Verifying Deadlock-Freedom of Communication Fabrics

Alexander Gotmanov[1], Satrajit Chatterjee[2], and Michael Kishinevsky[2]

[1] Intel Corporation, Moscow, Russia
alexander.gotmanov@intel.com
[2] Intel Corporation, Hillsboro, Oregon, USA
{satrajit.chatterjee,michael.kishinevsky}@intel.com

Abstract. Avoiding message dependent deadlocks in communication fabrics is critical for modern microarchitectures. If discovered late in the design cycle, deadlocks lead to missed project deadlines and suboptimal design decisions. One approach to avoid this problem is to get high level of confidence on an early microarchitectural model. However, formal proofs of liveness even on abstract models are hard due to large number of queues and distributed control. In this work we address liveness verification of communication fabrics described in the form of high-level microarchitectural models which use a small set of well-defined primitives. We prove that under certain realistic restrictions, deadlock freedom can be reduced to unsatisfiability of a system of Boolean equations. Using this approach, we have automatically verified liveness of several non-trivial models (derived from industrial microarchitectures), where state-of-the-art model checkers failed and pen and paper proofs were either tedious or unknown.

Keywords: liveness, deadlocks, communication fabrics, networks-on-chip, microarchitecture, high-level models, formal verification.

1 Introduction

Consider a simple system consisting of two agents and a trivial communication fabric. Each agent generates requests for the other and processes incoming requests to produce responses. Requests and responses share the same physical channel but use different virtual channels to avoid deadlock. Even on an abstract microarchitectural model of this system, automatically verifying liveness is intractable using existing model checkers. (For the curious, the precise model we consider here is shown in Figure 3 using the xMAS notation which will be explained in Section 3.) Now suppose k is the number of credits allocated to each virtual channel in the system. If we use ABC [3] to verify liveness on the abstract microarchitectural model, we find that for $k = 1$ and $k = 2$, ABC is able to verify liveness in less than a second. However, for $k = 3$ (which is not even enough to saturate the link) ABC cannot prove liveness in 18 hours. ABC checks liveness by converting it into a safety problem [4] which it then proves using

R. Jhala and D. Schmidt (Eds.): VMCAI 2011, LNCS 6538, pp. 214–231, 2011.

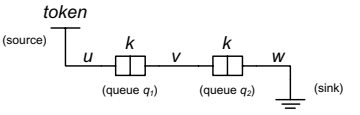

Fig. 1. A simple xMAS model with a source that generates tokens, two queues that can store k elements each and a sink. The components are connected by channels u, v and w.

interpolation. It is important to note that we add critical safety invariants [5, §5] which enable interpolation to converge. Classical LTL model checking as implemented in NuSMV [1] does worse on this example: it takes 11 hours to prove the $k = 1$ case.

The above example is probably the simplest example that is interesting from a liveness perspective. Most of our real examples involve much more complicated agents and fabrics with several different types of messages, multiple virtual channels with ordering constraints between them, deep pipelining and dozens of messages simultaneously in-flight.

In this paper we present a lightweight, *automatic* approach that allows us to prove liveness on a large class of real examples drawn from the domain of communication fabrics. Our microarchitectural models are described by instantiating and connecting components from a library of primitives. We refer to these models as xMAS networks (xMAS stands for eXecutable MicroArchitectural Specification). The properties to be verified are specified on these networks. The semantics of xMAS networks are specified using synchronous equations for each primitive. Thus every xMAS network has an associated synchronous system which we call the *synchronous model*.[1] The modeling methodology is described in more detail in [6] and its use in safety verification is described in [5]. Our definition of deadlock in xMAS models (see Section 4) is *local* i.e. it permits part of the model to be forever blocked while the rest continues processing packets. Such local deadlocks are also called livelocks.

The main idea behind our method is to exploit the high-level structure of the model in order to reason about liveness. We show that all non-live behaviors of xMAS network, or *structural deadlocks*, can be characterized by pure structural reasoning. Unreachable structural deadlocks are ruled out using safety invariants which are also obtained through automatic analysis of the model.

Our method is best explained on a simple example. Consider the system shown in Figure 1 which has a source that non-deterministically creates packets and a sink that non-deterministically consumes packets. The source and sink are connected by two queues in series. It is obvious that the system is live (i.e. activity on channels never ceases) as long as the sink is fair i.e. it always eventually consumes packets, and the source is fair i.e. it always eventually sends packets.

[1] If there is a combinational cycle in the synchronous model, the corresponding xMAS network is *ill-formed*. We do not consider such networks in the paper.

Consider a fair execution S of model M in Figure 1 and assume that in this execution channel v eventually becomes inactive (we say "stuck inactive" and express it in the paper using LTL[2] as $\mathbb{F}\,\mathbb{G}(inactive)$). If v stops transferring then q_1 must be out of tokens and unable to send or q_2 must have become filled to its maximum capacity and unable to receive. To formalize the argument, we describe "eventual stuck-at" states of channel v and queues q_1, q_2 by Boolean variables

$$\begin{aligned}\mathbf{Inactive}(v) &\equiv \text{“}v \text{ eventually stuck inactive”},\\ \mathbf{Empty}(q) &\equiv \text{“}q \text{ eventually stuck empty”},\\ \mathbf{Full}(q) &\equiv \text{“}q \text{ eventually stuck full”}.\end{aligned}$$

Note: to visually differentiate between the propositional statements capturing the instantaneous state of the system (e.g. a queue is empty now) and the temporal statements such as the eventual stuck-at properties, throughout the paper we use bold font for the temporal statements and their functions.

