Compositional Performance Verification of Network-on-Chip Designs

Daniel E. Holcomb, and Sanjit A. Seshia, *Senior Member, IEEE*

Index Terms—Verification, Quality of Service, On-chip Networks.

Abstract—This work presents a compositional approach to formally verify quality-of-service (QoS) properties of network-on-chip (NoC) designs. A major challenge to scalability is the need to verify worst-case latency bounds for hundreds to thousands of cycles, which are beyond the capacity of state-of-the-art model checkers. The scalability challenge is addressed using a compositional model checking approach. The overall latency bound problem is divided into a number of smaller sub-problems, termed *latency lemmas*. The sub-problems imply the overall latency bound, but are easier to prove on account of being inductive. A method is presented for computing these lemmas based on the topology of the network and a subset of relevant state, and the latency lemmas are verified using k -induction. The effectiveness of this compositional technique is demonstrated on illustrative examples and an industrial ring interconnection network. In the ring network, a latency bound that cannot be verified in 10,000 seconds without lemmas is proved inductively in just 75 seconds when the lemmas are used.

I. INTRODUCTION

Network on chip (NoC) is a paradigm for communication within large many-core system on chip (SoC) designs. An NoC architecture is a network of interconnected nodes, where each node has networking logic and is associated with a processor core, memory controller, specialized IP block, etc. Industrial examples include the Tiler TILE64TM processor [1], STI Cell BE [2], and Intel Larrabee [3]. NoCs typically offer quality of service (QoS) guarantees on worst-case latency, jitter, and throughput for some classes of network traffic. QoS violations are often caused by specific inputs, and can be easily missed in performance simulation.

High-level modeling of NoCs [4], [5], [6], [7], [8] and automatic abstraction [9] help to hide unnecessary details, easing the way for formal analysis. Even so, verifying QoS properties can be challenging for industrial NoC designs, both due to the scale of the design and the property to be verified. Consider for instance the problem of proving an upper bound on the latency of sending a packet from one node in the network to another. In principle, this property can be expressed in linear temporal logic (LTL), and the problem can be solved using model checking. The LTL property expresses a *bounded liveness property*, written in English as “every packet from source A gets to its destination B within N cycles.” Bounded liveness is equivalent to a safety assertion where one adds some extra logic to track the progress of time. One can use state-of-the-art model checking strategies

such as k -induction [10], interpolation [11], and IC3/property-directed reachability (PDR) [12], [13] to verify this property. However, regardless of the strategy, it is generally necessary to analyze at least N consecutive cycles to either prove or disprove the latency bound, assuming that N is tight. Typical latency bounds for NoCs can be in hundreds or thousands of clock cycles. Unrolling of model transition relation to such depth is beyond the capacity of state-of-the-art model checking engines. Property-directed reachability [12], [13], while avoiding explicit unrolling of the transition relation, still does not scale past tens of clock cycles, as will be shown in Sec. VI and Sec. VII.

This paper addresses the scalability challenge using a popular approach in formal verification: *compositional reasoning*. In compositional reasoning, one breaks up the overall monolithic proof obligation into a number of “smaller” proof sub-goals, which are much easier to verify, such that if all of the sub-goals are proved, then so is the original monolithic property. The approach may involve decomposing the system description, the property, or both. The key is to devise a decomposition that is well-suited to the verification task at hand. When the overall property is a latency bound of N cycles through a network, a natural decomposition is to prove smaller bounds on a packet’s progress through particular subpaths of the network. These proof sub-goals are termed *latency lemmas*, and the core contributions of this paper are methods to discover and apply them.

Specifically, this work shows that for some common network topologies, one can enumerate finitely many *stages* that a packet can go through. Each location in the network belongs to at least one stage at every time moment. Stages are arranged into a directed, acyclic *stage graph*, to capture the order in which they can be visited by a packet. A stage graph is defined through the use *age lemmas* that bound the total time between when a packet is injected into the network and when it exits each stage. The age lemmas are in turn created through the use of *progress lemmas*, which bound the number of cycles that a packet can spend in each stage. Age lemmas and progress lemmas are collectively referred to as *latency lemmas*, and by proving the latency lemmas one proves bounds corresponding to all paths through the network.

To summarize, this paper makes the following novel contributions:

- A compositional approach to proving latency bound properties in NoCs by decomposition into latency lemmas.
- Methods of formulating latency lemmas for a particular type of microarchitectural descriptions using a *stage graph*. The stage graph formulation is automated for acyclic networks.
- Experimental results on illustrative examples showing that the proposed technique can reduce the runtime of

Authors are with the Department of Electrical Engineering and Computer Sciences, University of California Berkeley, Berkeley, CA, 94702 USA email: holcomb@eecs.berkeley.edu, ssesia@eecs.berkeley.edu.

Copyright (c) 2014 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

inductive verification of latency bounds by 20-50x, and causes k-induction to verify latency bounds 4-10x faster than can be achieved with the state-of-the-art IC3/PDR technique using all of the same strengthenings. Furthermore, it is demonstrated that verification runtime can be traded off against tightness of proved latency bounds.

- Experimental results on an industrial-style ring interconnection network showing that latency lemmas give a speedup of greater than 50x, and in all cases allow latency bounds to be proved inductively. For an 8-agent ring, latency is verified using k-induction in 75 seconds with latency lemmas, and cannot be verified in 10,000 seconds without them.

While the approach given in this paper is shown to be efficient for proving latency bounds, two limitations should be noted. The first limitation is that it assumes a network to be primarily described using a particular microarchitectural modeling language called xMAS (see Sec. II). The second limitation is that the approach is not automated for cyclic networks. Yet, the ideas presented in this work are more general than the specific implementation given, and can be applied to most latency verification works that are based on model checking.

The remainder of the paper is organized as follows. Sec. II introduces basic terminology and sketches the compositional latency verification approach using an example. Sec. III presents notation. Sec. IV describes the compositional approach formally and in more detail, including rules for creating the stage graph. Sec. V presents methodology used to evaluate the approach, and demonstrates efficient encoding of packet ages. Results for illustrative examples are presented in Sec. VI, and for the ring network in Sec. VII. Related work is presented in Sec. VIII and Sec. IX concludes. This paper is a significantly extended and revised version of a conference paper [14].

II. PRELIMINARIES

A. Background

In this work, NoC designs are described using a high-level modeling formalism called executable micro-architectural specifications (xMAS models) [5]. The motivation for xMAS is to model network microarchitectures in a way that is expressive enough to capture interesting behaviors, yet sufficiently regular to enable formal reasoning. Toward this goal, an xMAS model $\mathcal{N}$ is a composition of simple kernel primitives (Fig. 2) connected via communication interfaces known as channels. Each channel c comprises three signals: $c.data$ and $c.irdy$ controlled by the initiator primitive, and $c.trdy$ controlled by the target primitive (Fig. 1). A packet with value $c.data$ is transferred from initiator to target when $c.irdy$ and $c.trdy$ are both asserted in the same cycle. Each channel is specified to exclusively communicate either data packets or dataless tokens; the difference is that in the former case $c.data$ encodes information such as destination address, and in the latter case $c.data$ is ignored. A channel c is said to be blocked (by the target) when $c.irdy$ is asserted and $c.trdy$ is not. A channel c obeys a liveness bound x if the temporal logic formula $\mathbf{G}(c.irdy \implies \mathbf{F}^{\leq x} c.trdy)$ holds, where $\mathbf{G}$ is the temporal operator “Globally” and $\mathbf{F}$ is “Eventually”. In other words, x is the largest number of consecutive blocked cycles on channel c ; a liveness bound of $x = 0$ means that a channel never blocks.

Transfer attempts are persistent, meaning that if a transfer is blocked on a channel c , $c.irdy$ remains asserted until the block is resolved and the eventual transfer occurs [15].

Fig. 1: Channel c expanded into constituent signals.

The stateful xMAS primitives are implemented as finite state machines, and the stateless as combinational logic. The network has a single initial state in which all queues are empty. Each data queue comprises one or more queue slots, and every data queue slot in network $\mathcal{N}$ is indexed by a unique identifier i . To save space, only brief descriptions of the xMAS components are provided here. The interested reader is referred to papers by Chatterjee *et al.* [5], [7]. This work adopts the convention that data transformations are always unary, as in the so-called restricted primitives of previous work [5]. Under this assumption, deviating from the original xMAS descriptions [5], it suffices to have the function primitive be the only one that transforms data.

- 1) *Queue*: parameterized by its depth (number of slots). Data packets or tokens are read from the fixed head slot, and written to a tail position that varies with the number of packets stored in the queue. When a packet is read from the head slot, all other packets in the queue advance by one slot toward the head.
- 2) *Function*: transforms input data on i to output data on o using a deterministic function that is a parameter f of the primitive. The function primitive directly connects the *irdy* and *trdy* signals from its input channel and output channel, and therefore is transparent with respect to timing.
- 3) *Data or Token Source*: a source sends packets through its output channel o . A *Data Source* non-deterministically decides when to send a packet, and the data of said packet is also non-deterministic. A *Token Source* is eager and attempts to send a token on every cycle.
- 4) *Data or Token Sink*: a sink consumes packets from a channel i . A *Data Sink* non-deterministically decides when to consume a waiting data packet within x cycles, where x is a parameter of the sink¹. A *Token Sink* eagerly consumes tokens from channel i .
- 5) *Fork*: a synchronization primitive that consumes a data packet or token from i and produces a packet of the same type on b , as well as a token on a .
- 6) *Join*: a synchronization primitive that consumes a token from a and a data packet or token from b , and outputs on o a packet of the same type as b .
- 7) *Switch*: a routing primitive parameterized by a switching function $f : i \mapsto \mathbb{B}$, a switch consumes a data packet from i and produces it on a if $f(i) = \mathbf{true}$, and on b otherwise.
- 8) *Merge*: arbitration primitive that consumes a packet from either a or b , and produces the same packet on o . A state

¹Sinks are assured of satisfying liveness bounds by their transition relations; if the blocking condition $i.irdy \wedge \neg i.trdy$ has been **true** for the past x cycles, then the sink is forced to assert $i.trdy$. Note that this constraint on sink behavior differs from that of Chatterjee *et al.* [5], but a known bound on sink blocking is a necessary condition for the existence of a finite latency bound. Chatterjee’s work doesn’t require this because it is proving liveness instead of latency.

bit u stores the arbitration priority among the inputs, and its updation ensures local fairness.

