Quick Formal Modeling of Communication Fabrics to Enable Verification

Satrajit Chatterjee, Michael Kishinevsky and Umit Y. Ogras, Intel Corporation

***Abstract*—Although communication fabrics at the microarchitectural level are mainly composed of standard primitives such as queues and arbiters, to get an executable model one has to connect these primitives with glue logic to complete the description. In this paper we identify a richer set of microarchitectural primitives that allows us to describe complete systems by composition alone. This enables us to build models faster (since models are now simply wiring diagrams at an appropriate level of abstraction) and to avoid common modeling errors such as inadvertent loss of data due to incorrect timing assumptions. Our models are formal and they are used for model checking as well as dynamic validation and performance modeling. However, unlike other formalisms this approach leads to a precise yet intuitive graphical notation for microarchitecture that captures timing and functionality in sufficient detail to be useful for reasoning about correctness and for communicating microarchitectural ideas to RTL and circuit designers and validators.**

I. Introduction

Typically, when writing a microarchitectural model (be it in a simulation language such as Verilog or SystemC or a formal modeling language such as Esterel [2], Murphi [8], TLA [10], or SMV[18]), one uses the full power of the modeling language to describe the system. On the other hand, most communication microarchitectural models are mainly composed of standard blocks such as queues and arbiters. When writing a model these standard components are usually instantiated from a library, but in order to complete the model one has to write a large amount of "glue" logic in the underlying language to connect these components.

The question we ask in this paper is: Do we need this ad hoc glue logic? Instead can we identify a set of primitives that is rich enough to permit a purely structural description of interesting systems?

We answer the last question in the positive by identifying a small set of primitives that have proven useful in modeling a number of examples drawn from the domain of communication fabrics. In practice, we have been able to use these primitives to capture the salient microarchitectural details of several large blocks including a memory switch targeted at SoC products and a distributed tag directory for a multi-core graphics processor.

We see three advantages of this approach. First, by having a small set of well understood primitives, it becomes easier to build correct models. An analogy might be to structured programming. Even though it is possible to write programs with "goto" statements, a great advancement in programming was to restrict the use of goto. By using a standard set of control structures such as "while" loops and "for" loops (which capture standard patterns of goto usage), a class of programming errors is eliminated. Similarly, by constructing our models from standard building blocks we hope to eliminate certain classes of errors such as inadvertent dropping or overwriting of data.

Second, since the model is built by purely structural composition, it is much faster to write as the description is really a wiring diagram. One does not have to think about the subtle control and timing logic in these highly concurrent systems. It is also easier to change the model as the microarchitecture evolves since it is closer to the level of abstraction at which the microarchitects think.

Third, since our models are built from a structural composition of these primitive blocks, it is natural to represent these models diagrammatically. This leads to a schematic view of the microarchitecture (similar to schematics for circuits). Unlike the normal box diagrams seen in microarchitectural documents which have no semantics, these diagrams have precise semantics that capture both timing and functionality. In addition to being a formal description of the system, we have found these diagrams to be very useful for communication (amongst microarchitects and between microarchitects and RTL writers) and for reasoning about correctness.

We call these models xMAS for eXecutable Microarchitectural Specification. Given the close correspondence of the diagrams and the models we use the terms xMAS model and xMAS diagram interchangeably.

The proposed approach results in formal models that are used for model checking as well as dynamic validation and performance modeling. To address the verification problem, we generate abstract Verilog from xMAS models and use in-house and academic verification tools for formal bug-hunting using bounded model checking. In addition we can also automatically generate inductive invariants of the systems that are comprised from local invariants of our well defined primitives and global invariants (such as preserving the number of credits in virtual channels) that are extracted through formal flow analysis of the system. Generating these invariants allows us to do quick formal proofs of non-trivial system properties.

In the following, we start by looking at simple cases where we can eliminate the glue logic and build a model by simply connecting some well chosen primitives and demonstrating how by having standard interfaces to these primitives it is possible to built common microarchitectural idioms by structural composition alone. Using a series of examples, we show how the primitives can be composed to achieve progressively harder microarchitectural tasks such as credits, virtual channels, ordering logic, etc. We then discuss some results of modeling realistic industrial examples: I/O switching fabrics, distributed tag directory, coherent interconnect based

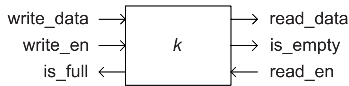

Fig. 1. A synchronous FIFO queue with a write port (on the left) and a read port (on the right). The queue can store k data elements. In each clock cycle, if the queue is not full, a new element may be inserted; and if the queue is not empty, the oldest element may be removed. read_data exposes the oldest element if the queue is not empty. There is no bypass: if the queue is empty, the incoming data appears at the output one cycle later.