Then for every execution S of model M, it is true that

$$\mathbf{Inactive}(v) \Rightarrow \mathbf{Empty}(q_1) + \mathbf{Full}(q_2).$$

Queue q_1 can become empty, only if its input channel u stops sending tokens (let us denote that as $\mathbf{Idle}(u)$). Similarly, if queue q_2 is full then its output channel w must forever block, i.e. stop receiving tokens ($\mathbf{Block}(w)$). Therefore,

$$\begin{aligned}\mathbf{Empty}(q_1) &\Rightarrow \mathbf{Idle}(u),\\ \mathbf{Full}(q_2) &\Rightarrow \mathbf{Block}(w),\\ \mathbf{Inactive}(v) &\Rightarrow \mathbf{Idle}(u) + \mathbf{Block}(w).\end{aligned}$$

The consequent of last implication is obviously in contradiction with our fairness assumptions. Indeed, in fair execution source is required to periodically produce tokens and sink is required to periodically consume tokens, which is captured by

$$\mathbf{Fair} = \neg\,\mathbf{Idle}(u) \cdot \neg\,\mathbf{Block}(w).$$

Finally, we conclude that

$$\mathbf{Fair} \Rightarrow \neg\,\mathbf{Inactive}(v).$$

Also note that it is possible to perform all deductions completely automatically by forming a set of all equations and assumptions and checking it for combinational satisfiability.

The overall flow of our deadlock analysis is as follows:

– We first construct the characteristic function of all structural deadlocks in the xMAS system. This characteristic function is expressed entirely as a function of "eventually stuck at" variables similar to **Idle** and **Block** discussed above.

[2] Linear Temporal Logic. We use $\mathbb{G}$, $\mathbb{F}$, and $\mathbb{X}$ to represent "globally", "eventually", and "next" operators in LTL .

- We then derive local and global invariants of the xMAS model using an efficient automatic technique [5]. The invariants are formulated in terms of instantaneous variables (such as the current occupancies of various queues). Hence to use these invariants to prune out unreachable deadlocks we need to link stuck-at variables with instantaneous variables.
- Finally, we check the combined system of equations (structural deadlocks and model invariants) for satisfiability using an off-the-shelf SAT solver. If the problem is unsatisfiable then the xMAS system has no deadlocks. If there is a solution, one can examine it to construct a deadlock state which may be unreachable. Unreachable deadlocks may require additional invariants to rule out.

We start with a review of related work in Section 2. Next, Section 3 introduces xMAS primitives and Section 4 defines a few useful properties of xMAS models. Section 5 presents the basic structural deadlock analysis which is extended to rule out unreachable deadlocks using safety invariants in Section 6. Finally, we present experimental results for xMAS models of real microarchitectures in Section 7 and conclude with Section 8.

2 Related Work

Since the literature on detecting deadlocks in interconnect networks is large, we review only the most relevant work here. Most techniques for proving deadlock-freedom of interconnect structures are based on search and elimination of cycles in channel dependency graphs. Simple textbook static analysis techniques such as proving absence of structural cycles [7, §14.1] do not apply in our examples since there are cycles due to message and resource dependencies. Duato extends this analysis to structurally cyclic dependency graphs in which cycles can be broken by a particular choice of an adaptive routing function [8]. Recently, Taktak et al. [15] extended Duato's method to handle message dependencies in agents (e.g. responses generated as a result of requests) and Verbeek and Schmaltz formalized some of these arguments in ACL2 [16]. However, this general line of deadlock analysis critically depends on assuming a very structured network with specific assumptions on how routers and agents behave (e.g. see [15, §2.2] for a list of assumptions). Therefore these techniques cannot be directly used to analyze arbitrary xMAS models, and indeed in practice it is non-trivial to check that the assumptions used in the deadlock analysis hold on the microarchitecture. The method presented in this paper does not require any such assumptions. Finally, note that the router and agent models considered in these approaches can be captured in xMAS and the techniques in our paper can be used to argue liveness on the resultant models.

Also, closely related to the above is a rich body of *prescriptive* techniques to avoid deadlocks in various specific network-on-chip proposals ([12] is a good entry point). Clearly, these cannot be of use to reason about deadlock freedom in general microarchitectures.

There is a large body of work on analysis of systems of finite state machines connected by unbounded FIFO queues (as opposed to small FIFO queues which are the norm in hardware). In general, reachability in these systems is undecidable and most of the work seeks to identify decidable subclasses. Ghafari et al.[10] is a good entry point though note that one of the conditions for decidability there is that the communication graph must be acyclic which limits its application in our domain.

The first step of our method – generation of structural deadlocks – is somewhat reminiscent of structural deadlock analysis in Petri nets [2]. However, rather than modeling the communication fabric with colored Petri nets, which leads to a significant overhead of using explicit back-pressure arcs, colors, and complexity in modeling the data-path, and then applying a graph theoretic analysis, we derive deadlock equations directly from more compact and natural xMAS specifications by characterizing xMAS primitives using LTL statements.

An alternative approach is to create executable models of the hardware and use model checking or theorem proving techniques to argue deadlock-freedom. Although model checking is useful to find bugs, as we saw above, it cannot converge on proofs even for the simplest examples. Theorem proving techniques (e.g. [14,13,9]) require expertise in formal methods and significant human effort to find suitable decompositions for proofs or suitable refinement relationships. With this work, we are aiming to exploit the high-level structure of our models to obtain the scalability of theorem proving with the automation of model checking.

Finally, there has been interesting work recently in using termination analysis based on synthesizing ranking functions to reason about some liveness properties of multi-threaded programs [11]. However, it remains to be seen if these techniques can be usefully applied to hardware. Also note that the approach presented in this paper is fundamentally different from synthesizing ranking functions and presents an alternative way to reason about liveness based on creating simple abstractions of components for liveness analysis.

3 xMAS Primitives

xMAS model is a network of primitives connected via typed data *channels*. In this paper we assume that all types are inhabited by only finitely many values and treat types as sets of values. 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 handshake between communicating components 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 its *output* port and another component called the *target* that "reads" from the channel via its *input* port.[3] 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. We call such cycles *transfer* cycles. Note that a channel

[3] All input and output ports of a component should be connected to channels.

