ADVOCAT: Automated Deadlock Verification for On-chip Cache coherence and Interconnects

Freek Verbeek*, Pooria M. Yaghini†, Ashkan Eghbal† and Nader Bagherzadeh†
*Open University of The Netherlands
Radboud University, Nijmegen
†University of California, Irvine

***Abstract*—Cache coherence plays a major role in manycore systems. The verification of deadlocks is a challenge in particular, because deadlock freedom is an emergent property. Formal methods often decouple verification of the protocol from verification of the communication interconnect. Modern communication fabrics, however, become more advanced and include a network topology, routing, arbitration, synchronization, and more. In this paper, an integrated approach is proposed that allows cross-layer verification of both the cache coherence protocol and the communication fabric all at once. An automated methodology for deriving cross-layer invariants is proposed. These invariants relate the state of the application-layer protocols to en route packets in the communication fabric. We apply this methodology in a case study where cross-layer deadlocks occur if queues are wrongly sized. Our methodology is generally applicable and shows promising scalability.**

1. Introduction

One of the major challenges in the design and verification of manycore systems is cache coherence. In the context of bus-based architectures or point-to-point networks, this problem has been subject to heavy research and a large number of solutions exist. When replacing such architectures with an interconnection network, however, many new problems arise. Efficient, correct and deadlock-free solutions for cache coherence in such manycore systems are still an open problem.

Often, formal verification of cache coherence analyzes the communication fabric independently from the protocol [1]. For example, a key characteristic of a bus-based architecture is that bus access requests are serialized by some arbitration logic. It is therefor assumed that modelling the communication on the bus can be done using, e.g., synchronous handshaking semantics [2]. However, when the cores are connected using an interconnection network, this assumption is no longer valid. Injected packets can be *en route* while the cores are active, possibly inducing contention, message reordering (e.g., in case of adaptive routing), etc. Deadlocks may *emerge* from a deadlock-free coherence protocol and a deadlock-free interconnect, e.g., when queue sizes are too small. Discovering these kinds of deadlocks requires an integrated approach.

This paper proposes `ADVOCAT` (Automated Deadlock Verification for On-chip Cache coherence and InTerconnects). An integrated approach, that combines a common way of modelling cache coherence protocols (i.e., IO state automata) with an effective way of modelling communication fabrics (i.e., XMAS models introduced by Intel). This allows *cross-layer* verification. Our methodology provides cross-layer invariants, which formulate relations between the states of the protocols and the packets en route in the communication fabric. Using these invariants, we prove the absence of cross-layer deadlocks.

Our methodology is sound, but subject to false negatives. A "deadlock-free" result ensures a deadlock-free system, but if a deadlock is found it might be unreachable. Section 5 presents an extensive case study. In this case-study any discovered deadlock is actually reachable, which is confirmed using UPPAAL for small versions of the networks. We show that these deadlocks cannot be resolved by using virtual channels. Instead, deadlock freedom depends on the queue sizes and thus we use our methodology to find the smallest possible queue size.

Running Example. Consider Figure 1. Two automata S and T are connected via a trivial communication fabric consisting of two queues. The left automaton injects requests and consumes acknowledgments, the right automaton the other way around. When ignoring the communication fabric and considering the composition obtained by synchronous handshaking [2], the two automata are deadlock-free. The composed system is either in state $\langle s_0, t_0 \rangle$ or $\langle s_1, t_1 \rangle$, and in both states there is an enabled transition.

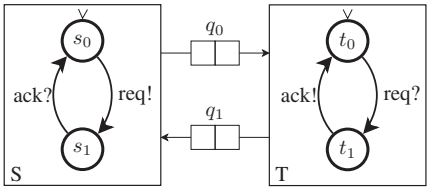

Figure 1. Two state automata connected by XMAS queues. The exclamation (question) mark represents sending (receiving) of a packet.