Fig. 2: The set of XMAS primitives [5]. Inputs and outputs are written in normal font and the parameters are in bold.

B. Sketch of Latency Lemmas

To make latency verification more efficient, this work adds two kinds of latency lemmas, termed progress lemmas and age lemmas. An informal sketch here shows how each type of lemma is derived and applied, using a credit loop network as an illustrative example. A more detailed description of progress and age lemmas will follow in Sec IV.

1) Description of Credit Loop Network: The credit loop network in Fig. 3, adapted from work by Chatterjee *et al.* [6], implements credited flow of data packets from a master agent to a target agent. The network has three queues: “available tokens” in the master, and “outstanding credits” and “ingress” in the target. The ingress queue stores data packets, and the other two store tokens. The master’s source non-deterministically injects data packets on channel a . The data packets consume one available token when propagating through the join and onward to the ingress. Packets stored in the ingress can be consumed only if there are outstanding credits, and if so the data sink on channel d cannot block progress for more than 5 cycles. Whenever the data sink consumes a packet from the ingress, an outstanding credit is also consumed by the token sink. The token source simultaneously adds tokens to both the available tokens and outstanding credits queue in every cycle when both have free space.

2) Progress Lemmas – Bounds on Time to Make Progress:

The first step toward proving an end-to-end latency bound is to compute a conservative upper bound on how long a packet might wait to advance once inside the ingress queue. In doing so one can assume that the ingress holds at least one data packet (i.e. $n_1 \neq 0$) and reason about different conditions for the rest of the network.

- (a) If a token is in the outstanding credits queue, then signal $d.irdy$ is asserted and the data sink may at any time consume the data packet while the token sink consumes the outstanding credit. The liveness bound of the data sink limits it to 5 cycles of blocking, so $c.trdy$ occurs within 5 cycles from states satisfying $n_2 \neq 0$.
- (b) If the outstanding credits queue is empty, and available tokens queue is not full, then the token source will add an outstanding credit, causing $d.irdy$ to be asserted in the next cycle. The sink can only block for 5 cycles once $d.irdy$ is asserted, so in the worst case $c.trdy$ will occur within 6 cycles from states satisfying $n_2 = 0 \wedge n_0 \neq 2$.
- (c) If the outstanding credits queue is empty, and the available tokens queue is full, then no token can ever be injected into the outstanding credits queue, and the packet in the ingress will never advance.

Fig. 3: Credit loop network $\mathcal{N}$, with stage graph $\mathcal{G}$ above it. The dashed arrows show the automatically-derived correspondence between queue slots in $\mathcal{N}$ and stages in $\mathcal{G}$. Bold channels in $\mathcal{N}$ carry data packets and the rest carry tokens. The n_i associated with each queue is the variable representing the number of packets in it. The first number above each stage in $\mathcal{G}$ is an upper bound on the time spent in the stage, and the second is an upper bound on the age of a packet in the stage.

Next the approach assumes the network is deadlock-free, and moreover hypothesizes that every reachable network state in which a packet awaits progress will satisfy either (a) or (b) described above. This hypothesis is formalized as the candidate inductive invariant $\theta_{c,STATE}$ (given by Eq. 1 for the credit loop). If $\theta_{c,STATE}$ is valid, and all reasoning is sound, a packet in the ingress should always make progress in, at-worst, the 6th future cycle (denoted by $\delta_{abs} = 6$ in Fig. 3), and property $\theta_{c,TIMING}$ (Eq. 2) explicitly checks this on channel c .

Properties $\theta_{c,STATE}$ and $\theta_{c,TIMING}$ are collectively called *progress lemmas*. The progress lemmas are conservative and they over-approximate reachable state. As will be shown later (by Eq. 15 in Sec. VI-B), the condition ($n_2 = 0$) is unsatisfiable when the ingress contains packets, so condition (b) discussed above is unachievable.

$$\theta_{c,STATE} := c.irdy \implies (n_2 \neq 0) \vee (n_2 = 0 \wedge n_0 \neq 2) \quad (1)$$

$$\theta_{c,TIMING} := c.irdy \implies \mathbf{F}^{\leq 6} c.trdy \quad (2)$$

3) Age Lemmas – Bounds on Time Since Injection: If the queue slots are visited in a known order, and the progress lemmas provide a way to bound the time spent at each slot, then it becomes possible to compute a bound on the total propagation delay through the network. The credit loop has an obvious ordering among queue slots, as a packet injected from the data source first occupies the tail of the ingress (or bypasses it), then the head of the ingress, then reaches the sink. The progress lemmas assert that channel c will transfer any waiting packet in the 6th future cycle in the worst case. This means that a packet will advance every 7 cycles. If a packet cannot spend more than 7 cycles in either ingress slot, an age bound of 8 cycles is implied for the ingress tail slot, and 15 cycles for the ingress head slot; these specialized bounds are called *age lemmas*, and formulated using a *stage graph* as shown at the top of Fig. 3.

It is clear that the total latency is bounded by 15 if the age lemmas can be proved, yet including the latency lemmas makes the 15 cycle bound compositional and easier to verify

using k-induction. Proving the 15 cycle latency bound requires an induction depth of 13 frames without latency lemmas, versus just 8 frames with them; the reduced induction depth translates to a 2x speedup in this case. In general, the induction depth required to prove a latency bound property without the lemmas is proportional to the total latency, while induction depth to prove the same bound using lemmas is proportional to the time for a packet to make progress. The speedup from using latency lemmas is therefore more pronounced when proving large latency bounds, as will be shown in Sec. VI and VII.

III. FORMALISM

As sketched in the previous section, a set of conjectured latency lemmas makes it possible to efficiently verify a possibly loose bound on the worst case end-to-end latency from any source in the network to any sink. The model being verified is an xMAS model $\mathcal{N}$. Every data-carrying queue slot (i.e. the slots belonging to queues where input and output channels carry data packets instead of tokens) in the network is assigned a unique index i , and variable q_i refers to the content of the i^{th} queue slot. A latency bound is translated to a simple safety property by checking the *age* of a packet in each cycle. The age of the packet in slot i is denoted $age(q_i)$ and is stored using specification variables².

A. Checking Cumulative Latencies as Age Bounds

The mapping from queue slots in $\mathcal{N}$ to stages in $\mathcal{G}$ can depend on the state of $\mathcal{N}$. This allows the same queue slot to represent different stages of progress depending on certain aspects of network state. The possible mapping from the i^{th} queue slot to the j^{th} stage is defined by a formula $p_{i,j}$; the i^{th} slot maps to the j^{th} stage whenever $p_{i,j}$ is **true**.

$$p_{i,j} : Q_i \times W \mapsto \mathbb{B} \quad (3)$$

- Q_i is the set of states of slot i ; q_i denotes a state of Q_i .
- W is the set of states for selected global variables including the number of items in any queue and the status of reservations in Sec. VII; w denotes a state of W .

A few special cases are worth mentioning. If a slot i never maps to stage j , then $p_{i,j} = \mathbf{false}$ regardless of q_i and w . If slot i always maps to stage j , then $p_{i,j} = \mathbf{true}$, regardless of q_i and w .

Each stage j in the stage graph $\mathcal{G}$ has an associated t_j that is a claimed upper bound on the age of any packet that maps to the stage. An age lemma for the i^{th} slot and j^{th} stage of progress is written as $\phi(i, p_{i,j}, t_j)$ (Eq. 4). For each slot i , assume the existence of a specification variable $used_i$ that is **true** in every state where slot i stores a packet. The property $\phi(i, p_{i,j}, t_j)$ checks that, whenever the network state satisfies $p_{i,j}$, any packet stored in slot i must have been injected into the network less than t_j cycles prior.

$$\phi(i, p_{i,j}, t_j) := used_i \wedge p_{i,j} \implies age(q_i) < t_j \quad (4)$$

²Specification variables are defined here as variables in $\mathcal{N}$ that record expressions over system variables but do not influence them.

Let Φ^L denote a property that is **true** in a state of $\mathcal{N}$ if the age lemmas for all progress stages hold.

$$\Phi^L := \bigwedge_{i \in [0, M-1], s_j \in \mathcal{G}} \phi(i, p_{i,j}, t_j) \quad (5)$$

Let Φ_t^G denote the property for a global latency bound of t . The global bound differs from stage bounds in that it is checked on all used queue slots regardless of the state of $\mathcal{N}$.

$$\Phi_t^G := \bigwedge_{i \in [0, M-1]} used_i \implies age(q_i) < t \quad (6)$$

B. Auxiliary Invariants (Ψ)

An advantage of modeling microarchitectures using the xMAS formalism is automated invariant strengthening. The automatically generated invariants are unrelated to QoS, but are essential for verifying any type of property in xMAS networks because they prevent the verifier from exploring unreachable states that may include deadlocked states. The set of auxiliary invariants is denoted Ψ and comprises local invariants on queues [5], [7], persistency invariants on channels [15], and design-specific numeric invariants ψ_{num} [6], [16], [8]. Note that the design-specific numeric invariants used are automatically derived in earlier works [6], [8], and added manually in this work.