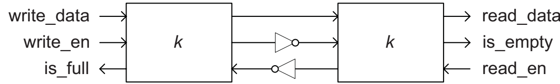

Fig. 2. Composing two queues requires additional glue logic.

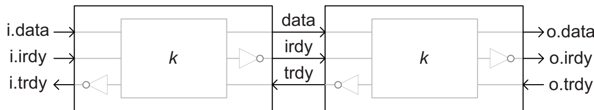

Fig. 3. By choosing an appropriate interface for the queue, it is possible to compose two queues without additional glue logic.

on rings, and an SOC northcomplex.

Timing model. We use a synchronous model of time [1]. Each primitive is defined by a set of equations that can be trivially translated into synthesizable Verilog with a single clock and edge-triggered flipflops. Giving asynchronous semantics to our primitives is equally possible, but is out of scope of this paper.

II. A Library of Primitives

A. Basic Communication

Consider a synchronous FIFO queue with a standard interface comprising a read port and a write port as shown in Figure 1. The queue has two parameters: size k (the number of elements it can contain) and a type τ (the type of elements it can contain). To compose two instances of such a queue back-to-back, we need some glue logic as shown in Figure 2 to ensure that data elements are transferred "correctly" from queue A to queue B i.e. a transfer happens if and only if queue A is not empty and queue B is not full.

Looking at Figure 2 it may be obvious how to redefine the queue interface of Figure 1 so that no additional glue logic is necessary. For instance, consider the following interface definition in terms of a read port o (output of the queue) and a write port i (input to the queue):

o.data := read_data	write_data := i.data
o.irdy := **not** is_empty	write_en := i.irdy
i.trdy := **not** is_full	read_en := o.trdy

where we define the new interface in terms of the old one by folding the inverters in the glue logic of Figure 2 into the queue itself.

With this new interface, one can directly connect two queues structurally without requiring any additional glue logic as shown in Figure 3. This is the central idea of this paper: by defining a library of elements that adhere to a standard interface, we wish to describe interesting microarchitectures by simple structural connections.

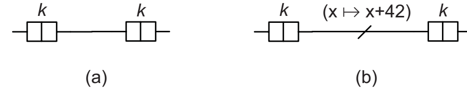

Fig. 4. (a) We can abbreviate Figure 3 by using a single line to represent the channel connecting the two queues (instead of three separate lines for the data, irdy and trdy wires of the channel). (b) A function primitive (Section II-B) can be used to transform data as it flows through. Using this pattern it is possible to easily model linear pipelines.

Note that the two queues are connected by three wires: data, irdy (for initiator ready) and trdy (for target ready). We call this triple of wires a *channel*. Channels are the only communication mechanism in our framework. As in the queue example, a channel always connects two ports: an *initiator* and a *target*. The data and irdy wires go from the initiator to the target, whereas the trdy wire goes from target to initiator. irdy indicates that the initiator is ready to send data, and trdy indicates that the target is ready to accept data. Data are transferred exactly on those clock cycles when both irdy and trdy are true. Each channel has a type τ which indicates the type of data it carries. Channels induce types on the ports they connect to. For example, both i and o ports of a queue have type τ and this is denoted by $i, o : \tau$.

Diagrammatically, rather than a line for each of the three wires in a channel, we show only a single line (for a channel). Thus the Figure 3 can be abbreviated to Figure 4(a). Note that with the queue interfaces as described above and given a queue implementation, Figure 4(a) is a very precise description of a system that specifies both functionality and timing. In particular this description is precise enough for model checking or generating synthesizable Verilog.

B. Specifying Functionality

Suppose we wish to modify the previous example so that as a packet moves from A to B, it gets incremented by 42 (assuming that A and B store say 32-bit integers). In a normal model, the glue logic would be modified to handle the increment, but how do we describe this new system structurally?

We solve this with a new primitive called *function* to model transformations on the data. It is parameterized by two types α and β and a function $f : \alpha \to \beta$ (just as a queue is parameterized by a single type τ and a number k). It has an input port $i : \alpha$ and an output port $o : \beta$ and is fully characterized by the following (combinational) equations:[1]

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

Figure 4(b) shows the new system diagrammatically where the function primitive is shown with a short slanting line. It should be clear that Figure 4(b) is as precise a description of the new system as Figure 4(a) was for the previous one.

So what is the advantage of this? Compared to the usual style of modeling, we get a clean separation of the transformation applied to data and the logic co-ordinating the

[1] 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.

