Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics

Satrajit Chatterjee and Michael Kishinevsky

Intel Corporation, Hillsboro OR 97124, USA,
satrajit.chatterjee@intel.com, michael.kishinevsky@intel.com

Abstract. Abstract microarchitectural models of communication fabrics present a challenge for verification. Due to the presence of deep pipelining, a large number of queues and distributed control, the state space of such models is usually too large for enumeration by protocol verification tools such as Murphi. On the other hand, we find that state-of-the-art RTL model checkers such as ABC have poor performance on these models since there is very little opportunity for localization and most of the recent capacity advances in RTL model checking have come from better ways of discarding the irrelevant parts of the model. In this work we explore a new approach for verifying these models where we capture a model at a high level of abstraction by requiring that it be described using a small set of well-defined microarchitectural primitives. We exploit the high level structure present in this description, to automatically strengthen some classes of properties, in order to make them 1-step inductive, and then use an RTL model checker to prove them. In some cases, even if we cannot make the property inductive, we can dramatically reduce the number and complexity of lemmas that are needed to make the property inductive.

1 Introduction

Consider the microarchitectural model shown in Figure 1. It consists of a source that non-deterministically generates packets that contain the 6-bit value 0. The source feeds into a pair of serially connected FIFOs each of size k, the second of which feeds into a sink that consumes a packet non-deterministically. The communication between the source, the FIFOs and the sink is by means of a simple handshake. We present a formal semantics for these microarchitectural primitives in Section 3, but we hope that for now this intuitive description suffices.

Consider the problem of verifying that any packet seen at the output of the second FIFO contains the value 0. If we generate Verilog from this description and use a state-of-the-art RTL model checking engine such as ABC [3] (winner of the 2008 and 2010 CAV Hardware Model Checking contests), we find that this apparently trivial problem is surprisingly hard even for small values of k. For instance, even for $k = 4$, ABC takes about 10 minutes to solve this problem on an Intel 3 GHz Xeon processor resorting to interpolation to prove it. Our experience with other industrial tools is similar. And this is for a system with only two queues and a simple topology. In our work on modeling the microarchitecture of communication fabrics we routinely encounter systems where a packet may traverse tens of queues in its lifetime (due to pipelining, path splitting and reconvergence, etc.) and there is complex control logic for resource management.

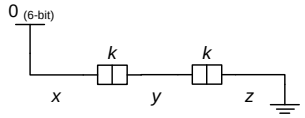

Fig. 1. A simple microarchitectural model with a source that generates the 6-bit value 0, two queues that can store k elements each and a sink. The components are connected by channels x, y and z.

Therefore, even if each queue is sized minimally and packets are represented abstractly, there is still a lot of state. RTL model checkers – though useful for bounded model checking – are unsuccessful in producing proofs for all but the simplest examples even when run for days or weeks. On the other hand, explicit state model checkers such as Murphi run out of memory since there are many interleavings due to non-determinism and deep pipelining.

If we go back to our example, it is obvious to a human designer that the property should hold. It is obvious since we are able to use our knowledge of queues in order to reason about the system. However, when we throw this problem to ABC or Murphi, this high-level information is lost. ABC sees a sea of gates, and Murphi a sea of rules. The traditional approach to handling such verification problems is to resort to theorem proving, or its cousin manual invariant strengthening. In manual invariant strengthening, a verification engineer adds additional invariants (called lemmas) to the model so that the entire set of invariants becomes inductive. Adding these additional invariants is a black art often requiring expertise both in formal verification and the system being verified [14] though some of the burden of inductive strengthening can be reduced with automated tool support (e.g. the STeP system [4]).

In this work, we seek a less labor-intensive way of exploiting the high-level structure of our models than theorem proving or invariant strengthening. The key idea is to require that the microarchitectural models be described in terms of a small set of primitives such as queues, arbiters, forks and joins. Using our knowledge of these primitives, we can *automatically* add a number of lemmas so that the whole set of invariants becomes (1-step) inductive. Most of these lemmas are not local primitive-specific invariants, but are obtained by global analysis of the model. The experimental results are very encouraging: with no or little human effort and little CPU time, we can now prove a number of properties on real models which could not be proved before. In our example above, all necessary lemmas are added automatically, and ABC discharges the resulting problem in almost no time.

The requirement that the model be expressed in terms of specific primitives could be a difficult one to satisfy in general. However, the set of primitives we use in this work originated in a project aimed at reducing the effort required to write microarchitectural models of communication fabrics [6]. Using this modeling methodology we have been able to capture the microarchitecture of a number of real designs and to validate them using simulation and bounded model checking. The goal of this work is to extend verification to obtain full proofs of correctness for some important types of properties.

This approach shares similarities with other work in automated invariant discovery for specific classes of programs using techniques such as abstract interpretation (e.g. see [4] for a good overview). Beyond the obvious difference on focusing on a new class of programs (i.e. synchronous microarchitectural models) where existing invariant

discovery methods do not directly apply, there is another interesting difference with prior work in invariant discovery: We do not directly analyse the program describing the state transition graph of the system. Instead, the object of our analysis is a higher level model which only indirectly specifies the state transition graph. We believe that this leads to more scalable invariant discovery in addition to allowing invariants to be discovered that would be very hard to find from the low-level program describing the state transition graph. We defer a more detailed discussion of this important point to Section 4.8 after we present our approach.

The use of "high-level structure" for more efficient model checking is a holy grail of hardware verification. We believe this work makes a contribution in that direction by presenting a concrete proposal for describing hardware at a level of abstraction higher than RTL along with a couple of analysis techniques that illustrate how such structure could be exploited for efficient verification. The properties we consider are simpler than those verified in previously published manual efforts (e.g. see [13, 14] and references therein) but we seek more automation. On the other hand, the use of high-level structure allows us to infer invariants which would be very difficult for existing automatic RTL-based methods (e.g. see [1] and references therein) to discover. What we present is only a beginning, as we focus here on simple safety properties, and we hope that these techniques can be extended to an even larger class of safety and liveness properties in future work. In fact, [11] already builds on this work to proving deadlock freedom in communication fabrics.

2 Methodology

Our microarchitectural models are described by instantiating components from a library of primitives and connecting them. We refer to these models as xMAS networks (xMAS stands for eXecutable MicroArchitectural Specification). The properties to be verified are specified on these networks. For verification, an xMAS network is compiled down into a synchronous model (single clock, edge-triggered Verilog to be precise) which is then verified. We refer to this model as the *synchronous model*.

Although the techniques presented in this paper could be used to directly verify xMAS models instead of the synchronous models, in this work, we simply use the high-level structure in the xMAS models to discover new invariants which are then used in the verification of the synchronous model. We choose this approach partly for engineering convenience (we use a conventional model checker as the trusted engine and view the analysis described in this paper as providing verification hints) and partly because the methods described in this paper cannot be used to prove all properties of interest (in particular liveness). For example we have found that adding the invariants generated in this paper to the synchronous model allows a liveness-to-safety-based liveness model checker to converge on some simple examples whereas without these invariants, the problem would be intractable (see [11] for details).

A nice side effect of this approach is that the invariants we add get checked by the model checker rather than being assumed as given.

Finally, the invariants generated using the techniques in this paper play an essential role in our recent work on scalable liveness verification [11] where these invariants are used to rule out structural deadlocks that are not reachable. This methodology has been used to verify microarchitectural liveness for a number of communication fabrics for future products.

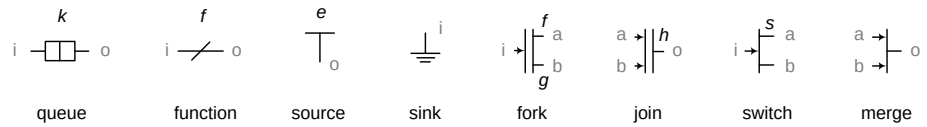

Fig. 2. A key showing the symbols for the various primitives used to model microarchitectural blocks. Section 3 describes these components in detail. The italicized letters (k, f, e, g, h and s) indicate parameters. Whenever we use these primitives in a diagram we need to specify values for these parameters. Often, to avoid clutter we do not show these values explicitly trusting that they are clear from the context. In contrast, the gray letters (i, o, a, and b) in this figure only indicate port names and are only shown to help you understand the formal definitions in Section 3. Observe that for some components such as the fork, we place the parameter close to the "corresponding" port in the diagram.