978-3-9815370-6-2/DATE16/

When considering the system and the communication fabric *integrally*, the proof of deadlock freedom becomes harder. State $\langle s_1, t_0 \rangle$ now is reachable. This state might be a deadlock, e.g., when the communication fabric is empty. This deadlock however is unreachable since whenever the automata are in state $\langle s_1, t_0 \rangle$, there is either a request en route from S to T or an acknowledgment from T to S. Invariants are needed to rule out these unreachable deadlocks. The invariant that is found automatically by our methodology is:

$$T.t_0 - S.s_0 = \#q_0.req + \#q_1.ack$$

This invariant is cross-layer, stating a relation between the states of the automata and the contents of the queues in the communication fabric. Variables $T.t_0$ and $S.s_0$ have numeric values that are 1 (0) if and only if the automaton is (not) in that state. The invariant shows that there is no backpressure in the fabric, since there is at most one packet in both queues. For that to happen, the automata must be in state $\langle s_1, t_0 \rangle$. Since we know that $S.s_0 + S.s_1 = 1$ (and the same for T), this invariant also shows that if the automata are not in state $\langle s_1, t_0 \rangle$, the communication fabric is empty. Finally, it also says that state $\langle s_0, t_1 \rangle$ is unreachable: in that case the left hand side of the equation would be -1, whereas the right hand side is always a natural number. This invariant is sufficiently strong to rule out any unreachable deadlock.

2. XMAS and IO Automata

An XMAS model is a network of primitives connected via channels. A channel connects an *initiator* primitive to a *target* primitive. Each channel consists of three *signals*. Channel signal $c.irdy$ is high when the initiator is ready to write to channel c. Channel signal $c.trdy$ indicates if the target is ready to read channel c. Channel signal $c.data$ contains data that is transferred from the initiator output to the target input if and only if both signals $c.irdy$ and $c.trdy$ are high.

The language XMAS provides eight basic primitives. A *queue* stores packets. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Messages are nondeterministically produced and consumed at *sources* and *sinks*. A source may process multiple message types. We assume sources are fair, i.e., they will always eventually be ready to transmit. Sinks are assumed to be either fair or dead. A *fork* duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two outputs are both ready to read. A *join* copies data from input a to its output, but such a transfer only takes place if the other input is ready to send as well. A *switch* uses its function parameter to determine towards which output an incoming packet must be routed. A *merge* is an arbiter. It grants its output to one of its inputs. We assume that merges are fair, e.g., round-robin or FIFO. For more explanation, we refer to [3].

XMAS Automata. To integrate IO automata with XMAS, they require an XMAS interface, i.e., they need to read from and write to channels consisting of $trdy$, $irdy$ and $data$ signals. An automaton can have multiple in-channels, allowing, e.g., to consume packets sent by the communication fabric and packets sent by the core. Similarly, it may also have multiple out-channels. The edges are labelled with an *event* and a *transformation*. An event is a function ε that takes as input an in-channel i and a packet d. It returns true if the automaton is ready to consume the packet. A transformation is a function φ that takes as input an in-channel i and a packet d. It returns a tuple (o, d') representing the emission of packet d' at channel o. It also may return $\perp$, indicating that no packet is produced by the transition.

Definition 1. Let D denote the type of packets. An *XMAS automaton* is a tuple $\langle S, T, s_0, C_I, C_O \rangle$ with S the set of states, T the set of transitions, s_0 the initial state and C_I (C_O) the set of in (out) channels. A *transition* $t \in T$ is a tuple $\langle s, s', \varepsilon, \varphi \rangle$ with s and s' the begin and end states, function $\varepsilon :: C_I \times D \mapsto$ ¯ an event and function $\varphi :: C_I \times D \mapsto (C_O \times D) + 1$ a transformation.

Semantics. The semantics of XMAS primitives are defined by equations denoting their effect on the $irdy$ and $data$ signals of its out-channels and the $trdy$ of its in-channel. We formulate these equations for XMAS automata. Whether a packet is consumed or produced by an XMAS automaton depends on whether transitions are *enabled*.

Definition 2. Let A be an XMAS automaton and let $A.s$ denote that automaton A is in state s. A transition $t = \langle s, s', \varepsilon, \varphi \rangle$ of XMAS automaton A *is enabled for in-channel* i, notation $\mathrm{enabled}(t, i)$, if and only if:

$$\begin{aligned} \mathrm{enabled}(t, i) \quad &\stackrel{\text{def}}{=} A.s \wedge i.irdy \wedge \\ &\qquad \varepsilon(i, i.data) \wedge \mathrm{rdy}(\varphi(i, i.data)) \\ \text{where } \mathrm{rdy}(\perp) &= True,\ \mathrm{rdy}(o, d') = o.trdy \end{aligned}$$

A transition is enabled if and only if the automaton is in the right state, and channel i is ready to transmit a packet that triggers the event and can be forwarded to an output (if applicable).