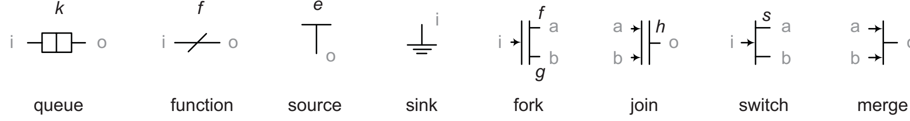

Fig. 5. A key showing the symbols for the various primitives used to model microarchitectural blocks. Section II describes these components in detail. The italicized letters (k, f, e, g, h and s) indicate parameters. Whenever we use these primitives in a diagram we need to specify values for these parameters. 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 II. Observe that for some components such as the fork, we place the parameter close to the "corresponding" port in the diagram.

movement of data. Furthermore, we get a precise diagrammatic representation of the system.

C. Sources and Sinks

Our examples so far have been fragmentary. To build closed systems we need new primitives for modeling the creation and destruction of packets. 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:

o.irdy := oracle **or pre**(o.irdy **and not** o.trdy)
o.data := e

where **pre** is the standard synchronous operator that returns the value of its (boolean) argument in the previous cycle and the value 0 in the very first cycle [1]; and 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.

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

i.trdy := oracle **or pre**(i.trdy **and not** i.irdy)

For model checking liveness properties, it is necessary to put fairness constraints on the oracles of sinks to rule out those traces where the oracle stays constant zero. In Linear Temporal Logic (LTL) this would be (**GF** oracle).[3] Also, occasionally it may be necessary to put a similar fairness constraint on a source (e.g. see Section III-B). We call such sources *fair*.

Sometimes it is convenient to have sources and sinks that are always ready i.e. o.irdy = 1 (for a source) and i.trdy = 1 (for a sink). We call these *eager*. Likewise, it is useful to have sinks that are never ready i.e. i.trdy = 0. We call these *dead*. However, even for other sources and sinks, it is natural to associate rates to control how often they attempt to inject or consume packets. Furthermore, these rates translate trivially into probabilities that the oracle associated with a source or a sink is 1 in any given cycle. This permits one to automatically generate a random test-bench to drive the oracles in simulation. We have found it very convenient to generate such random test-benches when generating Verilog from our models. They help with quick sanity checks and enable quick performance validation. (Rates are not used for model checking.)

Finally, one can imagine more complex sources that emit arbitrary values from a given set. However, for the rest of this paper it will suffice to consider only previously defined simple sources which emit constant values.

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

[3] **G** stands for "globally" (i.e. always) and **F** for "in future" (i.e. eventually).

D. Synchronization

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

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)

Note the duality of the join equations with the fork equations for the irdy and trdy signals.

E. Switching

A *switch* is a primitive to route packets in the network. It consists of one input port i and two output ports a and b, all of type α. It is parameterized by a switching function $s : \alpha \to \text{Bool}$. 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)

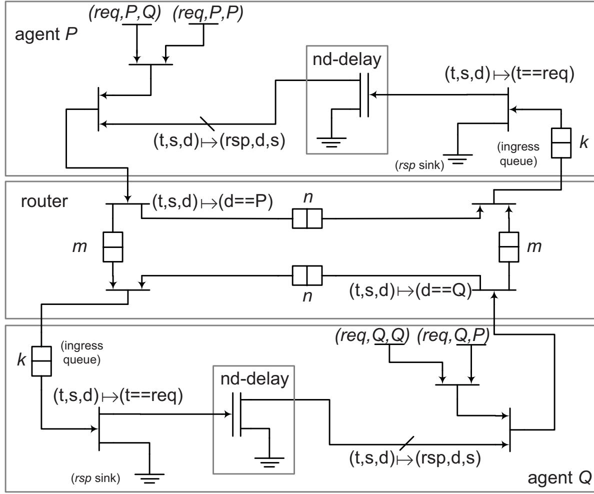

Fig. 6. Example showing a pair of agents communicating over a simple fabric. The nd-delay box models non-deterministic delay (the functions of the fork are identity). See Section III-A. Since each symbol has a precise formal semantics (see Section II) this figure is a precise executable description.

F. Arbitration

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 $o : \alpha$.

$$
\begin{array}{lll}
\text{o.irdy} & := & \text{a.irdy } \textbf{or} \text{ b.irdy} \\
\text{o.data} & := & \text{a.data} \quad \textbf{if } u \textbf{ and } \text{a.irdy} \\
 & & \text{b.data} \quad \textbf{if not } u \textbf{ and } \text{b.irdy} \\
\text{a.trdy} & := & u \textbf{ and } \text{o.trdy } \textbf{and} \text{ a.irdy} \\
\text{b.trdy} & := & \textbf{not } u \textbf{ and } \text{o.trdy } \textbf{and} \text{ b.irdy}
\end{array}
$$

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

$$
\begin{array}{llll}
u & := & 1 & \textbf{if } \text{a.irdy } \textbf{and not } \text{b.irdy} \\
 & & 0 & \textbf{if not } \text{a.irdy } \textbf{and } \text{b.irdy} \\
 & & \textbf{not pre}(u) & \textbf{if pre}(\text{o.irdy } \textbf{and } \text{o.trdy}) \\
 & & \textbf{pre}(u) & \textbf{otherwise}
\end{array}
$$

or we could leave the fairness logic abstract replacing it with non-determinism (for checking safety) or fairness constraints (for checking liveness).