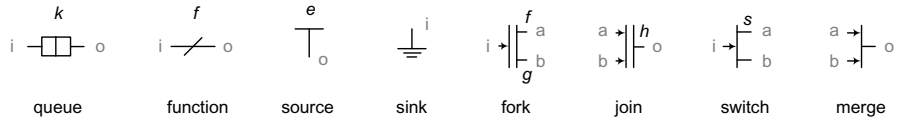

Fig. 2. A key showing the symbols for the various primitives used to model micro architectural blocks. The italicized letters (k, f, e, g, h and s) indicate parameters. Whenever we use these primitives in a diagram we need to specify values for these parameters. Often, to avoid clutter we do not show these values explicitly trusting that they are clear from the context. The gray letters (i, o, a, and b) indicate port names.

stores no state. It is represented in diagrams by a line. For example, in Figure 1, there are three channels u, v and w. For channel u, the initiator is the source and the target is the first queue.

Figure 2 shows the library of primitives. Each primitive is formally specified by its synchronous equations. For brevity we do not present these equations here but they can be found in previous papers [5,6]. Also note that even though the set of primitives is small, many other useful microarchitectural blocks such as credit logic, virtual channels, routers, out-of-order queues, scoreboards, etc. can be constructed as macro-blocks out of these primitives [6].

A *function* primitive is used to model transformations on the data. It is parameterized by a combinational function which is applied to an incoming packet to produce the outgoing packet. Note that using functions one can easily convert types of messages (e.g., from requests to responses) and hence model message dependencies both inside the fabric and in the model of the environment.

A *fork* is a primitive with one input and two outputs. 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.

A *join* is the dual of a fork. It has two inputs and one output and is parameterized by a two-input combinational function that specifies how the output packet is to be constructed from the two input packets. Like a fork, it ensures that a transfer only takes place when the inputs are ready to send and the output is ready to receive.

A *switch* is a primitive to route packets in the network. It has one input port and two output ports and is parameterized by a switching function that specifies how an incoming packet is to be routed (i.e. to which output).

Arbitration is modeled by a *merge* primitive that selects one packet among two competing packets. A merge has two 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. It is guaranteed that the requests are served fairly, i.e. that each of them will eventually be served.

A *queue* primitive is used to store packets. It is parameterized by a positive integer k that indicates its capacity. In our set of primitives, the queue is the only delay element: even if the queue is empty, an input packet is visible at the output only after 1 cycle.

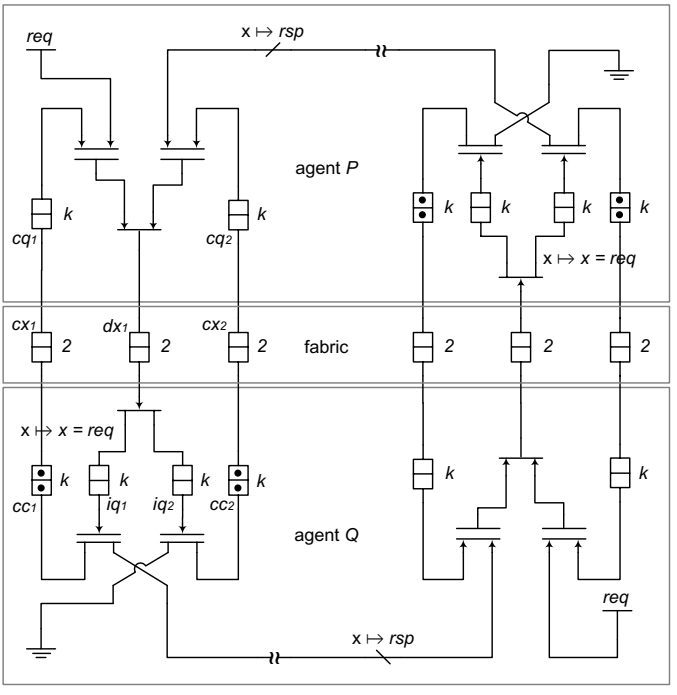

Fig. 3. Example showing a pair of agents communicating over a simple fabric (see text for details). Since each symbol has a precise formal semantics (see Section 3) this figure is a precise executable description.

Finally, *source* and *sink* primitives are used to create and consume packets non-deterministically.

Example. Figure 3 shows two agents P and Q communicating over a trivial fabric composed of six queues. Packets are modeled by an enumerated type that has two values: *req* (request) and *rsp* (response). Each agent creates new requests for the other agent. When an agent receives a request, it produces a response (by changing the packet type using a function) after a non-deterministic delay. The packet type conversion introduces a message dependence between requests and responses inside the agents. The response is sent back to the original agent where it is sunk when the sink is ready to receive it. Implementation of the non-deterministic delay macro block using xMAS primitives is shown in Figure 4(a). Thus each agent behaves like a master that produces requests and responses and a target that consumes responses and requests.

Communication between agents is done through the virtual channels. Consider agent P as example. It sends requests and responses to agent Q through the shared channel and the data transfer queue, dx_1, and then to two ingress queues iq_1 and iq_2, one per message type. An arbiter modeled by the merge primitive

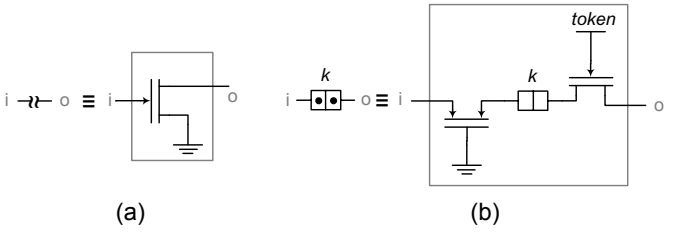

Fig. 4. Two macro-blocks used in Figure 3 and their implementation using xMAS primitives: a non-deterministic delay (a) and a credit counter (b). The functions of the forks in the two figures are identity.

selects fairly between *req* and *rsp* messages that are exposed to arbitration only if they have credit tokens inside the corresponding credit queues, cq_1 and cq_2. Credits are initialized inside the credit counters cc_1 and cc_2 to the values equal to the sizes of the ingress queues iq_1 and iq_2, i.e. to k. Implementation of the credit counter (using a queue) is shown in Figure 4 (b). Credits are returned through fabric credit queues, cx_1 and cx_2.