3 xMAS models

xMAS models are constructed by instantiating components from a library of microarchitectural primitives and connecting them with *channels*. Channels are typed. In the synchronous model, a channel x with type α has two boolean signals x.irdy (for initiator ready) and x.trdy (for target ready) for control and one signal x.data that has type α for the data.

A channel is connected to exactly two components: one component called the *initiator* that "writes" to the channel (via an *output* port) and another component called the *target* that "reads" from the channel (via an *input* port). In the synchronous model, the initiator drives irdy and data signals (and reads trdy) whereas the target drives trdy (and reads irdy and data). Intuitively, a data element (or a packet) is transferred across a channel in those cycles when both irdy and trdy are true. Note that a channel is just two control wires and a data bus and stores no state. A channel is represented in our diagrams by a line.

An xMAS network may be viewed as a directed graph with the components as nodes and channels as edges. Edges are directed from initiator to target.

Example. In Figure 1, there are three channels x, y and z. For channel x, the initiator is the source and the target is the first queue. Thus x is connected to the output port of the source and to the input port of the first queue. The output port of the first queue is connected to channel y.

Figure 2 shows the library of kernel primitives. We formally specify each primitive by providing the synchronous equations that are generated for it. We present this in some detail because the exact definitions are important to understand the invariants that we generate later. These definitions may be skimmed on a first reading.

Queue. In our models, storage is implemented by queues.[1] In terms of interface, a queue is one of the simplest primitives. It is parameterized by a type α of the elements stored in the queue and a non-negative integer k that indicates the capacity of the queue. It has one input port i which is connected to the target end of a channel that is used to write data into the queue. Clearly, this channel must have type α, and for convenience we say that port i also has type α, denoted by $i : \alpha$. Likewise, the output

[1] Our queues are always FIFO i.e. first-in-first-out.

port $o : \alpha$ is connected to the initiating end of the channel that reads data out of the queue. The equations for a queue are:

$$\text{o.irdy} := (\mathbf{pre}(\text{num}) \neq 0) \qquad \text{i.trdy} := (\mathbf{pre}(\text{num}) \neq k)$$
$$\text{enq} \quad := \text{i.irdy } \mathbf{and} \text{ i.trdy} \qquad \text{deq} \quad := \text{o.irdy } \mathbf{and} \text{ o.trdy}$$

where enq and deq are combinational signals defined for convenience, and num is the current occupancy of the queue given by:

$$\begin{aligned} \text{num} := \; & \mathbf{pre}(\text{num}) + 1 \; \mathbf{if} \text{ enq } \mathbf{and\ not} \text{ deq} \\ & \mathbf{pre}(\text{num}) - 1 \; \mathbf{if} \text{ deq } \mathbf{and\ not} \text{ enq} \\ & \mathbf{pre}(\text{num}) \quad \mathbf{otherwise} \end{aligned}$$

where **pre** is the standard synchronous operator that returns the value of its argument in the previous cycle and the value 0 in the first cycle [2]. The elements in the queue are stored in an array called mem of size k of signals of type α. These are indexed by head and tail pointers used for reading and writing, correspondingly.

$$\begin{aligned} \text{head} &:= \mathbf{if} \text{ deq } \mathbf{then\ inc}_k(\mathbf{pre}(\text{head})) \; \mathbf{else\ pre}(\text{head}) \\ \text{tail} &:= \mathbf{if} \text{ enq } \mathbf{then\ inc}_k(\mathbf{pre}(\text{tail})) \; \mathbf{else\ pre}(\text{tail}) \end{aligned}$$

where $\mathbf{inc}_k(x) \equiv \mathbf{if}\ x = k-1\ \mathbf{then}\ 0\ \mathbf{else}\ x+1$. For $j \in \{0, k-1\}$ we have

$$\text{mem}_j := \mathbf{if} \text{ enq } \mathbf{and}\ j = \mathbf{pre}(\text{tail}) \; \mathbf{then} \text{ i.data } \mathbf{else} \text{ pre}(\text{mem}_j)$$

and,

$$\begin{aligned} \text{o.data} := \; & \mathbf{pre}(\text{mem}_0) \quad && \mathbf{if\ pre}(\text{head}) = 0 \\ & \mathbf{pre}(\text{mem}_1) \quad && \mathbf{if\ pre}(\text{head}) = 1 \\ & \qquad \vdots \\ & \mathbf{pre}(\text{mem}_{k-1}) \quad && \mathbf{if\ pre}(\text{head}) = k-1 \end{aligned}$$

Among our set of primitives a queue is the only one that can store data. It is also the only delay element: even if the queue is empty, an input packet is visible at the output only after 1 cycle.

Source. A *source* is a primitive which is parameterized by a constant expression $e : \alpha$.[2] Each cycle, it non-deterministically attempts to send a packet e through its output port. A source has a single output port $o : \alpha$ and is governed by the following equations:[3]

$$\text{o.irdy} := \text{oracle } \mathbf{or\ pre}(\text{o.irdy } \mathbf{and\ not} \text{ o.trdy}) \qquad \text{o.data} := e$$

where oracle is an unconstrained primary input that is used to model the non-determinism of the source in the synchronous model. Each source has its own oracle. We define o.irdy in this specific manner to keep it persistent regardless of the oracle behavior: i.e. once a source makes a value available on the channel, it preserves that value until a transfer. Also note that one can imagine more complex sources which emit arbitrary values from a given set. However, for ease of exposition we stick to the simpler definition above.

Sink. Dually, a sink is a component which non-deterministically consumes a packet. It has one input port $i : \alpha$ and is characterized by the following equation:

$$\text{i.trdy} := \text{oracle } \mathbf{or\ pre}(\text{i.trdy } \mathbf{and\ not} \text{ i.irdy})$$

---

[2] Henceforth we only mention the value parameters of a component and leave the type parameters implicit.

[3] When o.irdy is false, o.data is a don't care. But for brevity in the equations, we always assign to o.data rather than only when o.irdy is asserted.

Function. A *function* primitive is used to model transformations on the data. It is parameterized by a function $f : \alpha \to \beta$. It has an input port $i : \alpha$ and an output port $o : \beta$ and is fully characterized by the following equations:

o.irdy := i.irdy o.data := f(i.data) i.trdy := o.trdy

Note that f is a combinational function that is applied to the input data to generate the output data.

Fork. A *fork* is a primitive with one input port $i : \alpha$ and two outputs ports $a : \beta$ and $b : \gamma$ parameterized by two functions $f : \alpha \to \beta$ and $g : \alpha \to \gamma$. Intuitively, a fork takes an input packet and creates a packet at each output. It coordinates the input and outputs so that a transfer only takes place when the input is ready to send and both the outputs are ready to receive. Formally,

a.irdy := i.irdy **and** b.trdy a.data := f(i.data)
b.irdy := i.irdy **and** a.trdy b.data := g(i.data)
i.trdy := a.trdy **and** b.trdy

Join. A *join* is the dual of a fork. It has two input ports $a : \alpha$ and $b : \beta$ and one output port $o : \gamma$. It is parameterized by a single function $h : \alpha \times \beta \to \gamma$. Intuitively, a join takes two input packets (one at each input) and produces a single output packet. It coordinates the inputs and output so that a transfer only takes place when the inputs are ready to send and the output is ready to receive. Formally,

a.trdy := o.trdy **and** b.irdy b.trdy := o.trdy **and** a.irdy
o.irdy := a.irdy **and** b.irdy o.data := h(a.data, b.data)