III. Examples

A. A Simple Fabric Connecting Two Master-Target Agents

Figure 6 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 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.

Discussion. The xMAS diagram makes explicit the dependencies in the system thus making it easier to reason about deadlocks. For example, in this system, if both agents create requests for a while without sinking responses, then all the queues in the system may fill up with requests leading to deadlock. Indeed, there is a trace that leads to deadlock in $k+\min(m, n)$ cycles.

By using virtual channels instead of simple queues, this deadlock can be avoided. In the next two sections we build up the machinery for virtual channels, but we illustrate using simpler systems where one agent is master and the other target and the two are directly connected.

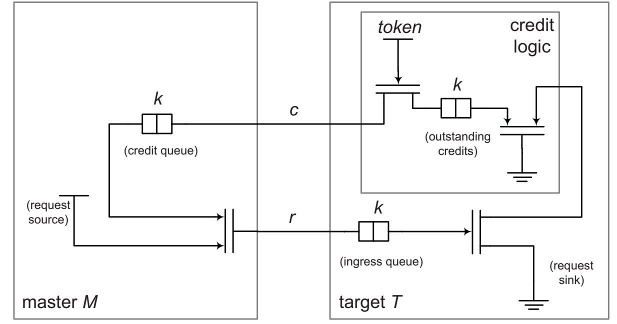

Fig. 7. Example showing how credits can be modeled with the primitives of Section II. In the credit logic, the source is fair and the sink is eager. See Section III-B.

B. Credit Logic

Credits are a common microarchitectural design pattern for flow control and resource allocation in distributed systems [7]. Figure 7 shows a simple example with credits where a master M communicates with a target T using a channel c (for credits) and a channel r (for requests).

In this system, credits are modeled by the unit type i.e. an enumerated type with only one possible value which we shall call *token*. The box labeled “credit logic” ensures that at most k outstanding tokens are sent to M at any given time. The queue in the credit logic contains as many tokens as there are outstanding credits issued to M and applies backpressure to the token source when it is full. The join in M ensures that for M to send data to T along r, its credit queue must have a token. Therefore, when M is ready to send a packet on r, it is *guaranteed* that the ingress queue in T has enough room (indeed this is the purpose of the credit mechanism). Thus we have the following LTL assertion: **G** (r.irdy $\implies$ r.trdy). Since this particular type of assertion is very common in our models, we call channels that satisfy such an assertion *non-blocking* channels. When the request sink in T is ready to consume the request, the join in the credit logic ensures that a corresponding token is removed from the outstanding credits queue providing room for a new credit token. The sink in the credit logic is eager and applies no back-pressure.

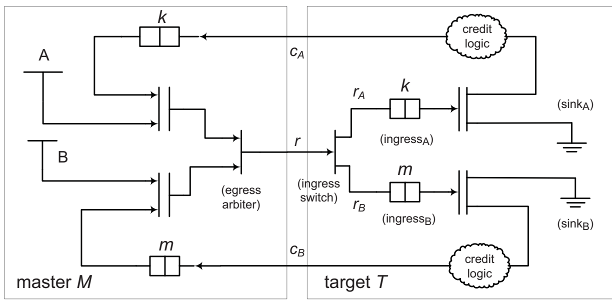

Fig. 8. Example showing how virtual channels can be modeled with the primitives of Section II. The credit logic bubbles encapsulate the logic shown in the credit logic box of Figure 7. The switch in the target routes A packets to r_A and B packets to r_B. See Section III-C for details.

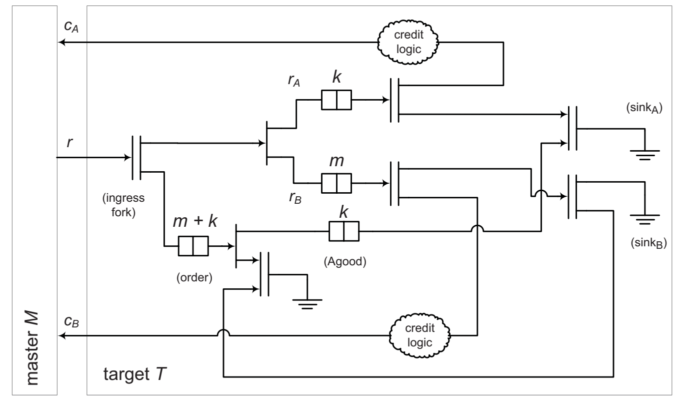

Fig. 9. Example that shows how bypassing and ordering logic can be modeled using the primitives of Section II. The master M is the same as in Figure 8. The switch at the output of the order queue sends A to Agood and B to the join. See Section III-D for details.