C. Proving a Latency Bound

The overall problem of proving a latency bound t is $\mathcal{N} \models \Phi_t^G$. With auxiliary invariants the problem becomes $\mathcal{N} \models \Phi_t^G \wedge \Psi$. Described in detail in the next section, this work further strengthens the problem using progress lemmas Θ and age lemmas Φ^L , such that the overall problem becomes $\mathcal{N} \models \Phi_t^G \wedge \Psi \wedge \Phi^L \wedge \Theta$. It will be shown that this property is compositional, and leads to inductive proofs with shorter induction depths and lesser runtimes.

IV. LATENCY LEMMAS

A distinguishing feature of this work is to strengthen overall latency properties using precise bounds termed age lemmas for different stages of progress arranged in a stage graph $\mathcal{G}$. Computing the amount of time that a packet can spend in each stage further requires computing transfer bounds for different channels in the network. Algorithms are given for automatically deriving age lemmas for a subset of possible xMAS networks, yet the power of age lemmas is more general than just the subset of designs that are handled automatically, as will be demonstrated in the ring interconnect example of Sec. VII. The remainder of this section first presents an automated approach for creating a stage graph, and then presents an automated approach for computing the transfer bounds that determine age bounds of each stage in the stage graph.

A. Generating Age Lemmas (Φ^L) using Stage Graph $\mathcal{G}$

A stage graph is a tool for constructing age lemmas that lead to compositional proofs of overall latency bounds. In each state of the network, every queue slot that stores a data packet maps to a stage in $\mathcal{G}$. The stage that a packet maps to determines a specialized age bound to check on the packet. Formally, a stage graph is an acyclic digraph $\mathcal{G} = (S, E)$ with vertices $s_0, s_1, \dots, s_{L-1} \in S$ called stages. A stage s_j has an associated lemma $\phi(i, p_{i,j}, t_j)$ asserting that any packet in slot i of $\mathcal{N}$ that maps to stage s_j in $\mathcal{G}$ must have an age less than t_j . The conjunction of all age lemmas is denoted Φ^L (Eq. 5).

Stage graph construction is automated for acyclic networks. Acyclic networks are those without cycles in data paths, where “data path” is defined as any sequence $c_0, c_1, \dots, c_N$ of data channels with each c_i and c_{i+1} being input and output channels of the same xMAS primitive. One can trivially check whether a network $\mathcal{N}$ is acyclic, for example by using depth-first search from each data channel. An automated approach for constructing stage graphs for acyclic networks is presented as a two step process: first creating stage graph topology, and second adding the age annotations to the stages.

1) *Creating Stage Graph Topology*: The queue slots of an acyclic network will always have a partial ordering with respect to when a packet can occupy them. The topology of the stage graph reflects this ordering. Each queue slot $(i; i \in [0, M-1])$ in network $\mathcal{N}$ maps to a stage $(s_j; j \in [1, M])$ in stage graph $\mathcal{G}$. The mapping from queue slots to stages is accomplished by setting $p_{i,j}$ to **true** for combinations of i and j where $i = j - 1$. A special source stage (s_0) is added for all data sources, and a sink stage (s_{M+1}) is added for data sinks. These source and sink stages are nonstandard in that no packets can ever map to them.

Edges in the stage graph reflect transitions that packets can make in $\mathcal{N}$. All data sources map to a stage s_0 , and all data sinks map to stage s_{M+1} . Stages s_x and s_y in the stage graph $\mathcal{G}$ are connected by an edge if the components (source, sink, or queue slot) mapping to s_x and s_y are adjacent slots within a single queue, or if there exists a queue-free data path in $\mathcal{N}$ from the first component to the second.

2) *Assigning Age Bounds to Stages*: Once the stages and edges of the stage graph are created, age bounds must be assigned to each stage. The first step towards this is to compute for each stage s_j , a value d_j that is the maximum residence time of the stage. Source stage (s_0) is assigned $d_0 = 1$, and sink stage (s_{M+1}) has $d_{M+1} = 0$; all other stages correspond to queue slots. For a stage (s_j) corresponding to a queue slot, the maximum residence time (d_j) cannot exceed 1 greater than the maximum blocking time of the channel that is the output of the queue containing this slot. While the maximum blocking time of each channel is not known a priori, the next subsection gives a way to compute an upper bound on it. For a channel c , the computed upper bound on blocking time is denoted δ_{abs} , and therefore $d_j = 1 + \delta_{abs}$.

Now that each stage in the network is assigned a residence time (d_j), age bounds for each stage are computed. The maximum age in any stage depends on the maximum residence time of that stage, and the maximum age of a packet when it enters the stage. The critical path for each stage s_j is therefore the path from s_0 to s_j in $\mathcal{G}$ for which the sum of d_j is largest. This path sum is denoted t_j and it is the age bound of stage

s_j .

3) *Global Age Bound T_L from Age Lemmas*: If every data packet in every reachable state of $\mathcal{N}$ maps to a stage in $\mathcal{G}$, then the largest t_j associated with any stage is a claimed global age bound for $\mathcal{N}$. Therefore, letting the largest t_j be denoted as T_L , $\Phi^L \implies \Phi_{T_L}^G$ provided that coverage of $\mathcal{G}$ is complete. T_L is often conservative for several reasons:

- The channel blocking bounds that are computed are conservative, causing the residence times of each stage to be over-approximated. This occurs in the credit loop (Sec. II-B) where the blocking time of packets in the ingress queue is overestimated.
- The stage graph conservatively over-approximates the connectivity of the network by ignoring logical propagation conditions.
- It may be impossible for any one packet to experience all of the the worst-case progress bounds, even if each is individually achievable.

4) *Cyclic vs. Acyclic Networks*: No automated procedure is given to construct a stage graph for cyclic networks. In such a network, a straightforward mapping from each slot to a single stage will induce cycles in $\mathcal{G}$, and this leads to infinite age bounds even if each stage has a known finite residence time. In such a case, the progress stages must be made more precise by refining the $p_{i,j}$ formulas to consider more than just the slot that a packet occupies. Sec. VII demonstrates on a ring interconnect that a manual refinement can be used to obtain an acyclic stage graph from a cyclic network.

B. Channel Blocking Bounds and Progress Lemmas (Θ)

The preceding section shows that stage residence times in the stage graph $\mathcal{G}$ depend on channel blocking bounds of the output channels of queue, and now a heuristic is given for using rule-based propagation to compute blocking bounds. As the heuristic propagation generates blocking bounds, it also generates “progress lemmas” that formalize the assumptions made in deriving the bounds. These progress lemmas are added to the overall verification problem to check the assumptions.

Given a channel c , a blocking bound of x cycles is the claim of $c.trdy \implies \mathbf{F}^{\leq x} c.trdy$. Such a blocking bound of a channel c is computed by first deriving a guarded bound set $r_{c.trdy}$ (Eq. 7). Each guarded bound $\langle g_i, \delta_i \rangle \in r_{c.trdy}$ is a predicate g_i on network state, and δ_i a bound on the number of cycles until $c.trdy$ is asserted from any network state satisfying g_i . In other words, the guarded bound set $r_{c.trdy}$ can be used to make the claim of Eq. 8.

$$r_{c.trdy} \equiv \{ \langle g_0, \delta_0 \rangle, \langle g_1, \delta_1 \rangle, \dots, \langle g_N, \delta_N \rangle \} \quad (7)$$

$$\bigwedge_{i \in [0, N]} g_i \implies \mathbf{F}^{\leq \delta_i} c.trdy \quad (8)$$

The guarded bound set is used to create a single unconditional bound on the number of cycles of blocking on channel c . To accomplish this, the guarded bound set $r_{c.trdy}$ (Eq. 7) is generalized into a single guarded bound $\langle g_{abs}, \delta_{abs} \rangle$, where $g_{abs} := g_0 \vee g_1 \vee \dots \vee g_N$ and $\delta_{abs} := \max(\delta_0, \delta_1, \dots, \delta_N)$. The generalized form of Eq. 8 is then Eq. 9.

$$g_{abs} \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy \quad (9)$$

$$\underbrace{c.irdy \implies g_{abs}}_{\text{guard coverage}} \wedge \underbrace{c.irdy \wedge g_{abs} \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy}_{\text{guard bounds hold}} \quad (10)$$

$$\theta_{c,STATE} := c.irdy \implies g_{abs} \quad (11)$$

$$\theta_{c,TIMING} := c.irdy \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy \quad (12)$$

$$\theta_c := \theta_{c,STATE} \wedge \theta_{c,TIMING} \quad (13)$$

A blocking bound is proved by way of the guarded bound set $r_{c.trdy}$ using Eq. 10. Property $\theta_{c,STATE}$ (Eq. 11) is equivalent to the portion of Eq. 10 labeled “guard coverage”; $\theta_{c,STATE}$ checks that the generalized guard holds in all states where the initiator of channel c is attempting to send a packet. Assuming that $\theta_{c,STATE}$ is valid, then Eq. 10 simplifies to property $\theta_{c,TIMING}$ (Eq. 12). Property $\theta_{c,TIMING}$ checks that the bound (δ_{abs}) implied by the guarded bound set does in fact hold in the network. This property can be considered as a way to check that the bounds that associated with the guards are correct. For each channel c , θ_c (Eq. 13) is checked.

Property θ_c is a roundabout way of proving $c.irdy \wedge g_{abs} \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy$, which is itself just a strengthened version of $c.irdy \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy$ (i.e. a strengthened version of $\theta_{c,TIMING}$). Given that the stage graph ultimately makes use of only δ_{abs} , one may question the motivation for proving $\theta_{c,STATE}$ and $\theta_{c,TIMING}$. These properties are checked because they formalize and validate the assumptions made in deriving δ_{abs} , and if any modifications are needed in generating the guarded bound sets, counterexamples to these properties will indicate where the modifications are required. If property $\theta_{c,STATE}$ fails, then a counterexample reaches a state where $c.irdy$ is **true** and g_{abs} is **false**; this can be remediated by adding to the guarded bound set a new guarded bound $\langle g_i, \delta_i \rangle$ such that g_i is **true** for the bad state of the counterexample. If property $\theta_{c,STATE}$ passes and $\theta_{c,TIMING}$ fails, then it means that there exists some $\langle g_i, \delta_i \rangle$ in the guarded bound set such that the guard (g_i) does not imply the bound (δ_i); this would indicate that the propagation rules used to create the guarded bound set are unsound.