Switch. A *switch* is a primitive to route packets in the network. It has an input port i and two output ports a and b, all of type α. It is parameterized by a switching function $s : \alpha \to$ Bool. Informally, the switch applies s to a packet x at its input, and if $s(x)$ is true, it routes the packet to port a, and otherwise it routes it to port b. Formally,

a.irdy := i.irdy **and** s(i.data) a.data := i.data
b.irdy := i.irdy **and not** s(i.data) b.data := i.data
i.trdy := (a.irdy **and** a.trdy) **or** (b.irdy **and** b.trdy)

Merge. Arbitration is modeled by a *merge* primitive that selects one packet among multiple competing packets. A merge has multiple input ports and one output port. Requests for a shared resource are modeled by sending packets to a merge, and a grant is modeled by the selected packet. For simplicity we present here a complete definition of a two-input merge that has two input ports $a : \alpha$ and $b : \alpha$ and one output $o : \alpha$.

o.irdy := a.irdy **or** b.irdy
o.data := a.data **if** u **and** a.irdy
  b.data **if not** u **and** b.irdy
a.trdy := u **and** o.trdy **and** a.irdy
b.trdy := **not** u **and** o.trdy **and** b.irdy

where u is a local Boolean state variable to ensure fairness. We could choose a specific fairness algorithm such as

u := 1 **if** a.irdy **and not** b.irdy
  0 **if not** a.irdy **and** b.irdy
  not pre(u) **if pre**(o.irdy **and** o.trdy)
  pre(u) **otherwise**

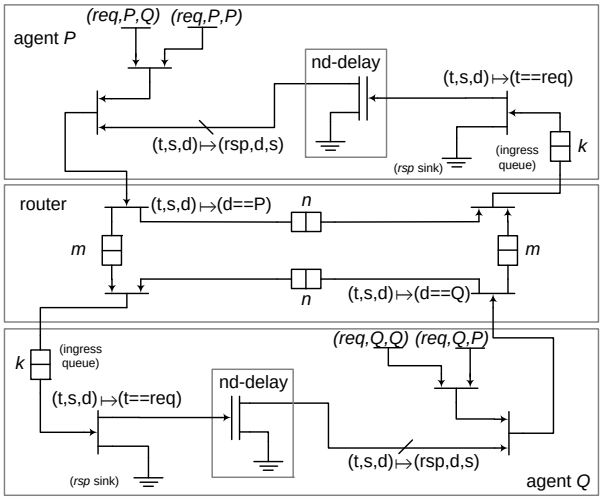

Fig. 3. Example showing a pair of agents communicating over a simple fabric (see text for details). The nd-delay box models non-deterministic delay (the functions of the fork are identity). Since each symbol has a precise formal semantics (see Section 3) this figure is a precise executable description.

Example. Figure 3 shows two agents P and Q communicating via a router. Packets are modeled by triples (t, s, d), where $t \in \{\text{req}, \text{rsp}\}$ is the type of the packet, $s \in \{P, Q\}$ is the source and $d \in \{P, Q\}$ is the destination. Each agent creates new requests for the other agent or for itself. When an agent receives a request (from the other agent or from itself) it produces a response by changing the type of the message and swapping the source and the destination. The response is produced after a non-deterministic delay. The response is sent back to the requester where it is sunk after a non-deterministic delay. The router routes messages according to their destinations i.e. d. (In practice this simplified microarchitecture would not be used since it deadlocks. Deadlocks can be avoided by using virtual channels as we discuss later.)

Finally, it is important to note that since the synchronous model is constructed by instantiating components from a library of primitives and connecting them, it is possible to create models that have combinational cycles. We consider such xMAS models to be ill-formed and do not consider them further in this paper. For a more detailed discussion of the xMAS modeling methodology and for more examples of how common micro-architectural patterns can be modeled with xMAS primitives please refer to [6].

4 Analysis for Channel Properties

A very common verification problem on xMAS networks is to check that all values flowing through a channel satisfy some property. For instance, at the input of an agent, we may wish to check that all packets that arrive have the agent as the destination. Invariants of this kind are called channel properties, and in this section we see how such invariants may be strengthened.

4.1 Channel Properties

If x is a channel that has type α, a *channel property* is a function $p : \alpha \to \{0, 1\}$. Intuitively, if a property p is asserted on a channel x, it means that whenever a valid value is seen on the channel (i.e. x.irdy is asserted), the data on the channel must satisfy p. Formally, a channel property p on a channel x corresponds to the LTL invariant $\mathbf{G}(x.\text{irdy} \implies p(x.\text{data}))$ in the synchronous model. For brevity, we sometimes simply say property instead of channel property.

Example. The verification problem in the introduction corresponds to verifying the channel property $v \mapsto (v = 0)$ on channel z.[4] This corresponds to the LTL property $\mathbf{G}(z.\text{irdy} \implies (z.\text{data} = 0))$ in the synchronous model.

4.2 Propagating Channel Properties

Given a channel property p, we can derive properties on other channels that are "implied" by p using a set of rules. These rules are similar in spirit to Hoare rules [12] used in program verification and are derived syntactically (i.e. no reasoning is involved). The goal is to strengthen the LTL invariant corresponding to p in the synchronous model with the additional invariants obtained from the new channel properties. The soundness of these rules may be verified from the definitions given in Section 3.

Rule for Queue. Since a queue does not modify the data it holds, a property holds on the output of a queue iff it holds on the input.

Example. In our running example (Figure 1), the property $v \mapsto (v = 0)$ holds at z iff it holds at y. Similarly the property holds at y iff it holds at x. It turns out that adding the LTL properties corresponding to the channel properties for x and y, does not make the resulting verification problem on the synchronous model inductive. We need further strengthening, and we return to this topic shortly.

Rule for Function. Given an instance of a function primitive with the parameter $f : \alpha \to \beta$, a channel property p holds at the output iff the property $p' = p \circ f$ holds at the input.

Rule for Switch. Consider an instance of a switch whose switching function is $s : \alpha \to \beta$. The channel property p holds at the output a iff the property $v \mapsto (s(v) \implies p(v))$ holds at the input. Likewise, a property p holds at output b iff the property $v \mapsto ((\neg s(v)) \implies p(v))$ holds at the input.

Rule for Merge. A channel property holds on the output iff it holds on each input.

Rule for Fork. A channel property p holds on the output a of a fork iff $p' = p \circ f$ holds on the input. Similarly, p holds on the output b of a fork iff $p' = p \circ g$ holds on the input.

[4] By "$v \mapsto (v = 0)$" we mean the function that is 1 iff the input is equal to 0, i.e. the function $\lambda v.(v = 0)$ using λ notation.

Rule for Restricted Join. Propagating a property across a join is tricky since the output of a join in general could be functionally dependent on both inputs. However, in our examples drawn from the domain of communication fabrics, joins are only used to control access to resources (e.g. see examples of credit logic and virtual channels in Section 5). Therefore, the join function depends only on at most one input of the join (called the *functional* input) i.e. it is of the form $h : \alpha \to \gamma$ (instead of $h : \alpha \times \beta \to \gamma$). In such cases the other input carries tokens (i.e. values having the unit type). It is easy to detect such joins automatically since the join function h syntactically depends only on one of the inputs. If h is constant, then either input may be taken as the functional input. Given such a join with the restricted function $h : \alpha \to \gamma$, a property p holds at the output iff $p' = p \circ h$ holds at the functional input of the join. Extending propagation to general joins appears to be a hard problem since it involves reasoning about multiple channels.[5]

4.3 Queue Invariants

If we have a channel property p at the output of a queue, using the rule for queues presented above, we also have the property p at the input of the queue. However, simply adding the invariants from these properties to the LTL model does not make the synchronous problem (1-step) inductive. It is easy to see why: Suppose a queue is in a state where it has more than 2 elements. Even if these properties hold at the output and input of the queue, at best they guarantee that only the oldest and youngest element in the queue satify p. They say nothing about the other elements in the queue.