Discussion. How does such a model compare with one written directly in a language such as Verilog? First, typically one would have a valid wire for r (corresponding to our irdy), but would not have the trdy (since there is no back-pressure from the ingress queue of T). However, by explicitly modeling the trdy wire, it becomes easy to add assertions to validate our microarchitectural timing assumptions. Furthermore, given such an assertion we can automatically optimize away the trdy wire of r and simultaneously introduce an assertion (in the manner of speculative reduction [19]).

Second, it may appear odd and wasteful to use queues to model counters to track credit. It is not wasteful: one can trivially detect queues whose type is the unit type and optimize them to counters.[4] However, by modeling credits in this manner, it is easy to ensure that credits don't get "lost." Other useful properties can be checked as well, for instance that channel c is non-blocking.

Third, in this style it is natural to write non-deterministic models. Once a credit is returned, a new credit is issued non-deterministically by the credit logic.

C. Virtual Channels

Virtual channels 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 [7]. Figure 8 shows a simple example where a master M sends two types of messages A and B (think of these as perhaps requests and responses of Section III-A) 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 7 is used to ensure that 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.

Discussion. It is clear from the diagram that for r to be non-blocking, r_A and r_B must also be non-blocking. Indeed if a channel is non-blocking, any other channel that it feeds into "combinationally", i.e. without going through a queue, must also be non-blocking. Thus from xMAS diagrams it is possible to reason about the propagation of microarchitectural timing constraints (and even channel utilizations). Furthermore, by using non-blocking assertions on important interface channels, it becomes easier to correctly attribute blame in the case of a design bug. Aside: if the output of a queue is non-blocking, then the queue can be simplified to a register since it contains at most one element at any time. For this reason we do not need a special flop-like primitive to model delays.

Note the advantage of modeling credits as packets in their own right. Suppose, instead of having two credit channels, we wanted to merge the credits onto a single channel, we could do this easily by using an enumerated type (instead of the unit type) to model the two different kinds of credits. We then introduce a merge in T and a switch in M. Also, from the diagram, it is easy to see that this merging of the credits would not lead to a throughput loss since the flow of credits would match the flow of data from M to T. This kind of reasoning and modification is much harder with a conventional model.

D. Bypassing and Ordering

In Figure 8, A and B messages can be sunk independently of each other at the target. Now suppose we have an ordering restriction that requires that an A can be sunk only if all Bs that came before it on r (r is called an *ordering point*) have been sunk. Such one-way[5] ordering restrictions are often needed to implement cache coherence protocols or to guarantee producer-consumer ordering for software yet avoiding deadlock as would be caused by total ordering.

Figure 9 shows how this may be implemented using our primitives. For reasons of space, we cannot fully describe this example, but we hope that by now the reader has gained some fluency in reading the diagrams so that further explanation would be unnecessary! We remark only that although the size of A and B messages could be large (a tuple of command and 32B data perhaps), the order queue simply needs to store whether a message is of type A or of type B. We assume that the fork at the ingress of T does such a projection.

[4] Indeed a channel whose type is unit does not even need the data wire.

[5] Note that this is different from total ordering: the rule allows a B to be sunk even if an earlier A is not sunk.

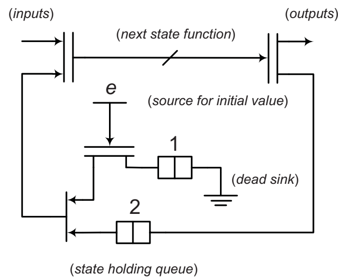

Fig. 10. A fragment of an XMAS diagram showing how a finite state machine may be modeled using the primitives of Section II. See Section III-E for details.

E. Finite State Machines

Any finite state machine (FSM) in the microarchitecture can be easily modeled using the primitives of Section II. Figure 10 shows an example of an FSM that receives an input packet, and computes the next state and output packet using the current state and the input packet. A queue of size 2 is used to hold the current state of the FSM.[6] The initial state e however is provided from an eager source. Since the source is connected to a dead sink by means of a queue of size 1, it only produces 1 token. Thus the merge in the digram is really a mutex, i.e. only one of its inputs has a packet at any given time. A join is used to combine the input and current state into a pair which is then transformed into a pair of next state and output. Finally a fork is used to separate the outputs from the next state.

Discussion. The ability to model arbitrary FSMs provides an indication of the theoretical modeling power of the set of primitives presented in Section II. Indeed, an arbitrary state transition system may be modeled using these primitives by describing it as an FSM and modeling it in the manner of Figure 10. However, in that case most of the internal structure of the system is lost, and the model begins to resemble a more conventional model written in Verilog, etc. Nevertheless, this pattern is useful to model natural FSMs in the microarchitecture such as those for cache coherence. Furthermore using this pattern it is easy to build models of memories and CAMs (content addressable memories).