At all times, multiple transitions can be enabled, i.e., there can be multiple in-channels that contain packets which can be consumed. We assume the existence of a fair *selection function* $\mathtt{sel}_A$ that chooses an enabled transition and the corresponding in-channel. The semantics of XMAS automaton A are defined as follows (the underscore matches any value):

$$\begin{aligned} i.trdy &:= \mathtt{sel}_A = (i, \_) \\ o.irdy &:= \mathtt{sel}_A = (i, t) \wedge \varphi(i, i.data) = (o, \_) \\ o.data &:= \mathtt{sel}_A = (i, t) \wedge \varphi(i, i.data) = (o, d')\ ?\ d' : \perp \\ \text{where } t &= \langle s, s', \varepsilon, \varphi \rangle \end{aligned}$$

The automaton is ready to receive a packet from in-channel i if it is selected, and thus there exists some enabled transition consuming a packet from i. The automaton is ready

to transmit a packet on out-channel o if it has selected some transition t whose transformation produces a packet at channel o. The data produced by the automaton is set only when a transition has been selected. In that case, it takes the value as dictated by the transformation. In other cases, it will have default value $\perp$.

3. Deadlock Detection

Gotmanov et al. propose a deadlock detection technique for XMAS networks based on *block and idle equations* [3]. This technique reduces the the problem of finding a deadlock to an SMT instance. A channel is permanently blocked (idle) if its trdy (irdy) signal is permanently low. For example, the block of an input of a fork is defined as the block of its output and the idle of the other input. Let d be a packet and let q be a queue. The block (idle) of the in (out) channel of queue q are defined as:

$$\begin{aligned}\mathbf{block}(q.in, d) &= \#q = q.size \wedge \\ &\qquad \exists_{d' \in \tau(q.out)} \cdot \mathbf{block}(q.out, d') \\ \mathbf{idle}(q.out, d) &= \#q.req = 0 \wedge \mathbf{idle}(q.in, req)\end{aligned}$$

Here, function τ takes as input a channel and returns an overapproximation of the packets that can possibly be in that channel. We will use the term *colors* for this, in the same fashion as colored Petri nets. The block equation states that a request is blocked if and only if the number of packets in the queue is equal to its size and if there exists a packet (occupying the head of the queue) that is permanently stuck. The idle equation states that the channel is idle for requests if and only if there are no requests in the queue and additionally a request will never arrive.

We define block and idle equations for XMAS automata. To this end, we first define an equation formulating when an XMAS automaton is in a deadlock. This equation introduces a new type of variable of the form $A.s$. This variable is either 0 or 1 depending on whether automaton A is in state s.

An automaton A can be in a deadlock if there exists a state for which all outgoing transitions can be dead, i.e., permanently not enabled. Whether a transition is dead depends on the packets at the in-channels of the automaton. Consider, for sake of simplicity, an event ε that is only true for packet d at in-channel i. Let (o, d') be the result of the transformation. The transition is dead if and only if channel o is permanently unable to accept packet d', *or* if packet d will never arrive at channel i at all. The first case can be formulated as the block of o for d'. The second case as the idle of i for d. This yields the following equation:

$$\begin{aligned}\mathbf{deadA}(A) = \exists_s \cdot A.s \wedge \forall_{t=\langle s,s',\varepsilon,\varphi\rangle} \cdot \\ \forall_i \cdot \forall_{d \in \tau(i)} \cdot \\ \varepsilon(i,d) \rightarrow (\mathbf{block}(\varphi(i,d)) \vee \mathbf{idle}(i,d))\end{aligned}$$

where $\mathbf{block}(\perp) = False$

When a packet arrives at the in-channel of an XMAS automaton, it can be permanently blocked because the automaton is in a deadlock. It is also possible that the automaton simply does not accept the packet at all. Similarly, an out-channel is idle for some packet if the automaton is in a deadlock or if the automaton never produces the packet in the first place. The block and idle equations for an XMAS automaton are therefore:

$$\begin{aligned}\mathbf{block}(i,d) &= (\forall_{t=\langle s,s',\varepsilon,\varphi\rangle} \cdot \neg\, \varepsilon(i,d)) \vee \mathbf{deadA} \\ \mathbf{idle}(o,d') &= (\forall_{t=\langle s,s',\varepsilon,\varphi\rangle} \cdot \forall_i \cdot \forall_{d \in \tau(i)} \cdot \\ &\qquad \varepsilon(i,d) \rightarrow \varphi(i,d) \neq (o,d')) \vee \mathbf{deadA}\end{aligned}$$

An unfolding of these equations yields an overapproximation of the possible deadlocks. When applied to the running example, this yields two deadlock candidates. First, when the queues are empty and the automata are in state $\langle s_1, t_0 \rangle$. Secondly, when q_0 is filled with reqs, q_1 is filled with acks and the automata are in state $\langle s_0, t_1 \rangle$. Both deadlocks are unreachable and are ruled out by the invariant presented in Section 1.