Therefore we need additional invariants to ensure that *every* element stored in the queue satisfies p. For $j \in [0, k)$, where k is the size of the queue, we add the LTL invariant (recall the state variables of a queue from Section 3)

$$\mathbf{G}(\text{used}_j \implies p(\text{mem}_j))$$

where used_j is a predicate over the state that indicates if the jth storage element in the queue is used or not. It is defined as follows:

$$\begin{aligned}\text{used}_j := {} & (\text{head} < \text{tail} \textbf{ and } (\text{head} \leq j \textbf{ and } j < \text{tail})) \textbf{ or} \\ & (\text{head} > \text{tail} \textbf{ and } (\text{head} \leq j \textbf{ or } \quad j < \text{tail})) \textbf{ or } (\text{num} = k)\end{aligned}$$

Along with this, we add the LTL assertions $\mathbf{G}(\text{num} \leq k)$, $\mathbf{G}(\text{head} < k)$ and $\mathbf{G}(\text{tail} < k)$ to ensure that these state variables are within bounds. Finally, we need to add the following invariants to establish the correct relationship between these 3 state variables:

$$\begin{aligned}&\mathbf{G}(\text{head} < \text{tail} \implies \text{head} + \text{num} = \text{tail}) \\ &\mathbf{G}(\text{head} > \text{tail} \implies \text{head} + \text{num} = \text{tail} + k) \\ &\mathbf{G}(\text{head} = \text{tail} \implies \text{num} = 0 \textbf{ or } \text{num} = k)\end{aligned}$$

These assertions are used to ensure that the head and tail pointers behave as expected and provide a local over-approximation of the state-space.

Finally, as an implementation note, it is important that if a bit-vector based reasoning engine is used, the arithmetic in the above invariants be done with adequate precision to avoid overflow.

[5] Even with general joins there is an easy case. If p is a property such that $p' = p \circ h$ depends on only one variable, then it suffices to propagate p' along the corresponding input.

4.4 A Note on Local Invariants

The queue invariants added above block off portions of the unreachable state space that would otherwise lead to false counter examples in induction. However since these invariants are local, any correlation between different queues is not captured. However, this is exactly how a human designer thinks about the system: for example seldom would the correctness of a design depend on say two head pointers in two different queues taking on the same value in all portions of the reachable state space.

Indeed, if the correct operation of a design relies on the correlation between different components, typically this is enforced in the design by some explicit communication structure between them. A common case in our models for this case is when the occupancy of multiple queues are correlated in the reachable state space. We study this problem in the next section where we follow this communication trail to infer the appropriate invariants.

The queue is our main state holding element. Among all the primitives, the only other interesting state holding element is the merge which maintains state for fairness. If the merge has multiple inputs, then the appropriate local invariants for the fairness logic need to be added. (For the particular 2-input merge presented in Section 3, we do not need to add constraints for the u variable since it can take on both 0 and 1 values in the reachable space.)

4.5 Propagation Algorithm

Given a property p on a channel, we try to maximally propagate it backwards using the obvious iterative algorithm. This is done by looking at the initiator of the channel, and applying the corresponding rule from Section 4.2. This creates new properties at the inputs of the initiator. This process is repeated for each newly added property. If the initiator is a source, then the property is not (cannot be!) propagated further. If the xMAS network has a directed cycle, the above process will not terminate. We handle this in practice by recording the "parent" and stopping when a cycle is encountered. After all properties have been propagated in this manner, for each queue in the system, we add the local invariant according to the scheme described in Section 4.3 for each property at the output of the queue.

Since property propagation terminates naturally in *acyclic* xMAS networks we can formulate a completeness theorem for such networks:

Theorem 1 (Completeness for Acyclic Networks). *Given an acyclic* xMAS *network $\mathcal{N}$ where all joins are restricted, and a channel property p on a channel x in $\mathcal{N}$ that holds, the above algorithm adds sufficiently many invariants to make the synchronous problem 1-step inductive.*

4.6 Remarks on Cyclic xMAS Networks

In cyclic xMAS networks, the backward propagation of Section 4.5 does not terminate naturally and so it is not possible to prove a completeness theorem along the lines of Theorem 1. Although the backward propagation is terminated for a property when a loop is encountered, in practice, most properties become tautologies during propagation (as in the tautology example above) i.e. before a loop is encountered. However, for those that do not become tautologies, it may be necessary to add additional channel invariants on loops to break the cyclic propagation of properties.

Example. Figure 4 shows a fragment that takes an input packet n which is an integer (say 10 bits wide) and sends it around the loop n times. The iteration is managed by first transforming the integer n into a tuple (n, i, j) with $i = 0$ and $j = n$. Each time the tuple goes around the loop, i is incremented and j decremented. When j becomes 0, the tuple exits the loop. A similar pattern can be used for any iterative computation (e.g. computing the GCD of two numbers or square roots by Newton's method, etc.) but we choose this example to keep the properties simple.

Now consider the properties $(n, i, j) \mapsto (j = 0)$ (call this property l) and $(n, i, j) \mapsto (i = n)$ (call this m) at channel z. Although they are syntactically similar, l is easy to discharge: simply propagating it past the switch results in a tautology. On the other hand, m does not become a tautology during one iteration through the loop and we stop propagating it when we encounter the channel x a second time.

We can make this inductive by adding a "loop invariant." In this case if we add the property $(n, i, j) \mapsto (n = i + j)$ (call it k) to channel y, then this system becomes inductive (though neither property m nor property k can be propagated until it becomes a tautology). One advantage of the present methodology is that the added channel property is an invariant at a convenient level of abstraction since it only deals with data. No "control" invariants had to be added.

Finally, we note that in most practical examples from the domain of communication fabrics, packets loop at most k times, where k is typically small (i.e. 1 or 2). We can handle such cases automatically by continuing property propagation even when a cycle is first encountered and stopping only when a property is propagated over a channel for the $(k+1)$th time. This corresponds to implicitly unrolling the diagram $k+1$ times.

Example. In Figure 3 packets from the self request source in agent P, i.e. the $(\text{req}, \text{P}, \text{P})$ packets, traverse the ingress queue in P twice, first as a request, req, and then as a response, rsp. Now consider property l defined as follows:

$$(t, s, d) \mapsto (s = P) \textbf{ or } (s = Q)$$

To make this property non-trivial, assume that s and d take on values from a larger set of addresses that includes P and Q. To prove that l holds at the input of rsp sink of P requires propagation twice through the ingress queue of P.

4.7 Implementation Notes

Even in moderately sized xMAS models that arise in practice, the propagation algorithm of Section 4.5 leads to the creation of a large number of channel properties. This slows down the propagation algorithm and furthermore the corresponding invariants in the

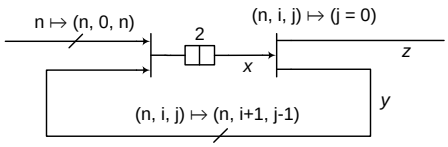

Fig. 4. A fragment of an xMAS diagram showing a common pattern for iterative computations. Due to the presence of the loop some properties on channel z may not be inductive. However, by adding a "loop invariant" i.e. a suitable property on channel x or y, the set of properties may be made inductive with little manual effort.

synchronous model can overwhelm the inductive verification engine. We use a number of heuristics to cut down on the number of invariants generated (without destroying inductivity of the whole set).

Tautological Properties. Many channel properties can be discharged locally since during the propagation process they become tautologies i.e. the constant 1 function. Therefore we use a reasoning engine to detect tautological properties and do not propagate them further. This is an important optimization in practice.

Example (tautology). In the example of Figure 3, let l be the property $(t, s, d) \mapsto (d = P)$ at the ingress queue of agent P. If we propagate l backwards through the queues and switches in the router using the above algorithm, we find that the properties that are obtained from l at each input of the router are of the form $(t, s, d) \mapsto ((d = P) \implies (d = P))$ which is a tautology. These tautological properties need not be propagated further.

Redundant Properties. Using simple structural hashing (similar to what is done in And-Inverter Graphs), it is often possible to efficiently eliminate redundant channel properties on a channel, and not propagate them further.

Invariants from Channel Properties. Somewhat suprisingly, we need not add any invariants corresponding to propagated channel properties to the synchronous model! This is because the value of the data signal of any channel in the xMAS model is a combinational function of the storage of a queue (i.e. the mem array) and the control state stored in the merges, sources and sinks. Thus as long as the queue invariants on the contents of the mem arrays are present (Section 4.3), the invariants corresponding to the propagated channel properties are not necessary to preserve inductivity. This optimization is a very important one in practice since it dramatically cuts down on the number of invariants that need to be checked during induction.