Due to correct sizing of credit counters, this system is free from deadlock. However if credit counters are sized incorrectly to provide more credits (say $k + 1$) than the capacity k of the ingress queues, the system deadlocks: responses can get blocked behind requests in the fabric data queues dx_1 and dx_2, while in turn requests blocking these queues cannot make forward progress to their ingress queues that are full due to over-provisioning of credits.

4 Channel Properties

Formally we define an execution of an xMAS network as an execution of the corresponding synchronous model, i.e., an infinite sequence of its consecutive states as computed by the next state functions that are composed from the equations for individual xMAS primitives. Note that synchronous models of xMAS networks are usually non-deterministic due to sources and sinks. To reason about xMAS model behavior, we use statements of LTL. For example, the LTL statement $\mathbb{G}\,\mathbb{F}(u.irdy)$ is true for an execution S if $u.irdy$ is true infinitely often in S. We denote this by:

$$S \models \mathbb{G}\,\mathbb{F}(u.irdy).$$

If some LTL formula ϕ is true for *all* executions of model M, we write $M \models \phi$ (or simply ϕ when there are no doubts about the model).

4.1 Persistency

Persistency is a desirable property of channel control signals. It significantly limits possible deadlocks and allows for simpler definitions of channel liveness.

It can be formulated as two LTL properties, one for *irdy* and the other for *trdy*. The former (*forward* persistency) guarantees that the initiator will keep sending the data until transfer occurs. The latter (*backward* persistency) states that the target will keep its input channel unblocked until transfer occurs.

$$\text{FwdPersistency}(u) \equiv \mathbb{G}((u.irdy \cdot \neg u.trdy) \Rightarrow \mathbb{X}\, u.irdy),$$
$$\text{BwdPersistency}(u) \equiv \mathbb{G}((u.trdy \cdot \neg u.irdy) \Rightarrow \mathbb{X}\, u.trdy).$$

Theorem 1. *Every channel u of any* xMAS *model M is persistent:*

$$M \models \text{FwdPersistency}(u), \qquad M \models \text{BwdPersistency}(u).$$

4.2 Life and Death of Channels

In Section 1, we illustrated a possible channel deadlock by the absence of transfers:

$$\mathbf{Inactive}(v) \equiv \mathbb{F}\,\mathbb{G}(\neg(v.irdy \cdot v.trdy)).$$

This definition is too strict for real applications. Communication fabrics are typically engineered to operate correctly even when some sources never inject packets into the network. Therefore channel liveness is defined as a "leads-to" property so that an idle channel that never wants to transfer a message is considered live:

$$\mathbf{Live}(u) \equiv \mathbb{G}(u.irdy \Rightarrow \mathbb{F}\, u.trdy). \tag{1}$$

Respectively, its negation defines channel's deadlock:

$$\mathbf{Dead}(u) \equiv \neg\, \mathbf{Live}(u) = \mathbb{F}(u.irdy \cdot \mathbb{G}\, \neg u.trdy). \tag{2}$$

Note that a channel can be live in one execution S_1, but may enter deadlock in another execution S_2. We say that there is a deadlock on channel u, if $\exists S.S \models \mathbf{Dead}(u)$.

For a persistent channel, a simpler liveness condition can be used.

Theorem 2. *If channel u is persistent then*

$$\mathbf{Live}(u) = \mathbb{F}\,\mathbb{G}(\neg u.irdy) + \mathbb{G}\,\mathbb{F}(u.trdy), \tag{3}$$
$$\mathbf{Dead}(u) = \mathbb{G}\,\mathbb{F}(u.irdy) \cdot \mathbb{F}\,\mathbb{G}(\neg u.trdy). \tag{4}$$

Since all channels of xMAS model are persistent (Theorem 1), the above theorem can be used for checking liveness of all channels.

4.3 Fairness of Sinks and Sources

If sinks of the system do not periodically consume messages the system gets into a trivial deadlock. It is therefore necessary to assume that some of system sinks are *fair*. Fairness of a sink with input channel v is defined as follows:

$$\mathbf{FairReceive}(v) \equiv \mathbb{G}\,\mathbb{F}(v.trdy). \tag{5}$$

Similarly some of the sources (such as a source initializating the credit logic in Figure 4 (b)) must be fair to avoid deadlocks. A source with output channel v is fair if:

$$\mathbf{FairSend}(v) \equiv \mathbb{G}\,\mathbb{F}(v.irdy). \tag{6}$$

A conjunction of fairness statements for *all* fair sources and sinks gives a compound fairness assumption for the model M denoted as $\mathbf{Fair}_M$.

5 Characterizing Structural Deadlocks

In this section we derive a system of Boolean equations characterizing structural deadlocks in xMAS model.

5.1 Variables

Let us first precisely define two inactive "stuck-at" states of a channel:

$$\mathbf{Idle}(u) \equiv \mathbb{F}\,\mathbb{G}(\neg u.irdy) \qquad\qquad \mathbf{Block}(u) \equiv \mathbb{F}\,\mathbb{G}(\neg u.trdy).$$

The characterization of the channel deadlock from Theorem 2 can be expressed in a slightly different form,

$$\mathbf{Dead}(u) = \neg\,\mathbb{F}\,\mathbb{G}(\neg u.irdy) \cdot \mathbb{F}\,\mathbb{G}(\neg u.trdy) = \neg\,\mathbf{Idle}(u) \cdot \mathbf{Block}(u). \tag{7}$$

Observe that **Idle** and **Block** have complete information to characterize channel deadlock. Moreover they can be propagated through xMAS primitives. Consider, for example, a fork (Figure 2) and assume that $S \models \mathbf{Block}(a)$ holds for fork's output a. Then $S \models \mathbf{Block}(i)$ also holds for the fork's input i. In fact, the following equality holds:

$$\mathbf{Block}(i) = \mathbf{Block}(a) + \mathbf{Block}(b). \tag{8}$$

Our immediate goal is to characterize all xMAS primitives with equations similar to (8). When conjuncted together for a given model M, these equations will describe all structural deadlocks in M. Before that we need to define a few more variables to capture data values on channels and internal state of queues and merges. All these variables correspond to "eventually stuck at" LTL expressions. While some of them are defined for separate data values which may appear exhaustive, recall that xMAS models are abstract microarchitectural models and therefore only few values relevant for control decisions are modeled.