[6]The queue only stores at most one element, but we need a queue of size 2 to ensure that we can read and write in the same cycle.

F. Scoreboards

Figure 11 shows how a two-entry scoreboard may be modeled using the primitives of Section II. An incoming transaction on the left needs to obtain a tag before it can enter the scoreboard. Different tags are used to distinguish different in-flight transactions in the scoreboard. In this example, the scoreboard supports two simultaneous in-flight transactions and hence there are two tags: tagA and tagB. Once the transaction enters the scoreboard it competes with the other transaction (if there is one) to enter the first phase of processing. The results of this phase may return out of order: tags are used to match a result with the corresponding transaction in the scoreboard. Once the result of the first phase is returned, the transaction moves on to the second phase. After the second phase is done, the transaction becomes eligible for retirement. When it wins arbitration, it retires and releases its tag which is then recycled for use by a future transaction.

Discussion. This pattern may be used to model complicated control logic in a natural transactional style. For example, using this pattern one can model, say, a multi-processor memory controller that issues snoops and collects replies (in phase 1) and updates memory (in phase 2).

It is easy to extend this example to track several simultaneous transactions in flight by increasing the set of tags, and to support transactions with multiple phases with complicated dependencies between them. Finally, note that even a big architectural change such as in-order retirement can be incorporated with little effort by adding a fork, a queue (to remember the orginal order) and a join per tag in the retirement phase.

IV. PRACTICAL EXPERIENCE

A. Tool Chain

We have built a system that provides a C++ API to describe these diagrams. Most of the modeling effort is in sketching the diagram and entering it. Modeling bugs are quickly found

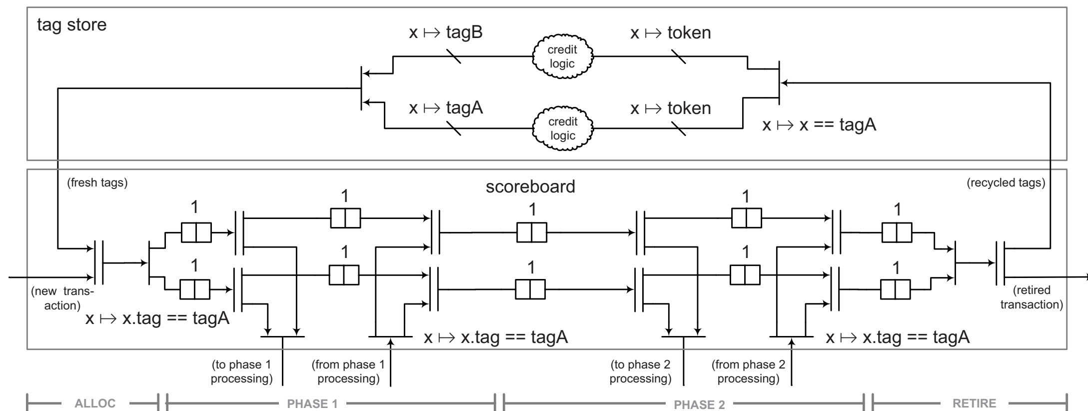

Fig. 11. A fragment of an XMAS diagram showing how a simple scoreboard to support out-of-order processing may be modeled using the primitives of Section II. This scoreboard can track two transactions in flight at a time. The transactions are tracked by their tags (tagA and tagB), and the two credit logic blocks in the tag store ensure that only one tagA and one tagB are in circulation. See Section III-F for details.

TABLE I. Experience on industrial examples. Effort is measured in person days. Since the models are parameterized and hierarchical, we roughly measure the design complexity by the number of unique instances. See text for more details.

Design	Effort	Complexity	Validation
Ring	2	42	Proof
SMF	2	47	Proof
IOF	5	194	BMC & dynamic
NC	10	≈ 200	Paper proof
TD	14	≈ 250	Paper proof

and corrected due to the automatically generated assertions (e.g. mutual exclusion of demux conditions) and build errors and warnings (e.g. dangling channels). Once the model type checks, the first bugs found are actual microarchitectural bugs such as an incorrect timing assumption (often detected by a failing non-blocking assertion) or a deadlock. We find that there are no subtle modeling bugs such as accidentally dropped or overwritten data.

From the C++ description we can generate output for several backends including synthesizable Verilog. This is used for performance analysis, dynamic validation, model checking and coverage analysis. Coverage targets are naturally formulated in terms of irdy and trdy signals and hence can be automatically generated.