4. Deriving Invariants

Chatterjee and Kishinevsky propose a method for generating inductive invariants over XMAS fabrics [4]. Their method is based on the notion of flows. Consider, e.g., the flow of packets d passing through a queue q with in-channel i and out-channel o. The technique of Chatterjee and Kishinevsky is based on the intuition that the number of times packets of color d pass through channel i is always equal to the number of times packets of color d pass through channel o minus the number of packets d that are stored in the queue. More formally, let λ_i^d denote the number of clock ticks in which $i.irdy \wedge i.trdy \wedge i.data = d$. Moreover, let $\#q.d$ denote the number of d-colored packets in queue q. Then the following is invariably true: $\lambda_i^d = \lambda_o^d - \#q.d$.

For each primitive, an invariant is defined. For example, a fork induces the following invariant: $\lambda_i^d = \lambda_a^d = \lambda_b^d$. These invariants form a matrix. This matrix is reduced to row echelon form using Gaussian Elimination. All λ-variables are swept away, yielding invariants solely concerning relations between the number of packets in queues.

We extend the methodology of Chatterjee and Kishinevsky by adding invariants for XMAS automata. This requires additional variables. First, variables of the form $A.s$, with A an automaton and s a state, are integer variables with value 0 or 1. Secondly, we introduce variables of the form κ_A^t. Here, A is an automaton and t is a transition. Variable κ_A^t represents the number of times transition t has fired, i.e., the number of times that $\mathtt{sel}_A = (\_, t)$.

The first invariant simply states that all automata A are in exactly one state, i.e., $\sum_{s \in S} A.s = 1$.

Let A be an automaton and s a state. The second invariant relates the number of times the ingoing transitions of s fire to the number of times the outgoing transitions of s fire. Consider the case where automata A is *not* in state s. Each time an ingoing transition has fired must have been matched by the firing of one outgoing transition. If automata A *is* in state s, the same holds up to the firing of one outgoing transition. This holds for any state, except for the initial state. Thus, the following invariant holds (proofs

of soundness of the invariants are straightforward and have been left out due to space restriction):

$$\sum_{t \in \langle \_, s, \_, \_ \rangle} \kappa_A^t = \Big(\sum_{t \in \langle s, \_, \_, \_ \rangle} \kappa_A^t \Big) + A.s - (s = s_0) \quad (1)$$

Example 1. Consider automaton S of the system in Figure 1. For sake of presentation, we use $\#req!$ and $\#ack?$ instead of κ-variables, to denote the number of times these transitions fired. Equation 1 induces the following equations: $\#req! = \#ack? + S.s_1$ and $\#ack? = \#req! + S.s_0 - 1$.

The third invariant relates the number of times packets arrive at in-channels of automaton A to the number of times transitions fire. To this end, we consider the set $\mathcal{I}$ of all tuples (i, d), with i an in-channel of A and $d \in \tau(i)$ a color of that channel. We partition this set in such a way that whenever two tuples can both enable some transition t, they are put in the same subset. Let $\sim$ denote the equivalence relation induced by the partition. Then the following holds for all tuples:

$$(\exists_{t=\langle s, s', \varepsilon, \varphi \rangle} \cdot \varepsilon(i, d) \wedge \varepsilon(i', d')) \implies (i, d) \sim (i', d')$$

Different partitions satisfying this criterion exist, but there is exactly one partition that is the most fine-grained, i.e., that contains the largest number of subsets. Let $\mathcal{I}_\sim$ denote that partition. Let $I \in \mathcal{I}_\sim$ be one of the subsets, i.e., a set of tuples of in-channels with colors. Each tuple can enable one or more transitions. The set of transitions enabled by set I is defined as:

$$T(I) \stackrel{\text{def}}{=} \{\langle s, s', \varepsilon, \varphi \rangle \mid \exists_{(i,d) \in I} \cdot \varepsilon(i, d)\}$$

Let $(i, d) \in I$ be a tuple. Whenever a packet d flows through in-channel i, one of the transitions in $T(I)$ fires. The third invariant states that for any equivalence class I:

$$\sum_{(i,d) \in I} \lambda_i^d = \sum_{t \in T(I)} \kappa_A^t \quad (2)$$

The number of times packets flow through one of the in-channels in subset I is equal to the number of times a transition fires that can be enabled by one of the in-channels in I.

