the captured values in the second action predicate.

4.2 Cache Coherency

To verify the coherency of the ACE states of all the caches of the system, we have to translate the state-based properties to action-based properties, using the ACE state offer added to transactions issued by cache lines (see Section 3.2). To simplify the formulæ and to reduce verification complexity, we rename these transitions using a unique gate `G` keeping only useful offers.

Two safety formulæ express coherency. The first one (φ_3), requires that if a cache line is in the state ACE_UD then all other cache lines containing the same memory line must be in the state ACE_I:

```
[ true * .
   {G ?m1:Nat ?indM:Nat "ACE_UD"} .
   ( not ({G !m1 !indM ?s1:String where s1<>"ACE_UD"}) ) * .
   {G ?m2:Nat !indM ?s2:String where (m2<>m1) and (s2<>"ACE_I")}
] false
```

The second formulæ (φ_4) is similar to φ_3 and requires that if a cache line is in the state ACE_SD then all other cache lines containing the same memory line must be either in the state ACE_SC or the state ACE_I.

```
[ true * .
   {G ?m1:Nat ?indM:Nat "ACE_SD"} .
   ( not ({G !m1 !indM ?s1:String where s1<>"ACE_SD"}) ) * .
   {G ?m2:Nat !indM ?s2:String
      where (m2<>m1) and (s2<>"ACE_I") and (s2<>"ACE_SC")}
] false
```

4.3 Data Integrity

To verify the data integrity of the system, we use a safety property (φ_5), which enforces a correct order of write operations to the shared memory:

```
[ true * .
   { W !"WRITEBACK" ?c:Nat ?l:Nat ?d:Nat } .
   ( not { W !"WRITEBACK" !"0" !l !d !c } ) * .
   { W !"WRITEBACK" !"0" !l !d !c } .
   (
      ( not { AC ?any of String ?any of Nat !c ?any of Nat !l } ) and
      ( not { W ?any of String !"0" !l ?any of Nat ?any of Nat } )
   ) * .
   { W ?any of String !"0" !l ?h:Nat ?any of Nat where h<>d }
] false
```

Once a master `c` initiates a *WriteBack* transaction to a memory line `l` of a data `d`, and effectively written to memory (which has port number `0`), the property forbids a data `h` different from `d` to be written to the same memory line `l` without previously receiving a snoop request concerning line `l`.[9]

4.4 Model-Checking Results

The verification results of the properties presented in Sections 4.1 to 4.3 on the LTSs of Section 3.8 are given in columns seven to eleven of Table 1. All LTSs including the global ordering requirements satisfy ($\surd$) all five properties.

For LTSs without global ordering constraints, coherency (φ_3 and φ_4) and data integrity (φ_5) properties may not be satisfied ($\times$). In this case, EVALUATOR 4.0 generates minimal counterexample sequences, which correspond to scenarios to be tested (using an industrial testbench) on any implementation of an ACE-compliant interconnect. This interests STMicroelectronics, because these intricate test cases challenge the (complex) implementation of the coherency constraints in an interconnect.

5 Related Work

Formal verification techniques, e.g., (symbolic) model checking and theorem proving, has been often applied to the verification of hardware designs of cache coherence protocols, using various modeling languages, temporal logics, and verification tools [17, 13]. Most works [4, 6, 7, 9, 12, 16, 18, 20, 21] concern elaborated protocols using more complex topologies than the fully connected snoop topology of our LNT model. The principal differences to our work are that we focus on a generic interconnect that includes the behavior of all correct implementations and that we study a heterogeneous SoC, rather than verifying a particular coherency protocol for a homogeneous system. Notice that the notion of a component without cache (ACE-Lite) snooping components with caches was introduced by the ACE specification.

[9] The number of parameters differs for the rendezvous on gate `W` between the CCI and the memory and those between a master and the CCI: for the former, the fifth parameter corresponds to index of the initiator

The only paper dedicated to the formal verification of the ACE specification is a methodological guide [19], which shows the benefits of high-level modeling of system-level cache coherency using Jasper's formal verification tools. Our approach differs from [19] by addressing heterogeneous systems (in particular ACE-Lite masters) and by presenting validation results on an example.

6 Conclusion

We developed a generic formal LNT model of the recent ACE specification [1]. The constraint-oriented specification style proved helpful in the modeling of general requirements expressed in natural language. Our model has been found valuable by STMicroelectronics architects, because it enables interactive and backtrackable step-by-step system-level simulation (using the OCIS tool) of all ACE-compliant behaviors. We expressed correctness properties as temporal logic formulæ (in MCL) and verified them automatically (using the EVALUATOR 4.0 tool). Hence, we found that formal verification techniques can be used for the analysis of heterogeneous coherent SoCs.