The preceding paragraphs demonstrate how to derive a conservative bound δ_{abs} for an arbitrary channel c . The residence times are created for each stage in the stage graph $\mathcal{G}$ by deriving such a bound for each channel c that is the output of a data queue. The property checked on the network is then Θ (Eq. 14). The following sections present the approach for computing a guarded bound set for each such channel. This is done using operations for combining guarded bound sets, applied according to the xMAS network connections. A primitive has operations to determine the guarded bound set for each of its output signals, and these operations are defined using recurrence relations over other guarded bound sets.

$$\Theta := \bigwedge_{\forall c \text{ in data queue outputs}} \theta_c \quad (14)$$

1) Recurrence Relations for Future Readiness: Analogous to how *irdy* and *trdy* mark current readiness of initiators and targets, guarded bound sets mark future readiness of channel initiators and targets. For any signal *irdy* marking current

readiness, $R(irdy)$ is a symbolic representation of the guarded bound set marking its future readiness. Just as the readiness signals *irdy* and *trdy* are defined using recurrence relations over other readiness signals and state variables, the guarded bound sets for future readiness are defined using recurrence relations over other guarded bound sets and state variables.

The recurrence relations for $R(irdy)$ and $R(trdy)$ are defined using the three operations MAX, PLUS, and ITE. MAX is used when readiness depends on the larger of two guarded bound sets. PLUS is used when readiness depends on the sum of two guarded bound sets. ITE is used when readiness depends on one of two guarded bound sets, with the choice determined by the state of some Boolean condition.

Defining recurrence relations for each output signal of each primitive is the first step toward each expanding each symbolic $R(irdy)$ or $R(trdy)$ into a concrete set of guarded bounds $r_{trdy} \equiv \{\langle g_0, \delta_0 \rangle, \langle g_1, \delta_1 \rangle, \dots, \langle g_N, \delta_N \rangle\}$. While there is no unique solution for defining useful recurrence relations, the relations for each primitive used in this work are shown in graphical form in Fig. 4, and explained in the following paragraphs.

Fig. 4: Dependency graph for future readiness recurrence relations of each primitive. Outputs at the top of each primitive depend on inputs and constants combined using MAX, PLUS, and ITE operations.

Data Source: Because a data source *non-deterministically* injects data packets onto channel o , there is no upper bound on when a packet will be injected. The guarded bound is therefore the empty set $\{\}$; from any state of the network there are no conditions that ensure $o.irdy$ will be asserted in the future. In other words, there is no assurance that the source will ever again attempt to inject a packet.

$$\begin{aligned} o.irdy &:= \{\text{true}, \text{false}\} \\ R(o.irdy) &:= \{\} \end{aligned}$$

Token Source: A token source is the initiator of a single channel o , and from any network state it tries to inject a token across o in every cycle.

$$\begin{aligned} o.irdy &:= \text{true} \\ R(o.irdy) &:= \{\langle \top, 0 \rangle\} \end{aligned}$$

Data Sink: A data sink is the target of a single channel i . The sink provides to the network a service guarantee to always receive a waiting packet within x cycles. This means that $i.trdy$ is guaranteed to occur no more than x cycles after $i.irdy$ does. In the progress lemmas, given that $R(i.irdy)$ is a (recurrence-defined) guarantee on when $i.irdy$ will occur, $R(i.trdy)$'s guarantee is x cycles later than that.

$$\begin{aligned} i.trdy &:= \{\mathbf{true}, \mathbf{false}\} \text{ s.t. } i.irdy \implies \mathbf{F}^{\leq x} i.trdy \\ R(i.trdy) &:= \text{PLUS}(R(i.irdy), \{\langle \top, x \rangle\}) \end{aligned}$$

Token Sink: A token sink is the target of a single token-carrying channel i . Like token sources, token sinks are eager and ready to accept a packet regardless of network state.

$$\begin{aligned} i.trdy &:= \mathbf{true} \\ R(i.trdy) &:= \{\langle \top, 0 \rangle\} \end{aligned}$$

Queue: Queue is the target of channel i and the initiator of channel o . The queue's recurrence relations for future readiness use ITE to introduce a case-split depending on the number of items stored in the queue (denoted n_i). Consider for demonstration $R(o.irdy)$, which evaluates to $\{\langle \top, 0 \rangle\}$ if $n_i \neq 0$, and otherwise evaluates to $\text{PLUS}(R(i.irdy), \{\langle \top, 1 \rangle\})$. In the first case the queue is non-empty, and the value $\{\langle \top, 0 \rangle\}$ reflects that the output is currently trying to initiate a transfer. In the second case the queue is empty, and the value $\text{PLUS}(R(i.irdy), \{\langle \top, 1 \rangle\})$ reflects that the output is ready to initiate no more than 1 cycle after the input is ready to initiate a transfer. The recurrence relation for $R(o.irdy)$ holds because any attempted transfer from the input is immediately received into the empty queue to cause an attempted output transfer in the next cycle. Target readiness $R(i.trdy)$ is handled similarly, with a case split depending on whether or not the queue is currently full.

$$\begin{aligned} o.irdy &:= n_i \neq 0 \\ i.trdy &:= n_i \neq \text{depth} \\ R(o.irdy) &:= \text{ITE}(n_i \neq 0, \{\langle \top, 0 \rangle\}, \text{PLUS}(R(i.irdy), \{\langle \top, 1 \rangle\})) \\ R(i.trdy) &:= \text{ITE}(n_i \neq \text{depth}, \{\langle \top, 0 \rangle\}, \text{PLUS}(R(o.trdy), \{\langle \top, 1 \rangle\})) \end{aligned}$$

Merge: A merge primitive arbitrates between two input channels a and b , and is the initiator of a single channel o . The merge primitive has a Boolean state variable u to store the current priority among inputs. To create a round robin arbitration policy, the state of u is updated with its negation whenever the high priority input transfers a packet through the merge. The recurrence relations abstract away the arbitration priority u to give a conservative bound for each input that does not depend on the value of u . Because the high priority input channel can only be blocked for $R(o.trdy)$ cycles before transferring, the low priority input achieves high priority in at most $\text{PLUS}(R(o.trdy), \{\langle \top, 1 \rangle\})$ cycles. Therefore, the progress bound for input channel of undetermined priority is the time to achieve high priority added to the progress time once high priority is achieved. A merge primitive that uses a priority scheme other than round robin would require modified recurrence relations.

$$\begin{aligned} o.irdy &:= a.irdy \vee b.irdy \\ a.trdy &:= o.trdy \wedge u \\ b.trdy &:= o.trdy \wedge \neg u \\ R(o.irdy) &:= \text{MAX}(R(a.irdy), R(b.irdy)) \\ R(a.trdy) &:= \text{PLUS}(\text{PLUS}(R(o.trdy), \{\langle \top, 1 \rangle\}), R(o.trdy)) \\ R(b.trdy) &:= \text{PLUS}(\text{PLUS}(R(o.trdy), \{\langle \top, 1 \rangle\}), R(o.trdy)) \end{aligned}$$

Switch: Boolean transfer equations for switch primitive are parameterized by a switching function f . In the recurrence relations for future readiness in the forward direction, no assumptions are made on the packet data, and therefore no upper bound is asserted for either output. This can optionally be refined using ITE to take into account the data value of packets on channel i .

$$\begin{aligned} a.irdy &:= i.irdy \wedge f(i) \\ b.irdy &:= i.trdy \wedge \neg f(i) \\ i.trdy &:= (f(i.data) \wedge a.trdy) \vee (\neg f(i.data) \wedge b.trdy) \\ R(a.irdy) &:= \{\} \\ R(b.irdy) &:= \{\} \\ R(i.trdy) &:= \text{MAX}(R(a.trdy), R(b.trdy)) \end{aligned}$$

Join: Join consumes an input packet from channel a and one from channel b to produce a single output packet on channel o . It is only ready to produce a packet on o if both inputs are ready to initiate. It is only ready to consume an input packet from a or b when the other input is ready to initiate and the output is ready to receive. Because each Boolean signal in the join is defined using a logical AND of two other signals, the future readiness of each signal depends on the latest future readiness of the two inputs.

$$\begin{aligned} o.irdy &:= a.irdy \wedge b.irdy \\ a.trdy &:= o.trdy \wedge b.irdy \\ b.trdy &:= o.trdy \wedge a.irdy \\ R(o.irdy) &:= \text{MAX}(R(a.irdy), R(b.irdy)) \\ R(a.trdy) &:= \text{MAX}(R(o.trdy), R(b.irdy)) \\ R(b.trdy) &:= \text{MAX}(R(o.trdy), R(a.irdy)) \end{aligned}$$

Fork: Fork consumes a single input packet from channel i and produces one output packet on a and one on b . Like Join, the Fork uses logical AND for current readiness and therefore uses MAX for future readiness.

$$\begin{aligned} i.trdy &:= a.trdy \wedge b.trdy \\ a.irdy &:= i.irdy \wedge b.trdy \\ b.irdy &:= i.irdy \wedge a.trdy \\ R(i.trdy) &:= \text{MAX}(R(a.trdy), R(b.trdy)) \\ R(a.irdy) &:= \text{MAX}(R(i.irdy), R(b.trdy)) \\ R(b.irdy) &:= \text{MAX}(R(i.irdy), R(a.trdy)) \end{aligned}$$

Function: Function transforms data, but is transparent to $irdy$ and $trdy$ signals and propagates future readiness unchanged.