Example 2. Consider again automaton S in Figure 1. It has two in-channels, each of which has only one color. Thus, $\mathcal{I} = \{(\text{src.out, token}), (q_1.\text{out, ack})\}$. This set is partitioned into two equivalence classes, since each enables a different transition. The first enables the transition injecting requests. The second enables the transition consuming acks. Equation 2 therefore induces the following equations: $\lambda^{\text{token}}_{\text{src.out}} = \#req!$ and $\lambda^{\text{ack}}_{q_1.\text{out}} = \#ack?$.

Finally, a fourth invariant relates the number of times packets are produced at the out-channels of automaton A to the number of times transitions fire. This is very similar to Equation 2, but instead of partitioning in-channels based on whether they can enable the same transitions, we partition tuples (o, d') of out-channels and packets based on whether they share transitions that can produce packet d' at channel o.

We add these equations to the matrix, sweeping κ-variables away while keeping $A.s$ variables. This yields invariants relating states of the automata to the contents of the queues in the communication fabric.

5. Case Study

We present a case-study that includes a cross-layer deadlock. In the example, we put an abstract version of a directory-based MI protocol on a 2D mesh with XY routing. We assume store-and-forward switching, so that a queue with queue size n is able to store n complete packets. For the sake of presentation, we have initially omitted as much as possible, such as data transfer, cache-to-cache forwarding, nacks and virtual channels (VCs). We also present a version including all these facets. We assume there is exactly one directory and will vary its position.

The resulting protocol has been proven deadlock-free for bus-based or point-to-point communication using UPPAAL. A 2D mesh with XY routing is well-known to be deadlock-free. The composition, however, may have deadlocks, regardless of the use of virtual channels. We show that deadlock freedom depends on whether the queue sizes are "sufficiently large". Whether they are sufficiently large depends on both size of the mesh and the location of the directory.

Figure 2 shows the protocol. On a load or store miss, the cache sends out a get signal to access a block of cache (data or instruction). When it gets an invalidate from the directory or when a $repl$acement is triggered from the core itself, it flushes the block of cache and notifies the directory via a put. Then, it goes to intermediate state MI to wait for an acknowledgment. The directory waits for a get to go from I to M and stores the cache that owns the block. The M and MI states are thus parameterized with the owner c. The directory may decide at any time to send an invalidate to the owner. Whenever it receives a put from that cache, it will return to its initial state. When a packet can currently not be consumed, it is stalled and moved to the end of the queue.

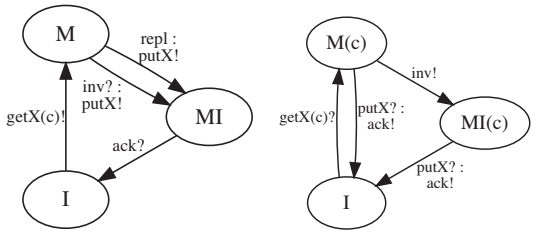

Figure 2. Artificial protocol

Figure 3 describes a cross-layer deadlock in a 2×2-mesh with all queue sizes equal to 2. Cache $(0, 0)$ has sent consecutively a get and a put to the directory. The directory,

being in a state where cache $(1,0)$ owns the block, cannot accept these as it is waiting for the space to inject an inv destined for $(1,0)$. However, the queue leading to that cache is already filled with such invs. Cache $(1,0)$ cannot accept an inv, since in order to accept it, it must inject a put.