4.8 Comparison with Conventional Invariant Strengthening

There is a subtle but important difference between conventional invariant strengthening using weakest preconditions (e.g. [4, §3.4]) and that based on the channel property propagation presented here. In channel property propagation we do not work directly with a program that describes how the state of the xMAS system changes; indeed the program that directly corresponds to such a description would be the synchronous model. Instead, we work with the xMAS network which is only an indirect description of how the state of the xMAS system changes. However, it is a direct description of how the "state" of an individual packet changes as it flows through the xMAS network. Intuitively, the channel propagation algorithm uses the high-level information of how each packet evolves to make a claim on how the system as a whole evolves. Consequently, just the weakest pre-condition computation is not enough, and we need the extra invariants for the queues as in Section 4.3.

On the other hand, if one wished to apply conventional invariant generation to xMAS systems, the obvious choice would be to work with the synchronous model. Indeed the k-induction based techniques such as those used by ABC are based on automatic strengthening of invariants on the synchronous model [15]. To see why induction on the synchronous model is not useful, observe that the strengthened invariants obtained by one pass of the propagation algorithm of Section 4.5 do *not* correspond to those obtained by one pass of induction strengthening on the synchronous model even for an acyclic xMAS network if it has queues. Indeed one would have to unroll the synchronous

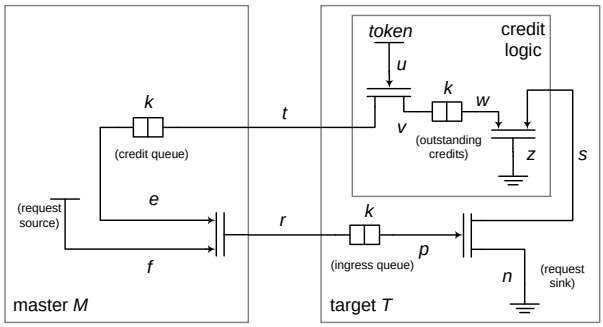

Fig. 5. Credits introduce correlation between the occupancies of different queues. Both joins are restricted in the sense of Section 4.2 since at least one input is a token.

model many times (the exact number would depend on the maximum total capacity of queues on any path from a source to a sink). This of course assumes that loop-free (i.e. pair-wise uniqueness) constraints are added to the induction unrolling to make it complete [15]. Without uniqueness constraints, no amount of unrolling would suffice in general since we have systems where a packet sits in a queue indefinitely due to non-determinism of sinks.

5 Invariants from Flows

As remarked in Section 4.4, if the correct operation of a design relies on correlation between state variables in different components then in a real design there is usually an explicit communication mechanism between them for coordination. In this section we present an algorithm to analyze a commonly-occuring communication of this form that leads to correlation among the occupancies of different queues in the system. The invariants added by this analysis allow us to prove an important class of safety properties that check that the queues in a system are sized correctly. Such safety properties are necessary for reasoning about liveness.

5.1 The Basic Idea

Example (credits). Consider the xMAS network shown in Figure 5 which shows a master agent M communicating with a target T. The credit logic portion of T issues at most k outstanding credits to M at any given time. Credits are modeled as values of the unit type called *tokens*. M has to wait for a credit before it can send a request to T. The purpose of this mechanism is to ensure that there is always room in T's ingress queue for requests from M i.e. nothing gets stuck on channel r. Thus r is *non-blocking* i.e. satisfies the LTL property: $\mathbf{G}\ (r.\text{irdy} \implies r.\text{trdy})$. Credits are freed up when data is read from the ingress queue of T.

The non-blocking property on r is not inductive. However, by adding the invariant

$$\mathbf{G}(\text{num}_c + \text{num}_i = \text{num}_o)$$

to the synchronous model, the problem becomes inductive.[6] Here, num_c is the num variable of the credit queue in M, num_i the same for the ingress queue in T and num_o for the outstanding credits queue in T.

[6] assuming that the (local) assertions $\mathbf{G}(\text{num} \leq k)$ for each queue have already been added.

The question now is how can we detect such global assertions automatically? If x is a channel, let λ_x denote the number of packets that have been transferred on x upto a given point in time (i.e. λ_x is the count of the number of cycles so far in which x.irdy and x.trdy were both asserted). Now, from the equations of a join in Section 3 it is easy to see that either a transfer happens on both inputs and the output of a join or there is no transfer at any input or the output. Thus for the two joins in Figure 5 we have,

$$\lambda_e = \lambda_f = \lambda_r \qquad \text{and} \qquad \lambda_s = \lambda_w = \lambda_z.$$

Similarly, for a fork it can be verified that either a transfer happens on both output and the input or there is no transfer at all. Thus for the two forks in the system we have the equations

$$\lambda_u = \lambda_t = \lambda_v \qquad \text{and} \qquad \lambda_p = \lambda_n = \lambda_s.$$

A queue is more interesting. Any packet that enters a queue is either still in the queue or has exited through the output channel. Thus from the three queues in Figure 5 we get the following three equations:

$$\lambda_r = \text{num}_i + \lambda_p \qquad \lambda_t = \text{num}_c + \lambda_e \qquad \lambda_v = \text{num}_o + \lambda_w$$

From these 7 equations, we can eliminate the λ variables to get the desired relationship between the num variables. This can be done automatically in the following manner. First we create a matrix from the equations where all λ variables are to the left and all num variables are to the right. Then this matrix is converted to Reduced Row Echelon (RRE) form by Gaussian elimination (over the rationals). Finally, we select the equations from the RRE form which involve only the num variables (i.e. the coefficients of all λ variables are 0).

Note that the λ variables are unbounded and by this elimination process, we are only left with relations in only the num variables (which are bounded by the size of the queues). Hence these relations can be added as invariants to the synchronous model.

The technique described in this section resembles generation of place invariants in Petri nets [7]. However, rather than modeling the communication fabric with Petri nets (which leads to an overhead of using explicit back-pressure arcs and complexity in modeling the data-path) we derive those invariants directly from more compact and natural xMAS specifications.

5.2 Shared Communication

In the presence of shared communication channels the approach presented above needs to be refined.

Example (virtual channels). Virtual channels lead to sharing. They are commonly used in communication fabrics to multiplex multiple logical streams onto a single physical link with the guarantee that even if one stream is blocked at the receiver, the other streams still make progress [9].

Figure 6 shows a simple example of virtual channels. A master agent M sends two types of messages A and B (think of these as perhaps requests and responses) to a target T over a single channel r. The ingress switch in T routes A and B packets to their respective ingress queues. The credit pattern of Figure 5 is used to ensure that

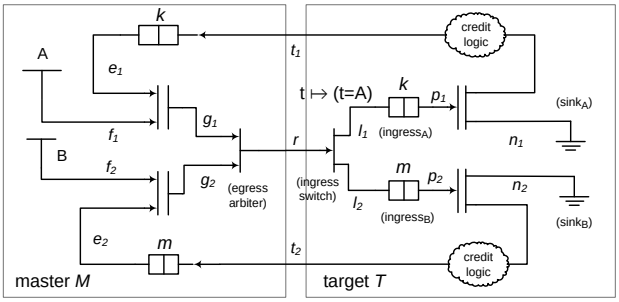

Fig. 6. Example of a shared communication path that requires more precise flow analysis. The credit logic bubbles encapsulate the logic shown in the credit logic box of Figure 5. The switch in the target routes A packets to l_1 and B packets to l_2. The joins in M are restricted and have the identity function. Also note that although this example *looks* cyclic, it is not.

whenever a packet is presented to the egress arbiter of M, there is guaranteed to be room in the corresponding ingress queue in T. Thus channel r is non-blocking.