Given channel $u : \alpha$ and a value x, $x \in \alpha$, we define

$$\mathbf{Idle}^x(u) \equiv \mathbb{F}\,\mathbb{G}(\neg u.irdy + (u.data \neq x)), \tag{9}$$

$$\mathbf{Block}(u) \equiv \mathbb{F}\,\mathbb{G}(\neg u.trdy). \tag{10}$$

Note that $\mathbf{Idle}^x(u)$ is a refinement of $\mathbf{Idle}(u)$. $S \models \mathbf{Idle}^x(u)$ holds during execution S if and only if the initiator of channel u eventually stops sending packets

with value x. It is straightforward to see that $\mathbf{Idle}(u) = \prod_{x\in\alpha} \mathbf{Idle}^x(u)$ for channel $u : \alpha$. We will also use a generalized notation:

$$\mathbf{Idle}^{p(x)}(u) \equiv \prod_{x\in\alpha\colon p(x)} \mathbf{Idle}^x(u),$$

where $u : \alpha$ and $p(x)$ is arbitrary predicate on α.

Now, given a merge m, define

$$\mathbf{Select}_0(m) \equiv \mathbb{F}\,\mathbb{G}(m.u = 0), \tag{11}$$
$$\mathbf{Select}_1(m) \equiv \mathbb{F}\,\mathbb{G}(m.u = 1). \tag{12}$$

Merge variables $\mathbf{Select}_0(m)$ and $\mathbf{Select}_1(m)$ capture possible "stuck at" behavior of the internal state of the merge primitive. $S \models \mathbf{Select}_i(m)$ holds in execution S if the "priority" variable $m.u$ selecting input of merge m eventually gets stuck at value i (see merge definition in [5]).

Given a queue q of size k with output channel $o : \alpha$ and a value x, $x \in \alpha$, define

$$\mathbf{Full}(q) \equiv \mathbb{F}\,\mathbb{G}(q.num = k), \tag{13}$$
$$\mathbf{Empty}(q) \equiv \mathbb{F}\,\mathbb{G}(q.num = 0), \tag{14}$$
$$\mathbf{Idle}^x(q) \equiv \mathbb{F}\,\mathbb{G}((q.num = 0) + (q.front \neq x)), \tag{15}$$

where $q.num$ is a number of packets in the queue and $q.front$ is a data value of the first packet (see the queue definition in [5]). Queue variables $\mathbf{Full}(q)$, $\mathbf{Empty}(q)$, $\mathbf{Idle}^x(q)$ are not necessary[4] but handy for dealing with queue invariants (Section 6). $\mathbf{Idle}(q)$ and $\mathbf{Idle}^{p(x)}(q)$ are defined in complete analogy with channel variables.

5.2 Deadlock Equations

Following is a list of equations characterizing all xMAS primitives. The proof is a straightforward exercise in LTL and is based on the definition of xMAS primitives. It can be automatically carried out in NuSMV. We assume that input and output channels of all primitives are persistent and named as in Figure 2.

Function. Let α be type of channel i, and β be the type of channel o. For each value $y \in \beta$,

$$\mathbf{Block}(i) = \mathbf{Block}(o),$$
$$\mathbf{Idle}^y(o) = \prod_{x\in\alpha\colon f(x)=y} \mathbf{Idle}^x(i).$$

[4] They can be expressed through "stuck-at" variables at input and output channels of the queue.

Fork. Fork is parameterized by two functions f and g. However, it can be equivalently decomposed to a fork with identity functions followed by two function primitives. We assume this simpler fork in the equations below.

Let α be a type of channel i. For each value $x \in \alpha$,

$$\begin{aligned}\mathbf{Block}(i) &= \mathbf{Block}(a) + \mathbf{Block}(b),\\ \mathbf{Idle}^x(a) &= \mathbf{Idle}^x(i) + \mathbf{Block}(b),\\ \mathbf{Idle}^x(b) &= \mathbf{Idle}^x(i) + \mathbf{Block}(a).\end{aligned}$$

Join. Writing deadlock equations for 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. 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). Given such a join with the restricted function $h : \alpha \to \gamma$, for simplifying deadlock equations we can further assume that h is an identity function and represent a more general restricted join by composing with a function primitive.

Let α be a type of channel a. For each value $x \in \alpha$,

$$\begin{aligned}\mathbf{Block}(a) &= \mathbf{Block}(o) + \mathbf{Idle}(b), \qquad \mathbf{Block}(b) = \mathbf{Block}(o) + \mathbf{Idle}(a),\\ \mathbf{Idle}^x(o) &= \mathbf{Idle}^x(a) + \mathbf{Idle}(b).\end{aligned}$$

Switch. Let α be the type of switch channels i, a, and b, and $s\colon \alpha \to \mathrm{Bool}$ be the switching function. For all values $x \in \alpha$,

$$\begin{aligned}\mathbf{Block}(i) &= \mathbf{Idle}(i) + \mathbf{Block}(a) \cdot \mathbf{Idle}^{\neg s(x)}(i) + \mathbf{Block}(b) \cdot \mathbf{Idle}^{s(x)}(i),\\ \mathbf{Idle}^x(a) &= \neg s(x) + \mathbf{Idle}^x(i),\\ \mathbf{Idle}^x(b) &= s(x) + \mathbf{Idle}^x(i).\end{aligned}$$