2) *Computing Guarded Bound Sets:* The guarded bound sets that describe bounds on future readiness (e.g. $r_{c.trdy}$) are created by expanding the symbolic representations of the same (e.g. $R(c.trdy)$) using the dependency graph of the recurrence relations. The dependency graph is created by composing the recurrence relations (Fig. 4) for each primitive in the network according to common signals. Each $irdy$ or $trdy$ signal in $\mathcal{N}$ is an input of one primitive and an output of another; similarly each $R(irdy)$ or $R(trdy)$ is dependent upon the relations of one primitive, and also depended upon by the relations of another primitive. Note that circular dependencies can exist.

For each data queue in the network, with output channel c , **CREATEGUARDEDBOUNDSET** (Procedure 1) is called to extract the guarded bound set $r_{c.trdy}$ from the dependency graph. Starting from the dependency graph node $R(c.trdy)$, function **EXPANDSUBTREE** recursively computes the guarded bound sets for each node's future readiness, and includes an

additional check to short-circuit to an empty set in the case of a cyclic dependency. The guarded bound sets for each node are computed in the tail of the recursion, and thus computed over two concrete guarded bound sets. The procedure for combining the guarded bound sets of a node's left and right dependency graph children is according to whether the node is implementing MAX (line 20), PLUS (line 24), or ITE (line 28). For MAX and PLUS, the guarded bound sets are combined as Cartesian products augmented by the appropriate numeric operation. The ITE operation does not take a Cartesian product because the guards of the two children are made disjoint by including the predicate of the ITE in opposing polarities.

Finally the algorithm returns to line 2 with a guarded bound set ($r_{c.trdy}$) for target readiness of channel c . Because channel c has a queue as its initiator, and only cases where channel c is blocked are relevant, all guards in the set are strengthened with the condition that the initiating queue is non-empty (lines 3-6). A pruning step (line 7) then removes any guarded bounds ($\langle g_i, \delta_i \rangle$) that are trivially unsatisfiable, such as a guard asserting that a single queue is both full and empty; an improved pruning step could also remove guards that violate numeric invariants. Finally, CREATEGUARDEDBOUNDSET returns with the final guarded bound set $r_{c.trdy}$, which is then used as explained at the start of this section (Eq. 7) to derive residence times of stages in $\mathcal{G}$.

3) Limitations in Deriving Blocking Bounds: This work presents recurrence relations that are found to be useful for deriving blocking bounds for motivating example networks. These recurrence relations will not be precise enough to handle all possible networks. Property Θ (Eq. 14) is formulated such that its failure will serve to indicate when the recurrence relations are not suitable for a given network. Furthermore, a counterexample to Θ can guide the development of rules that lead to a more precise set of recurrence relations. Some situations in which this may arise are highlighted.

One abstraction used in the recurrence relations is to consider only whether a queue is full or empty, and not the number of items in the queue. A similar abstraction is made in proving deadlock freedom by Verbeek *et al.* [16]. The significant difference between this work and Verbeek's approach is that this work goes beyond deadlock freedom to include numeric progress bounds in the reasoning. Yet, like Verbeek's work, the abstraction causes the approach to be sound but incomplete for proving latency bounds.

A second conservative abstraction is that the recurrence relations presented make no assumptions about data values of packets. This abstraction prevents the recurrence relations from giving any bounds on readiness outputs of the switch primitive. In a network where progress of a data packet depends on switch output, then refinement would be needed. This situation can be addressed with manual refinement, by modifying the recurrence relations of the switch to take into account the data value of the input. Ongoing work by Viktorov and Gotmanov [17] aims to overcome this limitation by propagating rules that can be automatically refined to handle cases such as this. Counterexample-guided abstraction-refinement techniques [18] could also be used.

Procedure 1 For channel c that is initiated by a data queue storing number of items n_x , expand recurrence relations to obtain a guarded bound set $r_{c.trdy}$. The members of set $r_{c.trdy}$ are guarded bounds $\langle g_i, \delta_i \rangle$, where guard g_i is a condition on network state, and δ_i is a claimed bound on when $c.trdy$ will occur starting from any state satisfying g_i .

```

1: procedure CREATEBOUNDEDBOUNDSET(channel  $c$ )
2:    $r_{c.trdy} \leftarrow \text{EXPANDSUBTREE}(R(c.trdy), \{\})$ 
3:    $nonEmpty := (n_x \neq 0)$   $\triangleright$  queue is initiating on  $c$ 
4:   for  $\langle g_i, \delta_i \rangle \in r_{c.trdy}$  do
5:      $\langle g_i, \delta_i \rangle \leftarrow \langle g_i \wedge nonEmpty, \delta_i \rangle$ 
6:   end for
7:    $r_{c.trdy} \leftarrow \text{PRUNE}(r_{c.trdy})$   $\triangleright$  prune members w/infeas  $g_i$ 
8:   return  $r_{c.trdy}$   $\triangleright r_{c.trdy} \equiv \{\langle g_0, \delta_0 \rangle, \dots, \langle g_N, \delta_N \rangle\}$ 
9: end procedure

10: function EXPANDSUBTREE( $node, visited$ )
11:   if  $node \in visited$  then  $\triangleright$  cycle detected
12:     return  $\{\}$ 
13:   else if  $node$  is childless then  $\triangleright$  e.g.  $node = \{\langle \top, 1 \rangle\}$ 
14:     return  $node$ 
15:   else  $\triangleright$  expand children, compose guarded bound sets
16:      $visited \leftarrow visited \cup node$ 
17:      $r_n \leftarrow \{\}$   $\triangleright$  guarded bound set for  $node$ 
18:      $r_{left} \leftarrow \text{EXPANDSUBTREE}(left, visited)$ 
19:      $r_{right} \leftarrow \text{EXPANDSUBTREE}(right, visited)$ 
20:     if  $node = \text{MAX}(left, right)$  then
21:       for  $\langle g_i, \delta_i \rangle \in r_{left}, \langle g_j, \delta_j \rangle \in r_{right}$  do
22:          $r_n \leftarrow r_n \cup \langle g_i \wedge g_j, \max(\delta_i, \delta_j) \rangle$ 
23:       end for
24:     else if  $node = \text{PLUS}(left, right)$  then
25:       for  $\langle g_i, \delta_i \rangle \in r_{left}, \langle g_j, \delta_j \rangle \in r_{right}$  do
26:          $r_n \leftarrow r_n \cup \langle g_i \wedge g_j, \delta_i + \delta_j \rangle$ 
27:       end for
28:     else if  $node = \text{ITE}(predicate, left, right)$  then
29:       for  $\langle g_i, \delta_i \rangle \in r_{left}$  do
30:          $r_n \leftarrow r_n \cup \langle g_i \wedge predicate, \delta_i \rangle$ 
31:       end for
32:       for  $\langle g_j, \delta_j \rangle \in r_{right}$  do
33:          $r_n \leftarrow r_n \cup \langle g_j \wedge \neg predicate, \delta_j \rangle$ 
34:       end for
35:     end if
36:      $r_n \leftarrow \text{PRUNE}(r_n)$ 
37:     return  $r_n$ 
38:   end if
39: end function

```

V. METHODOLOGY

The methodology used across all experiments is described here. The xMAS models are created within a C++ framework³, with primitives as objects. Progress lemmas and age lemmas are added automatically, and flattened word-level Verilog is generated with all properties added as assertions. The Verilog is bit-blasted into an and-inverter-graph⁴ (AIG) using the VeriABC flow [19]. Verification is performed on the AIG using the bit-level model checker ABC [20]⁵ on a 2.4GHz Intel Core i5 processor with 4GB of RAM. The bounded model

³<https://github.com/danholcomb/xmas-front-end>

⁴<http://fmv.jku.at/aiger>

⁵Rev. d0170182dbd6; at <http://www.eecs.berkeley.edu/~alanmi/abc/>

checking⁶ (BMC), property directed reachability⁷ (PDR) [13], and k-induction⁸ calls are performed by ABC.

A. Evaluating Looseness of T_L with Bounded Model Checking

The global age bound (T_L) that is implied by the age lemmas can be loose. While the tight latency bound for a given network is not generally known, the looseness of T_L is bounded by comparison to T_{FEAS} , where T_{FEAS} is the smallest number such that $\Phi_{T_{FEAS}}^G$ cannot be disproved within allotted resource bounds. T_{FEAS} is found by iteratively increasing T and using BMC to disprove each Φ_T^G until reaching the first value of T that cannot be disproved; that value is T_{FEAS} . There cannot exist a tighter bound than T_{FEAS} because $\Phi_{T_{FEAS}-1}^G$ is disproved by counterexample. However, $\Phi_{T_{FEAS}}^G$ may not be proved, and could instead be an artifact of the BMC resource limits.

B. Efficient Encoding of Packet Ages

The formulation of an age bound property requires that the age of a packet in slot i (denoted $age(q_i)$) can be checked as a simple safety property; in other words, the age of a packet can be evaluated in a single state of $\mathcal{N}$. The age of a packet occupying slot i is denoted $age(q_i)$, and begins at 0 when the packet is first injected and then increments in every cycle until the packet is ejected. The variable $age(q_i)$ is considered a part of the data packet and requires each queue slot in the network to be widened by $\log_2(t_{max})$ bits to store it, where t_{max} is some number that exceeds the largest packet age bound that is checked. Because standard xMAS queues store data without modifying it, some extra logic is added to the queue primitives for incrementing packet ages. For each queue slot i , the age variable $age(q_i)$ that is stored is incremented by 1 (mod t_{max}) relative to the value that would be stored by a normal xMAS queue. This ensures that the ages are incremented both for packets remaining in slot i from the previous cycle, and for packets being newly written into slot i from the queue's input channel. An empirical runtime comparison [21] favors for inductive verification the particular encoding of packet ages employed here over that of earlier works [22].

VI. ILLUSTRATIVE EXAMPLES

Several examples highlight strengths and weaknesses of using latency lemmas. The primary strength is a dramatic improvement in verification runtime, and the weakness is that the bounds proved using lemmas are in some cases loose. All latency lemma reasoning including the stage graph construction is automated for the examples in this section. The subsequent example of a ring interconnect (Sec. VII) is a case of an xMAS extension where some manual reasoning is needed to create an acyclic stage graph and apply latency lemmas.