B. Design Examples

We have successfully modeled and validated the microarchitecture of a number of complex industrial designs. Table I summarizes our experience. **Ring** is a model of a ring interconnect which was validated for absence of injection starvation and deadlock freedom. **SMF** is a model of a sideband messaging router (and agent environment) which was validated for deadlock freedom in the presence of self-traffic. **IOF** is a pipelined memory switch (and agent environment) allowing peer traffic that was validated for deadlock and starvation. The last two examples are highly concurrent with virtual channels, ordering restrictions, split-transactions and deeply pipelined architectures allowing tens of simultaneous in-flight transactions even in minimal configurations.

In all cases the time taken to build the model was significantly shorter than the other modeling efforts. As a reference point, a SystemC/TLM model of **IOF** without peer traffic took several person-months. Furthermore, our experience developing and maintaining various models of rings in SystemC show that it is relatively easy to introduce bugs due to inadvertent packet overwriting. By switching to a more restricted modeling style as proposed, we believe that such bugs can be avoided improving productivity.

The last two examples in Table I are models of much larger blocks and were modeled at a higher level of abstraction. **NC** is an SoC memory complex and **TD** is a controller in a distributed tag directory. Most of the modeling effort was spent in understanding the microarchitecture from hundreds of pages of documentation. The final model was compact enough to fit on a few sheets of paper. This proved to be indispensible in reasoning about deadlock freedom.

C. Verification Experience

We have found that bug hunting with bounded model checking works well but proofs are hard even for safety properties due to high degree of concurrency and pipelining. Although these models are abstract, since they capture an entire system, they are hard for the model checking tools. For instance, to check agent injection liveness on **IOF** in minimal configuration (585 flipflops) took about 7 days on a 3 GHz Intel Xeon CPU for 25 frames of BMC. Here, the graphical notation has proven to be an indispensable fallback for pen-and-paper reasoning as well as for compositional proofs. Note that this is usually not an option with a more conventional model in say Verilog, Esterel, or Murphi.

Proof Generation. If an xMAS diagram satisfies certain properties, then an important class of safety properties can be proved automatically. This is done by leveraging the high-level structure available in the xMAS diagram to add additional invariants. By adding these invariants, it is possible to make the entire set of properties 1-step inductive, thereby allowing RTL model checkers to prove them easily. For example, since all the queues in a system are easily identified, we can add invariants saying that if the output of a queue satisfies a particular property, then the input of the queue and all packets in the queue must also satisfy the property at all times. It is also possible to infer more global invariants such as those relating the occupancies of several queues in the system. By adding these invariants, the task of the model checker becomes much simpler often allowing it to find otherwise intractable proofs in seconds using induction. The details of proof generation will be presented in a forthcoming paper [5].

D. Modeling Experience

The structural style of modeling we advocate lends itself well to describing highly pipelined, distributed systems with a lot of concurrency. Many microarchitectural blocks and paradigms that have not been discussed in this paper can also be modeled using the proposed framework. As we saw in Section III-E, FSMs that are being implemented by the microarchitecture (such as cache coherence) are easy to describe since the microarchitecture has explicit structures for looking up current state and computing next state. On the other hand, power management FSMs are harder to describe as they monitor periods of inactivity and, unlike the components described in this paper, make progress when irdy signals are low.

Finally, as in any synchronous framework, it is possible to create models with combinational loops. Often the combinational loops can be removed by introducing queues to break cycles, but sometimes this may not be acceptable since adding a queue introduces latency and may break atomicity. If the combinational cycle is constructive [1] it is possible to "cut" it by exploiting a non-blocking assertion in the design.

This is done with a *cut* primitive defined by:

$$\begin{array}{ll} o.irdy := i.irdy & o.data := i.data \\ i.trdy := 1 & \textbf{assert } \mathbf{G}\ (o.irdy \implies o.trdy) \end{array}$$

where the channel connected to port o must be non-blocking (as checked by the assertion).

A more robust approach may be to use synthesis algorithms to de-cyclify these constructive cycles at the cost of modularity.

V. Related Work

Modeling Frameworks. Our focus is on identifying a small set of primitives that is rich enough to model common microarchitectural design patterns used in practice; once the right primitives are identified, they can be expressed in any reasonable modeling framework. For concreteness we provide synchronous semantics for our components but in principle they could be compiled down to Petri nets [20]; be modeled by dataflow actors [14]; in guarded command style [4] or be written as SystemC processes. In the context of any one of these frameworks, the proposed method may be viewed as simply a specific modeling style that promises greater productivity by minimizing the inadvertent modeling bugs.

From a practical point of view, capturing a microarchitecture directly in terms of these primitives (as we do in the xMAS tool) gives us flexibility to generate models for different backends. Thus from the same description one may generate a Petri net, a Verilog model, a SystemC model or a model for Murphi or Spin model checkers allowing the strengths of each modeling framework to be leveraged fully.