This work can be pursued along several directions. First, the generic interconnect model can be used to analyze the impact of a coherent interconnect in a model of a concrete SoC. This requires to refine the models of masters and slaves to match those used in the SoC. Second, the formal model can be used to guide test and validation, as a reference model for co-simulation or by (automatically) extracting interesting test scenarios [11, 5]. For instance, the counterexamples of Section 4 seem interesting test cases for any ACE-compliant interconnect. STMicroelectronics has expressed interest in both directions, as they address issues faced in the development of future products.

Acknowledgements. We are grateful to R. Mateescu (Inria) and M. Zendri (STMicroelectronics) for their contribution and their valuable remarks. We would also like to thank H. Garaval (Inria), G. Barthes, C. Chevallaz, G. Faux, O. Haller, and M. Soulie (STMicroelectronics) for helpful discussions.

References

1. ARM. *AMBA AXI and ACE Protocol Specification*, Feb. 2013. version ARM IHI 0022E, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022e.
2. ARM. *CoreLink CCI-400 Cache Coherent Interconnect: Technical Reference Manual*, Nov. 2012. revision r1p1, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0470g/DDI0470G_cci400_r1p1_trm.pdf.
3. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to LOTOS translator (version 5.8). INRIA/VASY, 155 pages, Mar. 2013.
4. G. Chehaibar. Integrating formal verification with Murϕ of distributed cache coherence protocols in FAME multiprocessor system design. In D. de Frutos-Escrig and M. Núñez, editors, *Proc. of the Int. Conf. on Formal Techniques for Networked and Distributed Systems*, *LNCS* 3235, pp. 243–258. Springer, Sept. 2004.

5. M. Chen, X. Qin, H.-M. Koo, and P. Mishra. *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer, 2013.
6. X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Formal Methods in System Design*, 36(1):37–64, Feb. 2010.
7. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, Mar. 1995.
8. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proc. of the Int. Conf. on Computer Design: VLSI in Computers and Processors*, pp. 522–525. IEEE, Oct. 1992.
9. A. T. Eiríksson and K. L. McMillan. Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study. In *Proc. of the Int. Conf. on Computer Aided Verification*, *LNCS* 939, pp. 367–380. Springer, July 1995.
10. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2011: A toolbox for the construction and analysis of distributed processes. *Software Tools for Technology Transfer*, 15(2):89–107, Apr. 2013.
11. H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In *Proc. of the Int. Workshop on Testing of Communicating Systems*. Chapman&Hall, 1998.
12. H. K. Kapoor, P. Kanakala, M. Verma, and S. Das. Design and formal verification of a hierarchical cache coherence protocol for NoC based multiprocessors. *The Journal of Supercomputing*, 2013.
13. C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Trans. on Design Automation of Electronic Systems*, 4(2):123–193, 1999.
14. M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-Chip Cache Coherence Is Here to Stay. *Communications of the ACM*, 55(7):78–89, July 2012.
15. R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *Proc. of Int. Symposium on Formal Methods*, *LNCS* 5014, pp. 148–164. Springer, May 2008.
16. K. L. McMillan and Schwalbe. Formal Verification of the Encore Gigamax cache consistency protocol. In *Proc. of the Int. Symposium on Shared Memory Multiprocessors*, pp. 242–251, 1991.
17. F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, Mar. 1997.
18. F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. In *Proc. of the Int. Conf. on Parallel Processing*, *LNCS* 966, pp. 287–300. Springer, Aug. 1995.
19. R. Ranjan. Formal Techniques for Protocol Verification: A Case Study On Verifying the ARM ACE Protocol. *Electronic Design*, Jan. 2012.
20. A. Slobodová, J. Davis, S. Swords, and W. Hunt, Jr. A Flexible Formal Verification Framework for Industrial Scale Verification. In *Proc. of the Int. Conf. on Formal Methods and Models for Codesign*, pages 89–97. July 2011.
21. U. Stern and D. L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Correct Hardware Design and Verification Methods*, *LNCS* 987, pp. 21–34. Springer, 1995.
22. A. Stevens. *Introduction to AMBA 4 ACE*. ARM whitepaper, June 2011.
23. C. Thompson. *Verifying Cache Coherency Protocols with Verification IP*. Synopsis, Oct. 2012.