Once again, the non-blocking property on r is not inductive. However, if we add the invariants

$$\mathbf{G}(\mathrm{num}_{c_A} + \mathrm{num}_{i_A} = \mathrm{num}_{o_A})$$

and

$$\mathbf{G}(\mathrm{num}_{c_B} + \mathrm{num}_{i_B} = \mathrm{num}_{o_B})$$

for each credit loop, then the problem becomes inductive. Here, num_{c_A} refers to the num variable of the credit queue in M associated with the A packets, and so on. However, if we try the approach from the previous example (with suitable extensions for merges and switches) we find that we can only derive the weak invariant:

$$\mathbf{G}(\mathrm{num}_{c_A} + \mathrm{num}_{i_A} + \mathrm{num}_{c_B} + \mathrm{num}_{i_B} = \mathrm{num}_{o_A} + \mathrm{num}_{o_B})$$

which is not enough to prove the property.

We can improve the precision of the analysis by defining a λ variable *per flow* through a channel. For example we know that two types of values flow through channel r. Therefore we introduce two variables λ_r^A and λ_r^B for r where λ_r^A is a count of the number of cycles when r.irdy and r.trdy have been asserted *and* r.data was equal to A. Similarly, λ_r^B for B. Since there are two flows through r, we assume that two flows are possible through g_1 and g_2 and through f_1 and f_2 and associate two λ variables from each of these channels: one for A and one for B. For all the other channels, we associate only a single λ variable since there are only single flows through them. (We will see later how to automatically figure out the number of flow variables needed.)

Since the ingress switch in T routes A to channel l_1 and B to channel l_2, we have

$$\lambda_r^A = \lambda_{l_1} \qquad \text{and} \qquad \lambda_r^B = \lambda_{l_2}$$

For a merge, a packet at the output must come from one or the other input. Therefore, we have the following equations for the egress arbiter in M:

$$\lambda_{g_1}^A + \lambda_{g_2}^A = \lambda_r^A \qquad \text{and} \qquad \lambda_{g_1}^B + \lambda_{g_2}^B = \lambda_r^B$$

Observe that one input of each join in M is a token input i.e. the joins are restricted. We have the following relations between the outputs and the functional inputs:

$$\lambda^A_{f_1} = \lambda^A_{g_1} \qquad \lambda^B_{f_1} = \lambda^B_{g_1} \qquad \lambda^A_{f_2} = \lambda^A_{g_2} \qquad \lambda^B_{f_2} = \lambda^B_{g_2}$$

and the following relations between the token inputs and the outputs:

$$\lambda_{e_1} = \lambda^A_{g_1} + \lambda^B_{g_1} \qquad \lambda_{e_2} = \lambda^A_{g_2} + \lambda^B_{g_2}$$

Each source however generates only one type of packet. Therefore we can set the other λ variable on the output channel to zero i.e. $\lambda^B_{f_1} = 0$ and $\lambda^A_{f_2} = 0$.

All the other components only interface with channels carrying single flows, and we add equations as in the credit example. Finally, as before, by eliminating the λ variables using Gaussian elimination, we obtain the desired relations among the num variables.

5.3 Algorithm for Discovering Flow Invariants

Formally, if x is a channel that has type α, a *flow* on x is a function $p : \alpha \to \{0, 1\}$. Although the formal definition of a flow is the same as that of a channel property, the two notions should not be confused. Intuitively, a channel property is an assertion about *all* values that flow through a channel whereas a flow is a means of keeping track of a set of values that flow through a channel. Our goal in what follows is to compute the set of flows for each channel and the equations relating the λ variables for these flows.

Step 1. Sort the xMAS graph in reverse "topological" order starting from the channels that feed the sinks using the textbook depth-first-search (DFS) based topological sort algorithm [8, §22.4]. If the xMAS network is cyclic, this has the effect of topologically sorting the DAG obtained by deleting the backedges in the DFS.

Step 2. Assign the constant 1 function as the flow on the inputs to the sinks and on the backedge channels. Now we process each component in the network in the reverse "topological" order computed above by applying the following rules to propagate flows (we use the same parameter names as in Figure 2 and use port names to refer to the corresponding channels):

Queue. For each flow p on the output channel o, we create a new flow p on the input channel i. We also add a new state variable called num^p to the queue that tracks how many elements satisfying p are currently in the queue. We also add an assertion that equates num^p to the number of elements that satisfy $(\text{used}_j \implies p(\text{mem}_j))$ in the queue. Finally, we add the equations:

$$\lambda^p_i = \text{num}^p + \lambda^p_o$$

and

$$\text{num} = \sum_p \text{num}^p.$$

We call equations of the last type *queue occupancy equations* since they relate the occupancy of each flow to the total occupancy of the queue.

Function. For each flow p on the output channel o, we create a new flow $p' = p \circ f$ on the input channel i and add the equation $\lambda^{p'}_i = \lambda^p_o$.

Switch. For each flow p on the output channel a, we create a flows $p' = v \mapsto (s(v)\,\mathbf{and}\,p(v))$ on the input channel i and add the equation $\lambda_i^{p'} = \lambda_a^p$. Similarly for flows on output b.

Merge. For each flow p on the output channel o, we create a flow p on input a and another flow p on input b and add the equation $\lambda_a^p + \lambda_b^p = \lambda_o^p$.

Fork. For each pair (p, q) where p is a flow on output a and q on output b, we create a new flow $r = v \mapsto (p(f(v))\,\mathbf{and}\,q(g(v)))$ on input. For each flow p on output a, we add the equation $\lambda_a^p = \Sigma_r \lambda_i^r$ where r ranges over flows that were added to i due to p. Similar equations are added for each flow on b.

Join. Once again, we limit our attention to restricted joins. For each flow p on the output channel o, we add a flow $p' = p \circ h$ to the functional input (suppose it is the input a). We add the constant 1 flow to the other input (i.e. b) and the equations $\lambda_a^{p'} = \lambda_o^p$ and $\lambda_b^1 = \Sigma_p \lambda_o^p$ where p ranges over all the flows on o.

Source. For each flow p in the output o, we check if $p(e)$ is true or not. If $p(e)$ is false, then we add the equation $\lambda_o^p = 0$ and mark p as *dead*.

During the above process, each time a new flow is created, we record its parent(s). Furthermore, if a new flow is unsatisfiable i.e. the constant 0 function we mark it dead and do not propagate it further.

Step 3. If the xMAS network is cyclic, for each channel x that is a backedge, the above propagation process adds new flows. These need to be related to the constant 1 flow which was assigned before starting propagation. Therefore we add $\lambda_x^1 = \Sigma_p \lambda_x^p$ where p ranges over all the flows added to x during propagation.

Step 4. If all children of a flow p at a channel x are dead, we mark x as dead as well and add the equation $\lambda_x^p = 0$. We repeat this process until no new flows can be marked dead.

Theorem 2 (Inductivity). *The set of equations obtained by this process is an inductive invariant of the synchronous model.*

Step 5. Finally, the λ variables are eliminated as explained before to obtain relations between the num variables. Note that it is possible that there are no relations among the num variables (e.g. Figure 1).

Remark 1. Since the λ variables correspond to channels which hold no state, eliminating them does not destroy inductivity.

Remark 2. The flow invariants discovered using the above algorithm generates invariants capturing relations between the occupancies of different queues. Note that these relations are in general control dependent: based on flow analysis starting from switches that makes control decisions using data values of the packets the algorithm accounts for different types of messages residing in the queues (as illustrated by num_{c_A} and num_{c_B} counters for messages of type A and B in the shared communication example of Figure 6).

5.4 Implementation Notes

Queue occupancy equations. We found it useful to *prevent* queue occupancy equations from participating in the Gaussian elimination. If they are allowed to participate in the elimination, it often leads to equations with more terms than necessary where the occupancy of a particular flow i.e. num^p is expressed in terms of num and the

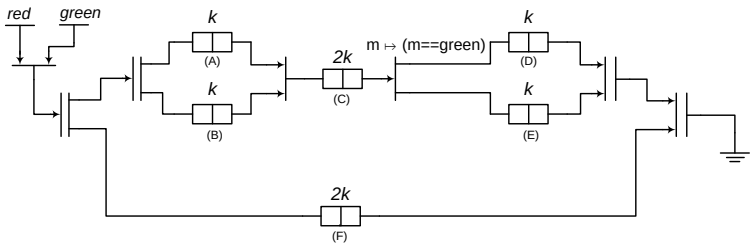

Fig. 7. A synthetic example that leads to fractional coefficients in the flow relations. The functions on the forks are identity and the functions on the joins are constants.

occupancies of the remaining flows. This leads to poorer performance of the SAT solver when checking these invariants on the synchronous model.