A. Single Queue

An example with a single queue (Fig. 5) demonstrates scalability of latency lemmas in a simple network without arbitration,

routing, or flow control. The liveness bound of the sink is fixed to 3 in this example, and the depth of the queue is varied. The progress lemmas give a bound of $\delta_{abs} = 3$ for the queue's output channel, and therefore each slot in the queue maps to a stage s_j in $\mathcal{G}$ with a weight of $d_j = 4$. The global bound from the lemmas is $T_L = 1 + 4 * depth$. For each queue depth it is found that $T_{FEAS} = T_L - 1$ (Fig. 6), indicating that T_L is one cycle larger than the tightest feasible bound. The bound T_L at each queue depth is proved inductively, using two different strengthenings of the latency property $\Phi_{T_L}^G$. The first property ($\Phi_{T_L}^G \wedge \Psi$) proves latency bound T_L without lemmas, strengthened only by auxiliary invariants Ψ . The second property ($\Phi_{T_L}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$) proves the same bound strengthened by the age lemmas (Φ^L) and progress lemmas (Θ). Without latency lemmas, the induction depth required for the proof is never less than T_{FEAS} . With latency lemmas, the induction depth is 4 independent of the queue depth, demonstrating the composability of the approach. In a queue with depth 10, a latency bound of 41 cycles is proved in under 2 seconds with latency lemmas versus 267 seconds without.

Fig. 5: Queue network with stage graph $\mathcal{G}$ shown above. For the particular case of $depth = 2$ that is shown, the global age bound implied by the lemmas is $T_L = 9$.

Fig. 6: Sweeping the queue depth in a network comprising a single queue, and a sink with liveness bound of 3. In the lower plot, for the two properties the Y-axis is the induction depth required to prove them; for T_{FEAS} the Y-axis is the tightest feasible bound; for T_L the Y-axis is the bound proved using the lemmas.

B. Credit Loop

The credit loop (Fig. 3) introduced in Sec. II-B is now revisited in more detail. Experiments are performed to explore the

⁶ABC commands "read_aiger foo.aig; bmc3; write_counter -n foo.cex;"

⁷ABC commands "read_aiger foo.aig; pdr -v;"

⁸ABC commands "read_aiger foo.aig; orpos; ind -avw; bmc3 -F k;" where k is the depth used by the ind command

scalability of the latency lemma approach, different verification engines, and the tradeoff of verification runtime versus tightness of proved bounds. A credit loop has a numeric invariant ψ_{num} [6] (Eq. 15) asserting that each outstanding credit corresponds to exactly one available token or data packet in the ingress queue; this numeric invariant is included as part of auxiliary invariant Ψ .

$$\psi_{num} := n_0 + n_1 = n_2 \quad (15)$$

1) *Sweeping the Depth of Queues:* As the depth of the credit loop queues are swept from 2 to 10 (Fig. 7), the bounds implied by the lemmas (T_L) at each depth exceed the tightest feasible bound (T_{FEAS}) on account of the conservativeness of the progress lemmas. As in Sec. II-B, the bound of the sink is 5. The inclusion of latency lemmas yields inductive latency proofs in 8 frames of unrolling and less than 11 seconds of runtime for all depths. For a queue depth of 10, the lemmas give a speedup of 120x.

Fig. 7: Comparing runtime and induction depth for proving latency bounds with and without latency lemmas while varying the depth of the queues in a credit loop (Fig 3).

2) *Comparing Proof Engines:* Induction is evaluated against the PDR verification engine when the latency property is formulated with and without latency lemmas. In this experiment, the queue depths are fixed to 6 and the sink bound is again 5. Attempts are made to prove 2 different bounds; the first is the tight bound (T_{FEAS}), and the second is the looser bound (T_L) implied by the stage graph $\mathcal{G}$ lemmas. The results are shown in Tab I. When proving tight bound T_{FEAS} , the PDR engine gives a 3x speedup over induction, and adding latency lemmas does not significantly impact runtime. When proving the looser bound T_L , adding latency lemmas causes a dramatic speedup in inductive verification. The speedup is 34x compared to induction without the lemmas, and over 7x compared to PDR with and without lemmas. The speedup is caused by the latency lemmas making the proof compositional and hence provable in only 8 frames of unrolling.

3) *Precision vs Scalability:* This section demonstrates that, when using latency lemmas, there exists a tradeoff of inductive verification runtime against looseness of proved bounds. This generalizes the speedup observed in Tab I when proving the looser bound T_L instead of tight bound T_{FEAS} . The tradeoff is

find bound	Runtime (s)	Frames	Cex	Engine	Property
$T_{FEAS} \equiv 35$	52.79	42	Y	bmc	Φ_{34}^G
	1045.62	200	-	bmc	Φ_{35}^G
verify bound	Runtime (s)	Frames	Proved	Engine	Property
$T_{FEAS} \equiv 35$	118.79	37	Y	kind	$\Phi_{35}^G \wedge \Psi$
	95.89	37	Y	kind	$\Phi_{35}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$
	1800.00	-	-	pdr	Φ_{35}^G
	29.57	41	Y	pdr	$\Phi_{35}^G \wedge \Psi$
	27.97	41	Y	pdr	$\Phi_{35}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$
verify bound	Runtime (s)	Frames	Proved	Engine	Property
$T_L \equiv 43$	150.97	37	Y	kind	$\Phi_{43}^G \wedge \Psi$
	3.41	8	Y	kind	$\Phi_{43}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$
	1800.00	-	-	pdr	Φ_{43}^G
	30.61	40	Y	pdr	$\Phi_{43}^G \wedge \Psi$
	25.73	47	Y	pdr	$\Phi_{43}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$

TABLE I: Proving latency bounds for credit loop with queue depth of 6 and a sink bound of 5. The tightest feasible bound is 35 cycles and the the bound implied by the lemmas is 43 cycles. The latency lemmas have no significant effect when proving the tighter bound, but allow the looser bound of 43 cycles to be proved in only 3.41 seconds and 8 frames.

shown by proving individually each bound from T_{FEAS} to $T_L + 5$ (Fig. 8). The black vertical line indicates the tight bound $T = T_{FEAS} = 35$ and the gray indicates $T = T_L = 43$. As the verified bound increases from T_{FEAS} to T_L the property including the lemmas gets progressively easier to prove, as evidenced by the reduction in both verification runtime and the number of frames needed for the proof. The points where the plotted data cross the black and grey vertical lines correspond to the four rows in Tab. I that use k-induction as the verification engine.

Fig. 8: For credit loop, when proving a latency property with latency lemmas, proving a larger latency bound (T) leads to a reduced runtime and reduced number of frames in the proof.

VII. NON-STALLABLE RING INTERCONNECT

A ring network [23] is a topology for routing traffic amongst a number of agents. Each agent in the ring comprises arbitration logic, a ring queue slot, and an ingress queue (Fig. 9). Packets

Fig. 9: Parameterized ring network, instantiated with 3 agents and an ingress depth of 2. All channels in ring network carry data packets, and control is implemented using sequential reservation logic instead of tokens.

reach their destinations by circling around the ring until being admitted into their destination agent's ingress. The ring network is parameterized by the number of agents, depth of ingress queues, and the liveness bound of each agent's sink.

A packet that is injected at agent i and destined for agent j will first occupy the ring slot of agent i . The packet circles the ring thereafter (e.g. occupying slots 6, 7, 8, 6, ... in Fig. 9) and requests admission to the ingress whenever arriving at agent j . If the request is denied, the packet *bounces* back onto the ring to repeat the request after making one trip around the ring. Unfair arbitration logic prioritizes traffic in the ring over traffic attempting to enter the ring from a source. This unfair arbitration ensures that traffic in the ring is never blocked, but it does permit sources to be blocked indefinitely.

Each packet that a source injects into the ring is non-deterministically assigned a destination address between 0 and $n - 1$ to indicate the agent to which the packet should be routed. This destination is stored using additional bits appended to nominal packet data in the same manner as the packet age. For a packet stored in queue slot i , let $dst(q_i)$ be its destination address. The auxiliary inductive invariant Ψ includes ψ_{dst} (Eq. 16) to block off unreachable states where packets have invalid destinations.

$$\psi_{dst} := \bigwedge_{i \in [0, M-1]} used_i \implies dst(q_i) \in [0, n-1] \quad (16)$$

A. Implementation of a Ring Agent

Each agent in the ring is created in xMAS design style using modified versions of the basic xMAS primitives (Fig. 2) to implement reservations and unfair arbitration. When a packet is transferred from one ring agent to the next, the first primitive encountered is a switch that routes the packet toward the admission logic if this agent is the packet's destination, and to the bypass channel otherwise. Packets that are routed to the admission logic encounter a demultiplexer that is controlled by sequential reservation logic. The state of the sequential reservation logic and the number of free slots in the ingress determine whether the packet is admitted or bounced. A merge primitive then propagates onward a bounced packet or bypass packet, or no packet at all. Finally, a priority merge primitive

gives priority to packets already on the ring, and allows the source to inject packets only if there is no competing packet on the ring.

B. Receive Reservation Logic

A naïve ring implementation can have infinite latency even though all sinks obey bounded liveness. A single packet on the ring may never be granted access to the ingress of its destination, despite an unbounded number of other packets being granted access to the same ingress. Receive reservations [24] are a mechanism to enforce fairness; together with bounded liveness of sinks, receive reservations ensure that packets on the ring have finite latency bounds. The receive reservation scheme used by the ring agents is described here:

- 1) Each agent can issue a single receive reservation. If an agent's reservation is available, then the agent issues it to any packet that is bounced (due to a full ingress). The next ingress slot to become free is reserved for this packet.
- 2) If the agent has an outstanding receive reservation, packets without the reservation are denied entry to the ingress unless more than 1 slot is free.
- 3) When a packet with a receive reservation returns, it is granted entry to the ingress if any slots are free, and the reservation becomes available for other packets. If no slots are free, the reservation is renewed by the packet and remains unavailable.