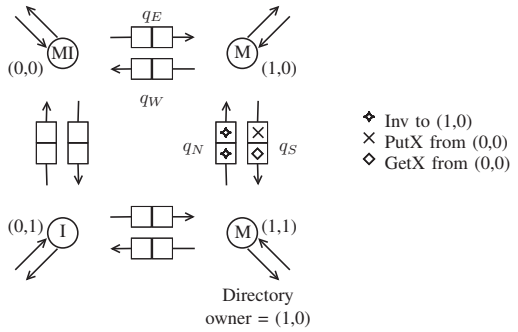

Figure 3. Cross-layer deadlock

If the queue sizes would have been 3 or more, the system is deadlock-free. First, note that due to XY routing, the packets occupying q_S can only be emitted from either $(0,0)$ or $(1,0)$. Secondly, a cache can only once emit a get/put before it gets stuck in its MI-state waiting for an ack of the directory. Thus, in the deadlock in Figure 3, if q_S would have had a third space, that space would not be occupied by any packet from $(0,0)$. Therefore, cache $(1,0)$ now is able to consume an inv and emit a put. Then, the get/put of cache $(0,0)$ in q_S would be stalled and the new put from $(1,0)$ would overtake them; the deadlock is resolved. Figure 4 shows the queue sizes necessary for deadlock freedom for different positions of the directory.

3	3
3	3

23	23	23	23
15	15	15	15
15	15	15	15
23	23	23	23

39	39	39	39	39
29	29	29	29	29
19	19	19	19	19
29	29	29	29	29
39	39	39	39	39

Figure 4. Queue sizes found by ADVOCAT for different mesh sizes and directory positions. For example, in a 4×4-mesh, when the directory is at $(1,1)$, queue sizes have to be at least 15.

Experimental Results. Consider a 2×2-mesh where the directory d is at the lower-right node. The following invariants are found automatically. They concern the left-upper cache $(0,0)$, here denoted with c:

$$1 = q_E.getX(c) + q_S.getX(c) + q_N.ack(c) + q_W.ack(c) + c.\mathrm{I} + d.\mathrm{M}(c) + d.\mathrm{MI}(c) \quad (3)$$

$$d.\mathrm{MI}(c) = q_E.putX(c) + q_S.putX(c) + q_N.ack(c) + q_N.ack(c) \quad (4)$$

The first invariant shows that whenever a $getX$ is en route from the cache to the directory, there is no corresponding ack en route, and the other way around. Whenever either a $getX$ or an ack is en route, the cache cannot be in state I and the directory cannot be in a state where cache c already owns the cache. Moreover, there can be at most one $getX$ or one ack en route (but not both at the same time). When the cache is in state I or when the directory is in a state where cache c owns the block, no $getX$ or ack is en route at all. Finally, it cannot be the case that the cache is in its I-state, while the directory is in a state where the cache owns the block of data.

The second invariant shows that there is either a $putX$ or an ack en route, if and only if the directory is in state MI and cache c owns the block of data. Similar invariants are found for the other caches, yielding 6 invariants in total for three caches. These invariants are sufficient for proving deadlock freedom.

A common approach to resolve deadlocks is to add virtual channels for different message types [5]. The deadlock as described above, however, cannot be resolved this way. The use of VCs does allow for smaller queue size. For example, a 6×6-mesh is deadlock-free for VC-sizes of greater than 29; without VCs the queues have to be of size 58.

A total verification effort – including τ-derivation, invariant generation, and a proof of deadlock freedom – for a 6×6-mesh with VCs and queue size 30 takes 67 seconds[1]. This example includes 2844 primitives, 36 automata and 432 queues. Note that the verification time of our algorithm does not depend on the queue size.

MI Protocol. We have adopted a version of the MI cache coherence protocol inspired on the version modelled using the GEM5 simulator [6]. This cache coherence protocol supports cache-to-cache transfer and includes the modelling of DMA accesses. It has been modified to exclude the deadlock described above. The communication of acking/nacking a replacement and the notification upon receiving data are added. The L2 cache model contains 5 states, the directory $4+n$ (with n the number of caches) and there are 8 different types of messages, each parameterized with destination and source nodes.

In a 2×2 setting, 14 invariants are found, of varying complexity. An example of an invariant is that when all automata are in their I state, there are no en route acks, no en route forwarded requests and no replacements. As another example, let C denote the set of caches, let d denote the directory and let $|invs|$ ($|acks|$) denote the number of en route invs and acks respectively. The following invariant is found:

$$\sum_{c \in C} c.\mathrm{MI} - d.\mathrm{MI} = |acks| - |invs|$$

Consider the case where the directory is in state MI but no cache is. There must be exactly one en route inv. When this is consumed, one cache goes to its MI state, meaning that the LHS equates to 0. This implies that at that point no acks or invs are en route.