There are secondary benefits of excluding occupancy equations from the elimination process. The remaining equations for a flow are independent of (i.e. have no variables in common with) the equations for other flows. This leads to (i) faster run-times for Gaussian elimination and (ii) an easier to understand set of equations after elimination.

Of course after elimination is done, we add the queue occupancy equations back into the set of equations.

Sparse Matrix for Gaussian Elimination. It is essential to use a sparse matrix representation for the Gaussian elimination since in real examples there are thousands of equations and variables. However, we did not implement any techniques to preserve or increase sparsity during elimination.

Non-unit Coefficients. From the equations it may appear that only 1 and -1 appear as coefficients in the equations. Indeed, in our early experience this was the case. However, on one industrial example, during the elimination process we encountered fractional coefficients, although in the final equations (i.e. in the equations that contain no λs), there were no fractions. However, we can create synthetic examples where the final relations may have fractional coefficients.

Example (fractional coefficients). In the example of Figure 7, after Gaussian elimination we have the following invariant:

$$\text{num}_F = \frac{1}{2}(\text{num}_A + \text{num}_B + \text{num}_C + \text{num}_D + \text{num}_E)$$

Indeed, for every packet injected into queue F there are two packets injected into the set of queues $\{A, B, C, D, E\}$.

Sensitivity to DFS *for cyclic networks.* If the network contains a cycle, then set of invariants that is found may depend on the set of backedges discovered during DFS in Step 1 of the algorithm in Section 5.3. In turn, the set of backedges discovered during DFS depends on implementation details such as the order in which edges and nodes are traversed. In some examples, an arbitrary choice of backedges may lead to important invariants not being found.

Example (bad backedges). Suppose that the virtual channel example of Figure 6 is embedded in a larger cyclic network[7] and further that channel g_2 becomes a backedge.

[7] A typical example of such a system would be when separate virtual channels are used for req and rsp in Figure 3 to avoid deadlock.

In this case, the precise invariant for the B packets i.e.

$$\mathbf{G}(\text{num}_{c_B} + \text{num}_{i_B} = \text{num}_{o_B})$$

would *not* be discovered by the algorithm. If however, a channel "outside" the virtual channel mechanism (for e.g. channel n_2) were to be a backedge, it would not adversely impact the set of invariants generated.

Therefore, it is useful to control the set of backedges in order to obtain a good set of invariants. But how to control the set of backedges? In terms of implementation, it is simplest to ask the user to provide this set. In our experience, even on very large examples this is very easy for the user to do. Usually this amount to finding a channel in an agent that is the input of a function primitive that converts requests to responses and is not part of a virtual channel mechanism. The system then merely has to ensure that all cycles are broken by the set of backedges provided by the user.

However, with a slightly more sophisticated implementation, in most cases, we need not rely on user annotation at all. The main idea is to grow a set of backedges (we call this a *cutset*) using several passes of a modified DFS algorithm. Each pass contributes new backedges to the cutset according to a pass-specific policy. A *policy* is simply a rule that says if a given channel is eligible to be in the cutset or not. For example, a policy could be that the channel has to be an input channel of a function primitive.

The standard DFS is modified as follows. First, it takes an additional input which is the policy that will be used to grow the cutset in this pass. Second, it is not allowed to traverse a channel that is already in the cutset. Third, in addition to starting DFS from the input channels of sinks, we also start from the channels in the cutset. Fourth, when a backedge channel is encountered during DFS, if it satisfies the policy, it is added to the cutset for the next round. However, if it does not satisfy the policy, the "obligation" to break the cycle is pushed up the stack of DFS calls until a channel is found that satisfies the policy. This channel is added to the cutset (instead of the original backedge). Of course, it is possible that no such channel is found i.e. there is a cycle that cannot be broken using only channels that satisfy the policy.

We experimented with different schemes and found the following scheme to provide the best results on our large set of examples. We initialize the cutset to the set of backedges provided by the user (which may be empty). Then we do the first modified DFS pass with the policy of selecting channels that are inputs to function primitives. Note that this does not result in every input channel of a function primitive to be added to the cutset — only those that are useful in breaking cycles get added.

It is possible that not all cycles are broken by the channels in the cutset after the first pass. Therefore, we do a second pass with the policy of selecting channels that are inputs of queues. Since we require that every cycle in an xMAS network be broken by a queue (otherwise there would be a combinational cycle in the synchronous model), as a result of this pass, the cutset should contain enough channels to break all cycles.

Finally, we topologically sort the acylic network obtained by deleting the channels in the cutset. This the starting point for Step 2 in Section 5.3.

This scheme is very useful in practice. In all but a set of related examples, this heuristic was able to find the set of backedges that provided all the invariants necessary without any annotation from the user at all. In that remaining example, the cutset was seeded with an initial annotation (which was straightforward for the user to provide).

5.5 Uses and Limitations of Numeric Equality Flow Invariants

The flow invariants discovered by the method in Section 5.3 are are extremely useful in practice. We use them to verify critical channel non-blocking properties and to rule out unreachable structural deadlocks in our work on efficient automatic proofs of liveness [11]. This has enabled proofs of liveness on industrial examples which were previous intractable.

However, the flow invariants we find are all numeric equality invariants. It is possible to construct examples where simply keeping count is not sufficient to prove interesting non-blocking properties. One may have to reason about order of packets or even about relative timing through different branches in the xMAS diagram. The simple nature of numeric equality invariants renders them inadequate in these cases as demonstrated by the following example:

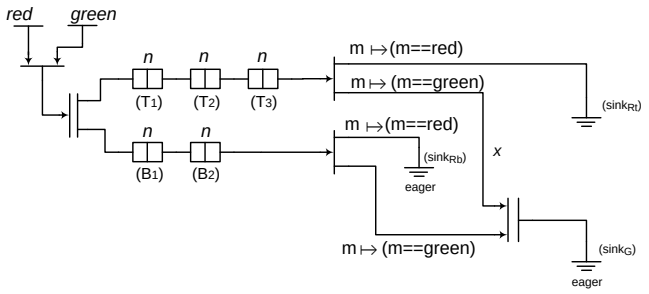

Fig. 8. An example for which non-blocking of channel x cannot be inductively proven using numeric equality invariants which do not capture relative order of tokens in the queues. The functions for the forks are identity and the function for the join is constant.

In the example of Figure 8 the sources non-deterministically inject red and green tokens that are forked into the chain of queues at the top of the figure (3 queues T_1, T_2, T_3) and a shorter chain of queues at the bottom (2 queues B_1, B_2). The red tokens from the top and the bottom chains are sunk independently, while the green tokens are synchronized by a join before they are sunk. The sink of green tokens and the bottom sink of the red tokens at the bottom of the figure are eager, i.e. they are always ready to receive packets.

The top input channel of the join, x, is non-blocking. Indeed, since the chain of queues at the bottom is shorter and red tokens are sunk from the bottom chain not slower than from the top one (since sink_{R_b} is eager, while sink_{R_t} is not), by the time a green token reaches the head of queue T_3 there must be a green token at the head of queue B_2. Since sink_G is eager these two green tokens are sunk right away. Hence the green token from the head of T_3 is sunk immediately after it enters the head of T_3 and channel x is non-blocking.

The non-blocking property of channel x cannot however be proven inductively with the invariants described in this paper. The algorithm of Section 5.3 discovers the following equality invariant on the number of green tokens in the chains of queues.

$$\text{num}_{T_1}^{\text{green}} + \text{num}_{T_2}^{\text{green}} + \text{num}_{T_3}^{\text{green}} = \text{num}_{B_1}^{\text{green}} + \text{num}_{B_2}^{\text{green}},$$

This is insufficient to prove non-blocking of x, since this invariant contains no information about relative order of the green and red tokens in the queue chains.