Merge. Let α be the type of merge channels a, b, and o. For each value $x \in \alpha$,

$$\begin{aligned}\mathbf{Block}(a) &= \mathbf{Idle}(a) + \mathbf{Select}_1(m) \cdot \mathbf{Block}(o) + \mathbf{Select}_0(m),\\ \mathbf{Block}(b) &= \mathbf{Idle}(b) + \mathbf{Select}_0(m) \cdot \mathbf{Block}(o) + \mathbf{Select}_1(m),\\ \mathbf{Idle}^x(o) &= \mathbf{Idle}^x(a) \cdot \mathbf{Idle}^x(b) + \mathbf{Idle}^x(a) \cdot \mathbf{Select}_1(m) + \mathbf{Idle}^x(b) \cdot \mathbf{Select}_0(m).\end{aligned}$$

Note that the above equations leave $\mathbf{Select}_0(m)$ and $\mathbf{Select}_1(m)$ variables unbound. Therefore they can generate a lot of "false" solutions. For example, solutions with $\mathbf{Select}_0(m) = \mathbf{Select}_1(m) = 1$ are not prohibited, however they obviously cannot be satisfied in any execution. We need additional equations to capture mutual exclusivity of $\mathbf{Select}_0(m)$ and $\mathbf{Select}_1(m)$ and other constraints characterizing the internal state of the merge primitive.

$$\begin{aligned}
\mathbf{Select}_1(m) &\Rightarrow \neg\, \mathbf{Select}_0(m),\\
\mathbf{Select}_1(m) &\Rightarrow \mathbf{Idle}(b) + \mathbf{Block}(o),\\
\mathbf{Select}_0(m) &\Rightarrow \mathbf{Idle}(a) + \mathbf{Block}(o),\\
\mathbf{Block}(o) &\Rightarrow \mathbf{Select}_0(m) + \mathbf{Select}_1(m).
\end{aligned}$$

Queue. Let α be type of channel o. For each value $x \in \alpha$,

$$\begin{aligned}
\mathbf{Block}(i) &= \mathbf{Full}(q),\\
\mathbf{Idle}^x(o) &= \mathbf{Idle}^x(q).
\end{aligned}$$

As in the case of merge, we need to provide additional constraints for the internal state of a queue to exclude false deadlocks. For each value $x \in \alpha$,

$$\begin{aligned}
\mathbf{Empty}(q) &\Rightarrow \neg\, \mathbf{Full}(q),\\
\mathbf{Full}(q) &\Rightarrow \mathbf{Block}(o),\\
\mathbf{Empty}(q) &= \mathbf{Idle}(q),\\
\mathbf{Block}(o) &\Rightarrow \mathbf{Idle}(i) + \mathbf{Full}(q),\\
\neg\, \mathbf{Block}(o) &\Rightarrow (\mathbf{Idle}^x(i) = \mathbf{Idle}^x(q)).
\end{aligned}$$

Finally, the equation that states that a blocked queue can only have one value stuck at its head. This equation is formulated for every pair of values. For every $x, y \in \alpha$, $x \neq y$,

$$\mathbf{Block}(o) \Rightarrow \mathbf{Idle}^x(q) + \mathbf{Idle}^y(q).$$

5.3 Sources and Sinks

There are no deadlock equations for sources and sinks, however some of them may have associated fairness assumptions as explained in Section 4.2. It is easy to see that fairness assumption for a source with output channel u can be rewritten as $\neg\, \mathbf{Idle}(u)$, while fairness of a sink with input v is equivalent to $\neg\, \mathbf{Block}(v)$. Hence a compound fairness assumption $\mathbf{Fair}_M$ can be formulated using only **Idle** and **Block** variables.

5.4 Checking Structural Deadlocks

Given xMAS model M with restricted joins let us denote a conjunction of the deadlock equations for all primitives of M as $\mathbf{StructDead}_M$. The main result of this section states that deadlock equations hold for all executions of M:

Theorem 3. *For every* xMAS *model* M *with restricted joins*

$$M \models \mathbf{StructDead}_M\,.$$

Assume now that we want to prove absence of deadlock on channel u of model M under certain fairness assumptions $\mathbf{Fair}_M$. To do so it is sufficient to prove unsatisfiability of $\mathbf{StructDead}_M \cdot \mathbf{Fair}_M \cdot \mathbf{Dead}(u)$, where $\mathbf{Dead}(u)$ is defined by (7). If this formula is unsatisfiable then model M has no execution, which simultaneously satisfies $\mathbf{Fair}_M$ and $\mathbf{Dead}(u)$. Hence channel u is live in every fair execution of M regardless of the initial state of the system. The example of Figure 1 can be proven deadlock-free in this way.

6 Adding Reachability Constraints

To rule out unreachable solutions and enable liveness proofs for practical examples, $\mathbf{Fair}_M$, $\mathbf{StructDead}_M$ and $\mathbf{Dead}(u)$ should be augmented with the additional model invariants, i.e. equations which hold for all *reachable* states of M, and do not hold for some of the unreachable states. We will refer to conjunction of all additional invariants as $\mathbf{Inv}_M$. For fully automatic proofs, $\mathbf{Inv}_M$ can be generated using local invariants and global *flow invariants* from [5]. A flow invariant is a linear equation relating occupancies of different queues in xMAS network. It holds in every reachable state. For example, an xMAS model in Figure 5 has the following flow invariant

$$q_1.num + q_2.num = q_3.num. \tag{16}$$

Although this xMAS model has no reachable deadlocks, $\mathbf{Fair}_M \cdot \mathbf{StructDead}_M \cdot \mathbf{Dead}(i)$ has 2 satisfying solutions: $\mathbf{Empty}(q_1) = \mathbf{Empty}(q_2) = \mathbf{Full}(q_3) = 1$ and $\mathbf{Full}(q_1) = \mathbf{Full}(q_2) = \mathbf{Empty}(q_3) = 1$. Hence Theorem 3 is not sufficient to prove deadlock-freedom. On the other hand, it is easy to see that (16) is in contradiction with both of the above assignments. Indeed, as follows from (16) in any reachable eventual stuck-at state it is impossible to have both q_1 and q_2 empty and the q_3 full, as well as it is impossible to have both top queues full and the bottom queue empty. Note that in the above reasoning we used the instantaneous $q.num$ variables to reason about LTL "stuck-at" variables **Full** and **Empty**. The translation is straightforward for this particular example, but let us illustrate the general approach.