1. All results have been obtained on a 2 GHz Intel Core i7, with 4GB of memory. All source code, UPPAAL models and case studies can be found online at: www.cs.ru.nl/~freekver/DATE16/

We have verified this protocol for all meshes up to 5×5. When queue sizes are too small and a cross-layer deadlock occurs, this is found in 32 minutes for a 5×5-mesh. A proof of deadlock freedom takes 56 minutes.

6. Related Work

Formal methods have played an important role in the design and verification of cache coherence protocols [7]. Various formal verification methodologies target cache coherence, using, e.g., Murphi [8], [9], SMV [10], [11], or protocol message flow charts [12]. German provided the German protocol as a challenge for *parametric* and *automated* verification [13]. Chou et al. present a semi-automated method for parametric verification based on model checking and a form of assume-guarantee reasoning [14]. Generally, however, these works decouple verification of the interconnect from verification of the protocol. They often assume a bus-based architecture or networks with VCs for all message types.

Zhang et al. derive informal design guidelines that ensure that parametric verification is possible [15]. These guidelines state, e.g., that all the nodes should be identical and that queue sizes should be independent of the number of nodes. Our methodology requires no adherence to such guidelines, at the price of not being parametric in the number of nodes.

The problem of cache coherence verification on the architectural level has been addressed in [16]. That approach uses the existing regular XMAS primitives to model cache coherence protocols. This requires manual, cumbersome and error-prone modelling. It is, e.g., hard to model snooping behavior using the existing XMAS primitives.

7. Conclusion

This paper targets cross-layer verification, by combining IO state automata for modelling application-level protocols with XMAS primitives for fine-grained modelling of communication fabrics. A methodology for deriving cross-layer invariants has been implemented. These invariants are used to prove absence of protocol-level, network-level and cross-layer deadlocks. The whole methodology, coined `ADVOCAT`, is fully automatic. It is generally applicable to communication fabrics that include routing, virtual channels (or not), arbitration, broadcasting, adaptive routing, etc. It can deal with both directory-based and snoopy cache coherence protocols. Our methodology is sound but subject to false negatives.

We applied `ADVOCAT` to an extensive case-study, where cross-layer deadlocks are possible when queue sizes are too small. `ADVOCAT` was used to derive the minimally possible queue sizes, which depend both on protocol- and fabric-related facets (the routing algorithm, the size of the mesh).

Challenging future work consists of the derivation of invariants that provide information on the order of arrival of en route packets. This will lead to more accurate and informative feedback in case deadlocks are found and can also decrease the number of false negatives. In the near future, we aim to distill IO automata models from GEM5 descriptions. Eventually, we aim to have a fully automated formal verification approach directly applicable to GEM5 cache coherence system architectures.

References

[1] S. German, "Tutorial on verification of distributed cache memory protocols," in *Formal Methods in Computer-Aided Design (FMCAD'04)*, 2004.

[2] C. Baier and J.-P. Katoen, *Principles of model checking*. The MIT Press, 2008.

[3] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, "Verifying deadlock-freedom of communication fabrics," in *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, 2011, vol. 6538, pp. 214–231.

[4] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," in *Computer Aided Verification (CAV'10)*, July 2010.

[5] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *Computers, IEEE Transactions on*, vol. 100, no. 5, pp. 547–553, 1987.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The GEM5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[7] S. M. German, "Formal design of cache memory protocols in IBM," *Formal Methods in System Design*, vol. 22, no. 2, pp. 133–141, 2003.

[8] J. W. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol verification using flows: An industrial experience," in *FMCAD*, 2009, pp. 172–179.

[9] 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*, vol. 36, pp. 37–64, 2010.

[10] K. McMillan, "Parameterized verification of the flash cache coherence protocol by compositional model checking," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, 2001, vol. 2144, pp. 179–195.

[11] E. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *Correct Hardware Design and Verification Methods*, 2003, vol. 2860, pp. 247–262.

[12] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Formal Methods in Computer Aided Design (FMCAD'08)*. IEEE Press, 2008, p. 10.

[13] A. Pnueli, S. Ruah, and L. D. Zuck, "Automatic deductive verification with invisible invariants," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'01. London, UK, UK: Springer-Verlag, 2001, pp. 82–97.

[14] C. tsun Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Formal Methods in Computer Aided Design (FMCAD'04)*. Springer, 2004, pp. 382–398.

[15] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin, "PVCoherence: Designing flat coherence protocols for scalable verification," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 392–403.

[16] F. Verbeek and J. Schmaltz, "Towards the formal verification of cache coherency at the architectural level," *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 3, p. 20, 2012.