It is an open problem to automatically generate suitable invariants for examples such as this one. One interesting approach may be to reinterpret a numeric invariant as a statement about sequences where a "num" variable denotes the sequence of packets in a queue and addition denotes sequence concatenation.

6 Experimental Results

6.1 Micro-benchmarks

Since the state-of-the-art model checking algorithms are unable to converge on any of our real examples, we present a comparison on the small examples from this paper. Table 1 shows the results of running ABC (version 91206p) on several examples (parameterized on k) without the addition of invariants as described in this paper.

The first example is from Figure 1 where each queue is of size k. In the second example we have a series of k queues (similar to Figure 1). In the third we check the property in the example of Section 4.5, but to make the example more realistic we set the source and destination to be 2 bits wide in the packet. Fourth and fifth are self-explanatory. In the last example we add k queues on the channel r in Figure 6.

Column i in the table is the number of primary inputs (oracles); r and n are number of registers and AIG nodes (after synthesis). A depth of (m, n) means interpolation converged in n iterations when starting from a BMC of depth $1 + m$. The time is in seconds (on a 3GHz Intel Xeon CPU) with a timeout of 300 secs indicated by a dash (and we show the final BMC depth in the previous column). Note that interpolation times out on many examples.

The first three rows correspond to examples for channel propagation. In all cases when we add the invariants as described in Section 4, ABC is able to solve the problem in no time. Even if we set $k = 100$, the first example is solved in 7 seconds, the second in 1 second and the third in 40 seconds.

The remaining rows correspond to examples for flow invariants. Again without the flow invariants, interpolation has a hard time. However, in these examples we found that BDD-based reachability could solve these quickly. In all cases in our experience, the algorithm for discovering flow invariants finds exactly those invariants that are interesting. For example, in the fifth example, there are initially 43 variables and 32 equations. After elimination, we are left with two invariants (with 6 terms) corresponding to the

Table 1. Comparison with interpolation on micro-benchmarks. See Section 6.1 for details. Most rows have two data points corresponding to different values of the parameter k of the corresponding example.

Description	k	i	r	n	depth	time	k	i	r	n	depth	time
Two queues of size k	3	2	49	348	(5, 12)	23	4	2	63	381	BMC 24	–
k queues of size 2	3	2	49	302	(9, 12)	76	4	2	64	396	BMC 20	–
Figure 3 with all queues sized to 2							-	8	99	659	BMC 11	–
Figure 5	8	4	14	104	(13, 9)	38	12	4	14	121	BMC 27	–
Figure 6 with all queues sized to 2							-	4	17	95	(7, 4)	40
above with k queues on r	1	4	24	135	(6, 7)	112	2	4	29	151	BMC 13	–

two credit loops as expected. The time needed for both property propagation and for detecting flow invariants is neglible.

6.2 Experience on Real Examples

We have applied the techniques described in this paper to verify a number of abstract models used to validate the microarchitecture of future designs. These are drawn from the domain of communication fabrics and are characterized by deeply pipelined logic for multi-phase transactions, presence of ordering logic and several virtual channels, and peer-to-peer traffic. Even in minimal configurations, there are tens of simultaneous transactions in flight.

As a data point, previously on one of our simpler examples comprised of 75 xMAS primitives (including 24 queues) we were able to obtain a proof of a critical non-blocking property,[8] only by severely limiting the state space by reducing the number of simultaneous outstanding transactions an agent can issue. The proof was obtained with an explicit state model checker with maximal reachability depth of 159 in 12 hours using 17GB of memory. In contrast, using the flow analysis from Section 5 on the *original* model, 16 flow relations are discovered (from an initial set of 176 equations on 220 variables) and ABC solves the resulting problem in 4.5 sec.

As a second data point, on one of our most complex examples with 680 xMAS primitives (including 131 queues) which is a model of multiple agents connected by two cascaded pipelined switches, a recent version of ABC is unable to converge in 4 weeks using either interpolation or Bradley's new algorithm for property driven reachability [5, 10]. Due to the memory required, explicit state model checking is not an option, even if we severely limit the functionality. BMC reaches 10 frames in 20 minutes, and saturates at 16 frames in a week. In contrast, with the flow analysis 271 flow relations are discovered (from an initial set of 2427 equations in 3298 variables) and ABC solves the resulting problem in 16 hours. This verification run proves more than 17,000 assertions including non-blocking properties of the channels, the 271 flow relations, local invariants of xMAS primitives, channel persistency properties, and mutual exclusivity and complete coverage of the switch output channels.

As seen from these data points, a big advantage of this technique is its robustness and scalability. Rather than be limited to minimal configurations (and consequently reduced concurrency), we can now verify more realistic models. Channel property verification is robust and most properties are discharged automatically. For a few properties we need to add additional channel properties to break loops. However, these invariants are natural and easy to add since they only talk about data and do not involve control at all. Finally, although it may appear that flow invariants could lead to scalability problems, so far we have not encountered any problems, even on our larger examples with dozens of flow invariants, many with tens of terms. For such problems, an inductive engine that assumes all invariants in one cycle and then checks each invariant separately in the following cycle appears to be scalable.

7 Conclusion and Future Work

The concrete proposals for capturing and exploiting high-level information in hardware models presented in this work have proved very useful in practice allowing us to prove

[8] to check for adequate buffering to avoid deadlocks

with little computational effort many hard sequential properties on real microarchitectural models which could not be proved before. The benefit seems to be in separating control from data and exploiting knowledge of the control to reduce the problem to a combinational one on the data.

The invariants we add may be seen as providing a *bag* abstraction for queues. Although the bag abstraction has proven adequate to handle our current examples from communication fabrics, as we saw in Section 5.5, there are examples where it is necessary to reason about sequences and even relative timing. It would be interesting to explore methods to exploit the high-level structure of the models to obtain richer invariants that allow us to handle a larger class of systems.

Finally, a lot of the computational overhead of verifying the synchronous model may be eliminated by switching to an axiomatic semantics for xMAS models for a more direct verification. This may also be an interesting direction for building bridges to the RTL implementation.

References

1. J. Baumgartner et al. Scalable conditional equivalence checking: An automated invariant-generation based approach, FMCAD 2009:120-127.
2. A. Benveniste et al. The synchronous language twelve years later, in *Proc. of the IEEE*, 91(1):64–83, Jan 2003.
3. Berkeley Logic Synthesis Group. http://www.eecs.berkeley.edu/~alanmi/abc/
4. N. Bjørner, A. Browne and Z. Manna, Automatic Generation of Invariants and Intermediate Assertions, in Theor. Comput. Sci. 173(1): 49-87, 1997.
5. A.R. Bradley. SAT-based model checking without unrolling, VMCAI 2011: 70-87.
6. S. Chatterjee, M. Kishinevsky and U.Y. Ogras, Quick formal modeling of communication fabrics to enable verification, HLDVT 2010:42–49.
7. J.M. Colom and M. Silva. Convex geometry and semiflows in P/T nets, in *Proc. of Appl. and Theory of Petri Nets*, pp. 79–112, 1991.
8. T.H. Corman et al. *Introduction to Algorithms*, Second Edition, MIT Press, 1990.
9. W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.
10. N. Eén, A. Mishchenko, and R. Brayton, Efficient implementation of property directed reachability, IWLS'11.
11. A. Gotmanov, S. Chatterjee and M. Kishinevsky, Verifying Deadlock Freedom of Communication Fabrics, VMCAI 2011: 214-231.
12. C.A.R. Hoare. An axiomatic basis for computer programming, *Comm. of the ACM*, 12(10):576580,583 1969.
13. R. Jhala and K.L. McMillan. Microarchitecture Verification by Compositional Model Checking, CAV 2001: 396-410.
14. R. Kaivola et al. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation, CAV 2009: 414-429.
15. M. Sheeran, S. Singh, G. Stålmarck, Checking safety properties using induction and a SAT-solver, FMCAD 2000, LNCS:1954.