NoC Performance Modeling and Synthesis. A number of mathematical and simulations models of communication fabrics for both power and performance are proposed in the networks-on-chip (NoC) literature [12], [17], [22]. A key difference is that unlike NoC simulators our models have a direct path to model checking. A similar consideration applies to processor performance modeling frameworks such as ASIM [9]. Other frameworks such as COSI [21] focus on the synthesis aspect of on-chip interconnect whereas our focus is on clear capture of design intent and validation.

Formal Frameworks. There is a rich literature on specifying systems by a set of actions or transactions that modify shared state [4]; on compiling these specifications to RTL [13]; and on verifying RTL against transaction-level specifications [15]. In this context, our approach may perhaps be seen as a better way to describe RTL. For some systems such as communication fabrics this may obviate the need for a state modification-based specification. Finally, the present notation may be a useful tool to formalize and to generalize the microarchitectural reasoning advocated in HAWK [16] and by Galceran-Oms et al. [11].

Elastic Systems. This work has many similarities with synchronous elastic [6] and latency insensitive [3] systems and indeed the handshake protocol used here is essentially the same as in SELF [6]. The big difference is that our systems are not necessarily latency-tolerant: By introducing the merge and switch primitives, we no longer preserve elasticity by composition. Thus we can guarantee fewer theoretical properties of our systems, but can model a much larger class of practical systems. Interestingly some islands in our models can be elastic, while others may rely on relative timing constraints to preserve correctness.

VI. Conclusion

So far our focus has been on modeling and validating existing microarchitectures. The designs that are most naturally expressed in this notation are highly concurrent and distributed i.e. systems that are designed in a physically-aware manner. By thinking of new microarchitectures in terms of these primitives, it may be possible to obtain designs that admit more efficient circuit implementations compared to designs obtained through standard RTL design which have more centralized control.

Acknowledgment

We thank Alexander Gotmanov and Mark Tuttle for many helpful discussions and Marc Galceran-Oms for reading an early draft.

References

[1] A. Benveniste et al. The Synchronous Language Twelve Years Later, in *Proc. of the IEEE*, 91(1):64-83, Jan 2003.
[2] G. Berry, G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation, *in Science of Computer Programming*, Vol. 19 No.2, pp.87-152, Nov. 1992.
[3] L.P. Carloni, K.L. McMillan and A.L. Sangiovanni Vincentelli. Theory of Latency-Insensitive Design, in *IEEE Trans. on CAD*, Vol.20, No.9, pp. 1059–1076, Sep 2001.
[4] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation,* Addison-Wesley 1988.
[5] S. Chatterjee and M. Kishinevsky. Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics, in *Proc. of Intl. Conf. on Computer Aided Verification*, 2010 (to appear).
[6] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of Synchronous Elastic Architectures. In *Proc. DAC*, 2006.
[7] W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.
[8] D. Dill. The Murphi Verification System, in *Proc. of the Intl. Conf. on Computer Aided Verification*, pp.390–393, 1996.
[9] J. Emer *et al.* Asim: A Performance Model Framework, *IEEE Computer,* vol. 35, no. 2, pp. 68-76, February, 2002.
[10] U. Engberg, P. Grønning, L. Lamport. Mechanical Verification of Concurrent Systems with TLA, in *Proc. of the Intl. Conf. on Computer Aided Verification*, pp.44–55, 1993.
[11] M. Galceran-Oms et al. Speculation in Elastic Systems, in *Proc. DAC 2009*, pp. 292-295.
[12] K. Goossens et al. A Design Flow for Application-specific Networks on Chip with Guaranteed Performance to Accelerate SoC Design and Verification, in *Proc. DATE*, Mar. 2005, pp. 1182-1187.
[13] J.C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions, In *Proc. ICCAD 2000*, pp. 511–518
[14] E. Lee and T. Parks. Dataflow Process Networks, in *Proc. IEEE,* vol. 83, pp. 1-63, May 1995.
[15] Y. Mahajan et al. Verification driven Formal Architecture and Microarchitecture Modeling, in *MEMOCODE,* 2007.
[16] J. Matthews and J. Launchbury. Elementary Microarchitecture Algebra, in *Proc. CAV* 1999.
[17] G. de Micheli and L. Benini. *Networks on Chips*, Morgan Kaufmann, 2006.
[18] K.L. McMillan. The SMV system. Cadence Berkeley Labs, 1999.
[19] H. Mony et al. Exploiting Suspected Redundancy without Proving it, in *Proc. DAC 2005*.
[20] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541580, 1989.
[21] A. Pinto et al. COSI: A Framework for the Design of Interconnection Networks, in *IEEE Design and Test of Computers*, 25(5):402-415, 2008.
[22] H. Wang et al. A power-performance simulator for interconnection networks, in *Proc. Int. Symp. Microarchitecture,* Nov. 2002, pp. 294-305.