The receive reservation of each agent is implemented using a modified xMAS switch in which an incoming packet is either bounced or admitted to the ingress (see Fig. 9) depending on the current state of sequential control logic. The sequential control logic does not use the xMAS design style, and hence this modified switch gives an example of how the latency verification approach of this paper can be applied beyond basic xMAS. As shown in Fig. 10, the control logic tracks the reservation using a sort of counter. When the receive logic is in state $rsv = n$, it is an indication that the packet with the reservation will return in n cycles. When the state reaches $rsv = 0$, the next arriving packet on the ring is the same one for which the reservation was made. State $rsv = \perp$ indicates that the reservation is available. Receive reservations are fair with respect to packets in the ring. Whenever one packet returns the reservation (see edge *return reservation* in Fig. 10), the packet trailing it on the ring has a chance to make the reservation in the next cycle. Each packet in the ring gets a turn at making a receive reservation in order.

The dashed edge labeled *renew reservation* in Fig. 10 is taken when a packet holding the reservation bounces at its destination on account of there not yet being any free slots in the destination's ingress. This situation of renewing a reservation cannot occur if the sink bounds are smaller than the delay around the ring (i.e. the number of agents), a condition that is here assumed to hold. Therefore, the dashed edge from Fig. 10 is ignored, and packets arriving when the reservation state is $rsv = 0$ will always find a free ingress slot and be admitted to the ingress.

C. Creating Age Lemmas using Stage Graph $\mathcal{G}$

The location of packets in the ring network is not a precise enough indicator of progress to create an acyclic stage graph

Fig. 10: The state machine for receive reservations of a single agent in a 3-agent ring. The reservation state is $rsv = \perp$ when the reservation is available, and is $rsv \in \{0, 1, 2\}$ when the reservation is outstanding. The state is $rsv = 0$ when the packet holding the reservation is the next to arrive, and the state is $rsv = 2$ when the reservation has just been made and the reserving packet is in the ring slot of its destination agent.

Fig. 11: Product automaton for state of receive reservation and index of currently occupied ring slot in a 3-agent ring (Fig. 9)

because a packet can occupy the same slot many times as it circles the ring. This precludes use of the automated stage graph construction of Sec. IV, which attempts to equate a packet's location to its stage of progress. Instead, some manual insight is required to devise a correspondence between packets circling the ring and ordered stages of progress in the stage graph. For packets circling the ring, progress is marked both by changes in its location, and changes in the reservation state of its destination agent. In an n -agent ring, there exists a stage in $\mathcal{G}$ for every combination of the n current packet locations, the n packet destinations, and the $n + 1$ reservation states, so the total number of stages for the ring slots is $n \times n \times (n + 1)$. For clarity, the explanation here deals only with the $n \times (n + 1)$ stages for packets destined for agent 2; in implementation all destinations are considered.

Using the 3-agent ring as an example, composing the reserva-

Fig. 12: Stage graph $\mathcal{G}$ for the ring network. Above each stage s_j is weight d_j and age bound t_j in parentheses

tion state machine of Fig. 10 (without dashed edge) with the packet's behavior of advancing to the next ring slot in every cycle produces the state machine of Fig. 11. This product machine is the foundation for creating a stage graph for packets in the ring. Each state in Fig. 11 has two labels, the first is the state of the reservation state machine (Fig. 10) and the second is the index of a ring slot (Fig. 9). As a packet destined for agent 2 moves around the ring, in every cycle it maps to some state of this product automaton. For example, state $(0, 7)$ in Fig. 11 is the state that a packet in slot 7 maps to when the receive reservation of agent 2 has state $rsv_2 = 0$. The mapping from packets to states in Fig. 11 could serve as an indicator of progress if only the product automaton were acyclic. An acyclic stage graph is obtained from the product automaton by discerning that there are pairs of edge-connected states in Fig. 11 that no single packet can map to. Removing these transitions reveals an ordering among progress stages.

- $(0, 7) \rightarrow (\perp, 8)$ can never be made by a packet destined for agent 2 because it corresponds to a packet bouncing at agent 2 (from slot 7 to slot 8) while agent 2 has its reservation returned (i.e. it transitions to state $rsv_2 = \perp$ that indicates that a reservation is available). The transition is impossible because bouncing and returning a reservation are exclusive; only admitted packets cause the reservation to return to the available state.
- $(\perp, 7) \rightarrow (\perp, 8)$ can never be made by a packet destined for agent 2 because it corresponds to a packet that bounces (from slot 7 to slot 8) while an available reservation remains available. The transition is impossible because any packet bouncing while the reservation is available would have the reservation issued to it.

Without the two transitions described above, Fig. 11 becomes acyclic and can be used to order the progress stages of a packet in the ring. The product machine of Fig. 11 then becomes the stage graph of Fig. 12 by simply removing the two unrealizable transitions, and adding stages for sources, ingress slots, and the sink. The mapping from queue slots in ring network to stages in Fig. 12 is given by the age lemmas in Tab. II.

stage s_j Fig. 12	d_j	i	age lemma $\phi(i, p_{i,j}, t_j)$ $p_{i,j}$	t_j
s_0	1	-	-	1
s_1	1	8	$dst(q_8) = 2 \wedge rsv_2 = \perp$	2
s_2	1	6	$dst(q_6) = 2 \wedge rsv_2 = 2$	3
s_3	1	7	$dst(q_7) = 2 \wedge rsv_2 = 1$	4
s_4	1	8	$dst(q_8) = 2 \wedge rsv_2 = 0$	5
s_5	1	6	$dst(q_6) = 2 \wedge rsv_2 = \perp$	6
s_6	1	7	$dst(q_7) = 2 \wedge rsv_2 = 2$	7
s_7	1	8	$dst(q_8) = 2 \wedge rsv_2 = 1$	8
s_8	1	6	$dst(q_6) = 2 \wedge rsv_2 = 0$	9
s_9	1	7	$dst(q_7) = 2 \wedge rsv_2 = \perp$	10
s_{10}	1	8	$dst(q_8) = 2 \wedge rsv_2 = 2$	11
s_{11}	1	6	$dst(q_6) = 2 \wedge rsv_2 = 1$	12
s_{12}	1	7	$dst(q_7) = 2 \wedge rsv_2 = 0$	13
s_{13}	3	4	true	16
s_{14}	3	5	true	19
s_{15}	0	-	-	19

TABLE II: The age lemmas for packets destined for agent 2. The first column is the stage in $\mathcal{G}$ (Fig. 12) that corresponds to the age lemma described on the row.

D. Results

The latency lemma approach is evaluated on a 3-agent ring and an 8-agent ring, each with ingress depth of 2 and sink

bound of 2. For the 3-agent ring, the tightest feasible bound (T_{FEAS}) is 18, and the bound implied by the latency lemmas (T_L) is 19; for the 8-agent ring, T_{FEAS} is 78 and T_L is 79. The runtimes and number of frames for proving a bound of T_L on each ring, with and without latency lemmas, are shown in Tab. III and IV. Property $\Phi_{T_L}^G \wedge \Psi$ proves the latency bound T_L without latency lemmas, and property $\Phi_{T_L}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$ proves it with the lemmas added. Strengthening the global latency bound property with latency lemmas reduces the verification runtime for both k-induction and PDR in both ring networks.

The latency bound for the 8-agent ring (Tab. IV) is only proved by each engine within 10,000 seconds when the lemmas are used. The induction engine is able to verify the property with the lemmas 9x faster than PDR verifies the same, and at least 130x faster than either engine does without lemmas.

find bound $T_{FEAS} \equiv 18$	Runtime (s)	Frames	Cex	Engine	Property
	24.12	20	Y	bmc	Φ_{17}^G
	868.28	200	-	bmc	Φ_{18}^G
verify bound $T_L \equiv 19$	Runtime (s)	Frames	Proved	Engine	Property
	62.34	18	Y	kind	$\Phi_{19}^G \wedge \Psi$
	1.31	4	Y	kind	$\Phi_{19}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$
	10,000.00	-	-	pdr	Φ_{19}^G
	88.39	12	Y	pdr	$\Phi_{19}^G \wedge \Psi$
	6.57	14	Y	pdr	$\Phi_{19}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$

TABLE III: Proving latency bounds for 3-agent ring with ingress depth 2 and sink bound 2. The 19 cycle bound that is implied by the lemmas exceeds T_{FEAS} by only 1 cycle.

find bound $T_{FEAS} \equiv 78$	Runtime (s)	Frames	Cex	Engine	Property
	3901.95	80	Y	bmc	Φ_{77}^G
	10,000.00	111	-	bmc	Φ_{78}^G
verify bound $T_L \equiv 79$	Runtime (s)	Frames	Proved	Engine	Property
	10,000.00	-	-	kind	$\Phi_{79}^G \wedge \Psi$
	75.28	10	Y	kind	$\Phi_{79}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$
	10,000.00	-	-	pdr	Φ_{79}^G
	10,000.00	-	-	pdr	$\Phi_{79}^G \wedge \Psi$
	662.15	73	Y	pdr	$\Phi_{79}^G \wedge \Psi \wedge \Phi^L \wedge \Theta$

TABLE IV: Proving latency bounds for 8-agent ring with ingress depth 2 and sink bound 2. The 79 cycle bound that is implied by the lemmas exceeds T_{FEAS} by only 1 cycle. This latency bound can only be proved within 10,000 seconds when latency lemmas are used.