For every queue q of size k let us introduce an auxiliary integer variable $\mathbf{N}(q)$ that characterizes possible occupancies of queue q that occur infinitely often in a given execution S,

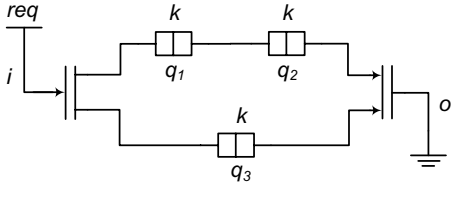

Fig. 5. Example of xMAS model requiring flow invariant $q_1.num + q_2.num = q_3.num$ to prove liveness. All channels are of unit type with single *req* value.

$$\mathbf{N}(q) \in \{n\colon \mathbb{G}\,\mathbb{F}(q.num = n)\}. \tag{17}$$

Conceptually, $\mathbf{N}(q)$ is similar to "stuck-at" variables (9-15) with only difference that constraint (17) allows for $\mathbf{N}(q)$ to non-deterministically assume any value from its execution-dependent domain. Following equations relate $\mathbf{N}(q)$ to $\mathbf{Empty}(q)$, $\mathbf{Full}(q)$ and $\mathbf{Block}(o)$ variables, where o is an output channel of q

$$0 \le \mathbf{N}(q) \le k, \tag{18}$$
$$\mathbf{Empty}(q) \Rightarrow \mathbf{N}(q) = 0, \tag{19}$$
$$\mathbf{Full}(q) \Rightarrow \mathbf{N}(q) = k, \tag{20}$$
$$\mathbf{Block}(o) \cdot \neg\, \mathbf{Empty}(q) \Rightarrow \mathbf{N}(q) > 0, \tag{21}$$
$$\mathbf{Block}(o) \cdot \neg\, \mathbf{Full}(q) \;\; \Rightarrow \mathbf{N}(q) < k. \tag{22}$$

On one hand constraints (18-22) relate "stuck-at" variables with $\mathbf{N}(q)$, while on the other hand, $\mathbf{N}(q)$ variables can be used to reformulate any flow invariant.

Theorem 4. *Given an* xMAS *model M, consider a predicate ψ on the queue occupancies. If*

$$M \models \mathbb{G}\,(\psi(q_1.num, \ldots, q_r.num)) \tag{23}$$

then for every execution S of M, $\exists\, \mathbf{N}(q_1), \ldots, \mathbf{N}(q_r)$ satisfying (17) such that

$$S \models \psi(\mathbf{N}(q_1), \ldots, \mathbf{N}(q_r)). \tag{24}$$

Using Theorem 4, we can now construct $\mathbf{Inv}_M$ from equations (18-22) and (24) capturing reachability constraints provided by original invariant (23). Note that predicate ψ in (23) can be arbitrarily complex. In particular, ψ can be defined as conjunction of all invariants we want to capture.

For the network in Figure 5, equations (18-22) provide:

$$\mathbf{Empty}(q_i) \Rightarrow \mathbf{N}(q_i) = 0, \qquad i = 1, 2, 3, \tag{25}$$
$$\mathbf{Full}(q_i) \Rightarrow \mathbf{N}(q_i) = k, \qquad i = 1, 2, 3. \tag{26}$$

Applying Theorem 4 with the predicate $\psi(q_1.num, q_2.num, q_3.num) = (q_1.num + q_2.num = q_3.num)$ gives

$$\mathbf{N}(q_1) + \mathbf{N}(q_2) = \mathbf{N}(q_3). \tag{27}$$

Now it is easy to see that $\mathbf{Empty}(q_1) = \mathbf{Empty}(q_2) = \mathbf{Full}(q_3) = 1$ and (25-27) lead to $0 + 0 = k$, while $\mathbf{Full}(q_1) = \mathbf{Full}(q_2) = \mathbf{Empty}(q_3) = 1$ and (25-27) lead to $k + k = 0$. Both statements are false for any positive value of k. Therefore, $\mathbf{Fair}_M$, $\mathbf{StructDead}_M$, $\mathbf{Inv}_M$ and $\mathbf{Dead}(i)$ cannot be simultaneously satisfied by any execution. Hence channel i is live in every fair execution, which also satisfies flow invariant (16).

Approach we described can be generalized to handle wider class of flow invariants from [5], which count tokens satisfying some properties. Consider for example an invariant such as

$$q_1.num^{p_1} + q_2.num^{p_2} = q_3.num^{p_3},$$

where $q.num^p$ is number of tokens in queue q satisfying predicate $p(x)$. It can be translated to

$$\mathbf{N}(q_1)^{p_1} + \mathbf{N}(q_2)^{p_2} = \mathbf{N}(q_3)^{p_3},$$

where $\mathbf{N}(q)^p$ is an integer variable with LTL constraint similar to (17) i.e.

$$\mathbf{N}(q)^p \in \{n\colon \mathbb{G}\,\mathbb{F}(q.num^p = n)\}.$$

Variables $\mathbf{N}(q)^p$ can be characterized by equations similar to (18-22), and generalization of Theorem 4 is straightforward.

7 Results

We have applied techniques described in this paper to prove deadlock-freedom of a number of xMAS models, including those derived from industrial microarchitectures. Drawn from the domain of communication fabrics, they are characterized by deeply pipelined logic for multi-phase transactions, presence of ordering logic and several virtual channels, and peer-to-peer traffic.