VIII. RELATED WORK

One way of addressing QoS guarantees at the architectural level is to use resource reservation and contention-free routing [25]. Analysis can be performed manually, but formal verification is still useful for providing guarantees. Network calculus [26] has been demonstrated as a useful tool for NoC performance analysis [27]. However, it has limited applicability and precision for networks with backpressure and complex circular message dependencies. Network calculus formalism relies on very high-level abstraction of arbiters, often modeling them as latency-rate servers. The synchronous protocol automata of Avnit *et al.* [4] model network components using

an xMAS-like formalism, and as in this current work make the distinction between data and token channels.

Recent abstraction-based model checking approaches have been applied to latency verification [9], [28], but these works address scalability by explicitly decomposing the overall problem into distinct subproblems. The proofs for the subproblems are then stitched together for an end-to-end proof. By contrast, this current work uses subgoals to strengthen the overall latency bound property to make it efficiently provable with induction, but avoids explicit decomposition.

Several works have explored (unbounded) liveness verification of communication fabrics. The standard approach of verifying liveness using a liveness-to-safety transformation [29] does not scale to large networks in practice [15]. Alternative approaches include reducing deadlock conditions to a set of equations [15], [16], and proving liveness using the help of intermediate safety assertions [30]. Higher-order logic has been applied to verifying deadlock freedom, using network models described in the PVS specification language [31]. Prior works compare liveness and safety methods for verifying grant latencies on a particular style of weighted round robin arbiter [32]. The notion of using LTL properties where all eventually properties have time bounds is also referred to as a prompt system [33]. Finite latency bounds imply deadlock freedom; however, because the latency verification approach in this paper assumes deadlock freedom as a starting point, it is intended to complement and not replace existing techniques for verifying deadlock freedom.

The presented verification approach is conceptually similar to ranking functions [34], i.e. numeric functions of model state that measure progress toward some goal. Typically, ranking functions are useful in proving termination or liveness properties, but they are also applicable for latency bounds. In fact, the stage graph can be viewed as a structural description of a ranking function for the model. Note, however, that stage graphs specify partial orders, rather than the linear orders that are typical for ranking functions. Viktorov and Gotmanov [17] propose a theorem-proving approach to latency verification in xMAS networks that is based on ranking functions. Their inference rules are analogous to the rule-based propagation of local bounds used in this work.

IX. CONCLUSION

This work presents a compositional approach to verifying latency bound properties of NoC designs. The key idea is to decompose the overall proof into a finite number of latency lemmas, based on the notion of stages that a packet can be in. The approach is fully automated for acyclic networks constructed from basic xMAS primitives, and some manual input is required for cyclic xMAS networks or xMAS-like networks that use an extended set of primitives. Promising directions for future work include automation of the stage graph construction for cyclic networks, and applying the approach to QoS properties other than latency. The latency lemma approach is applied to several examples including an industrial ring design, and is shown to decrease runtime for proving latency bounds, while also decreasing the induction depth needed to prove them.

Acknowledgments. The authors thank Satrajit Chatterjee, Alexander Gotmanov, Michael Kishinevsky and Yuriy Vik-

torov for technical discussions and feedback, and thank Bob Brayton and his research group for assistance with tools. This work was partially supported by the NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering (grant CCF-1139138), and was supported in part by the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

REFERENCES

- [1] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. Brown, and A. Agrawal, "On-chip interconnection architecture of the tile processor," *Micro*, 2007.
- [2] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation—A performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.
- [3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *Micro, IEEE*, vol. 29, no. 1, pp. 10–21, 2009.
- [4] K. Avnit, V. D'silva, A. Sowmya, S. Ramesh, and S. Parameswaran, "Provably correct on-chip communication: A formal approach to automatic protocol converter synthesis," *Transactions on Design Automation of Electronic Systems*, vol. 14, no. 2, Mar. 2009.
- [5] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "Quick Formal Modeling of Communication Fabrics to Enable Verification," in *High Level Design Validation and Test Workshop (HLDVT)*. HLDVT, 2010.
- [6] S. Chatterjee and M. Kishinevsky, "Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics," in *Computer Aided Verification*, 2010.
- [7] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification," *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.
- [8] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," *Formal Methods in System Design*, vol. 40, no. 2, pp. 147–169, 2012.
- [9] D. E. Holcomb, B. A. Brady, and S. A. Seshia, "Abstraction-Based Performance Analysis of NoCs," *Design Automation Conference*, Jun. 2011.
- [10] M. Sheeran, S. Singh, and G. Stalmark, "Checking safety properties using induction and a SAT-solver," *Formal Methods in Computer-Aided Design*, 2000.
- [11] K. L. McMillan, "Interpolation and SAT-based model checking," *Computer Aided Verification*, 2003.
- [12] A. R. Bradley, "SAT-Based Model Checking Without Unrolling," *Verification, Model Checking, and Abstract Interpretation*, 2011.
- [13] N. Een and A. Mishchenko, "Efficient implementation of property directed reachability," in *Proceedings of IWLS*, 2011.
- [14] D. E. Holcomb, A. Gotmanov, M. Kishinevsky, and S. A. Seshia, "Compositional Performance Verification of NoC Designs," in *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Jul. 2012.
- [15] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, "Verifying Deadlock-Freedom of Communication Fabrics," *Verification, Model Checking, and Abstract Interpretation*, 2011.
- [16] F. Verbeek and J. Schmaltz, "Hunting deadlocks efficiently in microarchitectural models of communication fabrics," *Formal Methods in Computer-Aided Design*, 2011.
- [17] Y. Viktorov and A. Gotmanov, "Latency Analysis in Microarchitectural Models of Communication Fabrics," *Problems of Advanced Micro- and Nanoelectronic Systems Development (MES)*, pp. 67–72, 2012.
- [18] E. Clarke, O. Grumberg, S. Jha, and Y. Lu, "Counterexample-guided abstraction refinement," *Computer Aided Verification*, 2000.
- [19] J. Long, S. Ray, B. Sterin, A. Mishchenko, and R. K. Brayton, "Enhancing ABC for LTL Stabilization Verification of SystemVerilog/VHDL Models," *International Workshop on Design and Implementation of Formal Tools and Systems*, Jan. 2011.
- [20] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," *Computer Aided Verification*, 2010.
- [21] D. Holcomb, "Formal Verification and Synthesis for Quality-of-Service in On-Chip Networks," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec. 2013.
- [22] B. A. Brady, D. E. Holcomb, and S. A. Seshia, "Counterexample-Guided SMT-Driven Optimal Buffer Sizing," *Design Automation and Test in Europe*, 2011.
- [23] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*.
- [24] M. Mattina, G. Z. Chrysos, and S. Felix, "Method and apparatus for synchronous unbuffered flow control of packets on a ring interconnect," *US Patent 7,539,141*, May 2009.
- [25] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal network on chip: concepts, architectures, and implementations," *Design Test of Computers, IEEE*, vol. 22, no. 5, pp. 414–421, Sep. 2005.
- [26] R. L. Cruz, "A calculus for network delay, part I. Network elements in isolation," *IEEE Transactions on Information theory*, vol. 37, no. 1, pp. 114–131, 1991.
- [27] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Modelling runtime arbitration by latency-rate servers in dataflow graphs," in *10th Workshop on Software & Compilers for Embedded Systems*. New York, NY, USA, 2007, pp. 11–22.
- [28] V. A. Palaniveloo and A. Sowmya, "Formal Estimation of Worst-Case Communication Latency in a Network-on-Chip," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 15–20, 2012.
- [29] A. Biere and C. Artho, "Liveness Checking as Safety Checking," *Electronic Notes in Theoretical Computer Science*, 2002.
- [30] S. Ray and R. K. Brayton, "Well-foundedness in Credit-Based Flow-Control Systems," *International Workshop on Logic Synthesis*, pp. 1–8, Mar. 2012.
- [31] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijkema, and A. Radulescu, "Deadlock prevention in the Æthereal protocol," *Correct Hardware Design and Verification Methods*, vol. 3725, no. Chapter 28, pp. 345–348, 2005.
- [32] B. A. Krishna, J. Michelson, V. Singhal, and A. Jain, "Liveness vs Safety—A Practical Viewpoint," *7th international Haifa Verification conference on Hardware and Software*, 2011.
- [33] O. Kupferman, N. Piterman, and M. Y. Vardi, "From liveness to promptness," *Formal Methods in System Design 2009*, Jan. 2009.
- [34] A. M. Turing, "Checking a large routine," *Conference on High Speed Automatic Calculating Machines*, 1949.

Daniel E. Holcomb . Daniel E. Holcomb is a researcher in the area of Formal Verification, VLSI, Embedded Systems, and Hardware Security. He received the B.S. and M.S. degrees from the University of Massachusetts Amherst in 2005 and 2007 respectively, and the Ph.D. from the University of California Berkeley in 2013. He has completed five internships with Intel Corporation and is currently a research fellow in the Security and Privacy Research Group at the University of Michigan.

Sanjit A. Seshia . Sanjit A. Seshia (S'99-M'05-SM'11) received the B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology, Bombay, India in 1998, and the M.S. and Ph.D. degrees in Computer Science from Carnegie Mellon University, Pittsburgh, PA, USA, in 2000 and 2005 respectively.

He is currently an Associate Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley, CA, USA. His research interests are in dependable computing and computational logic, with a current focus on applying automated formal methods to problems in embedded systems, electronic design automation, computer security, and synthetic biology. His Ph.D. thesis work on the UCLID verifier and decision procedure helped pioneer the area of satisfiability modulo theories (SMT) and SMT-based verification. He is co-author of a widely-used textbook on embedded systems.

Prof. Seshia is currently an Associate Editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, and served as co-chair of the Program Committee of the International Conference on Computer-Aided Verification (CAV) in 2012. His awards and honors include a Presidential Early Career Award for Scientists and Engineers (PECASE) from the White House, an Alfred P. Sloan Research Fellowship, the Prof. R. Narasimhan Lecture Award, and the School of Computer Science Distinguished Dissertation Award at Carnegie Mellon University.