We compared our approach for liveness verification with state-of-the-art model checkers ABC version 91206p (using conversion to safety [4]) and NuSMV version 2.4. We only show comparisons with ABC since ABC outperformed NuSMV on all examples that we looked at. Note that we add safety invariants corresponding to global flow invariants [5, §5] which ABC retains after the liveness to safety transformation as additional safety properties on the transformed model. In our experience this dramatically improves the convergence of interpolation on the transformed model.

Table 1. Experimental results using ABC

Example	i	r	n	depth	time
Fig. 3 ($k = 1$)	6	28	60	(11, 10)	1.06
Fig. 3 ($k = 2$)	6	37	128	(12, 13)	2.63
Fig. 3 ($k = 3$)	6	38	211	BMC 34	-
SMF1	4	83	455	BMC 31	-
SMF2	4	137	1200	BMC 17	-

Table 1 shows the results of running ABC on 5 benchmarks. The first 3 are derived from Figure 3 with different values of the parameter k. The next two examples SMF1 and SMF2 are industrial examples of messaging fabrics where the microarchitects wanted to ensure that no deadlocks are introduced by allowing an agent to send packets to itself using the messaging fabric. SMF1 allows complete reordering of messages whereas SMF2 imposes some ordering restrictions on packets to guarantee producer-consumer ordering. SMF1 and SMF2 are parameteric models and we set the parameter values to obtain minimal configurations for comparing with ABC.

Table 2. Experimental results using the proposed method. All instances are proved UNSAT in less than 1 second.

Example	c	q	v	e	f
Figure 3	54	12	176	241	4
SMF1	57	20	182	257	4
SMF2	71	24	232	322	6

Column i in the table is the number of primary inputs (oracles); r and n are number of registers and AIG nodes (after lightweight synthesis but before liveness to safety conversion). 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 12 hours indicated by a dash (and we show the final BMC depth in the previous column). Note that we are able to obtain proofs only for the first two benchmarks using ABC.

Table 2 shows the results of applying the method proposed in this paper to the 5 examples. Columns c and q refer to the number of channels and queues in the model. Columns v and e refer to the number of variables and equations constructed for the satisfiability problem. Finally column f indicates the number of flow invariants added to rule out unreachable deadlocks. These flow invariants are generated automatically according to the algorithm in [5, §5] (which takes negligible run-time). Note that Table 2 has only 1 row for Figure 3. This is because the formulation of the structural deadlock problem and the flow invariants do not depend on the size of the queues and hence is independent of k. Finally, we note that in all cases, the resulting satisfiability problem was solved by Minisat in less than 1 second.

8 Conclusion

The method presented in this paper allows us to obtain proofs of deadlock freedom on real examples which were previously intractable. Although the comparisons with ABC have been done on minimal configurations, our method is far more scalable than brute-force model checking. For instance, our method has no trouble accommodating much larger configurations.

Finally, although our method is sound, it is not complete since false deadlocks may be found. Although in all our practical examples so far, flow invariants have sufficed to rule out all false deadlocks, it is possible that in some cases they may not be adequate. However, if liveness proof fails, our method always provides a concrete deadlock scenario: By looking at the positive literals of a satisfying solution, one can re-construct the deadlock state. This counter-example can be analyzed manually or be presented as *safety* problem for a model checker to rule out.

Acknowledgment. We thank Sayak Ray for implementing the liveness to safety conversion in ABC and for careful reading of an earlier draft.

References

1. NuSMV home page, `http://nusmv.irst.itc.it/`
2. Barkaoui, K., Dutheillet, C., Haddad, S.: An efficient algorithm for finding structural deadlocks in colored Petri nets. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 69–88. Springer, Heidelberg (1993)
3. Berkeley Logic Synthesis and Verification Group: ABC: A system for sequential synthesis and verification, `http://www.eecs.berkeley.edu/~alanmi/abc/`
4. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science 66(2), 160–177 (2002)
5. Chatterjee, S., Kishinevsky, M.: Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 321–338. Springer, Heidelberg (2010)
6. Chatterjee, S., Kishinevsky, M., Ogras, U.Y.: Quick formal modeling of communication fabrics to enable verification. In: Proc. IEEE High Level Design Validation and Test Workshop (HLDVT), pp. 42–49 (2010)
7. Dally, W., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann, San Francisco (2004)
8. Duato, J.: A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. IEEE Trans. Paral. Distrib. Syst. 6(10), 1055–1067 (1995)
9. Gebremichael, B., Vaandrager, F.W., Zhang, M., Goossens, K.G.W., Rijpkema, E., Rădulescu, A.: Deadlock prevention in the æthereal protocol. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 345–348. Springer, Heidelberg (2005)
10. Ghafari, N., Gurfinkel, A., Klarlund, N., Trefler, R.J.: Algorithmic analysis of piecewise fifo systems. In: FMCAD, pp. 45–52 (2007)
11. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: POPL 2009: Proc. of Symp. on Principles of Prog. Lang., pp. 16–28. ACM, New York (2009)
12. Hansson, A., Goossens, K., Rădulescu, A.: Avoiding message-dependent deadlock in network-based systems on chip. VLSI Design 2007 (May 2007)
13. Manolios, P., Srinivasan, S.K.: Automatic verification of safety and liveness for pipelined machines using web refinement. ACM Trans. Des. Autom. Electron. Syst. 13(3), 1–19 (2008)
14. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 342–345. Springer, Heidelberg (1999)
15. Taktak, S., Desbarbieux, J.L., Encrenaz, E.: A tool for automatic detection of deadlock in wormhole networks on chip. ACM Trans. Design Autom. Electr. Syst. 13(1) (1995)
16. Verbeek, F., Schmaltz, J.: Formal specification of networks-on-chips: deadlock and evacuation. In: DATE 2010, pp. 1701–1706 (2010)