Automatic Extraction of Micro-Architectural Models of Communication Fabrics from Register Transfer Level Designs

Sebastiaan J. C. Joosten*† and Julien Schmaltz*
*Eindhoven University of Technology
†Radboud University Nijmegen
s.j.c.joosten@tue.nl, j.schmaltz@tue.nl

Abstract—**Multi-core processors and Systems-on-Chips are composed of a large number of processing and memory elements interconnected by complex communication fabrics. These fabrics are large systems made of many queues and distributed control logic. Recent studies have demonstrated that high levels models of these networks are either tractable for verification or can provide key invariants to improve hardware model checkers. Formally verifying Register Transfer Level (RTL) designs of these networks is an important challenge, yet still open. This paper bridges the gap between high level models and RTL designs. We propose an algorithm that from a Verilog description automatically produces its corresponding micro-architectural model. We prove that the extracted model is transfer equivalent to the original RTL circuit. We illustrate our approach on a typical example of communication fabrics: a scoreboard with credit-flow control.**

I. Introduction

Communication fabrics are key elements of modern multi-core processors and Systems-on-Chip. The interconnection of a large number of processing and memory elements require complex interconnection networks. These networks are characterized by a large number of queues and distributed control. The former induces a large state space and the latter restricts the use of localisation techniques [2]. Formally verifying Register Transfer Level (RTL) designs of such communication fabrics is still an open challenge.

To formally verify communication fabrics, *micro-architectural* models are commonly used. Intel recently proposed a graphical language – called xMAS – for formally specifying such micro-architectures [3]. For this language several properties can be found automatically for reasonably large networks, such as invariants [2], channel type information [5] and deadlock freedom [10]. On the other hand, properties of the high level model improve the effectiveness of hardware model checking [2], [6], [9]. In practice, high-level models are difficult to create and their relation to actual designs is unclear. Note that hardware designs analysed in the aforementioned works are always generated from xMAS models.

This paper proposes an algorithm that extracts an xMAS high-level model from an RTL design. The input of our flow is a Verilog description together with a specification of which modules represent queues, and when messages enter or leave a queue. After parsing the Verilog, the RTL design can be considered a set of queues and registers interconnected by combinatorial logic. Our approach extracts the micro-architectural structure of the design from this unstructured combinatorial logic. This is achieved by identifying when the transfer of messages depends on synchronisation with other messages or on routing or arbitration decisions. The result is an acyclic xMAS network *transfer equivalent* to the original RTL design. This means that a transfer occurs – a message moves from a queue to another one – in the xMAS model if and only if the same transfer occurs in the RTL design. To ensure that the generated xMAS satisfies the assumptions made by various high-level tools such as deadlock detection tools, we check local properties using a standard model checker. Our method has been automated and is available on-line together with its source code and that of the examples used in the paper[1].

This paper introduces the reader to the part of the xMAS specification relevant to this work. In Section III, we define a circuit based on properties, similar to how the next-state function is expressed in model checkers. The translation from Verilog code to such description of a circuit is automated and part of the code available online. In a circuit, we define the notion of port, which is the basic building block for our algorithm. We describe this algorithm in Section IV, and its correctness in Section V. In Section VI, we apply the algorithm on a nontrivial example, and discuss the result. There is no definition of what the 'best' abstraction would be, but the resulting abstraction is suitable for verification. Discussion and conclusion follow in Section VII and VIII

II. xMAS: Micro-Architectural Specification and Verification

An xMAS model is a network of primitives connected via typed *channels*. A channel is connected to an *initiator* and a *target* primitive. A channel is composed of three signals. Channel signal *c.irdy* indicates whether the initiator is ready to write to channel *c*. Channel signal *c.trdy* indicates whether the target is ready to read from channel *c*. Channel signal *c.data* contains data that are

[1] http://www.win.tue.nl/ jschmalt/publications/date15/date15.html

978-3-9815370-4-8/DATE15/©2015 EDAA

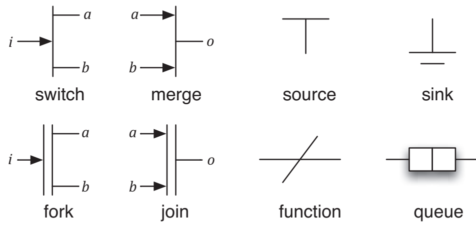
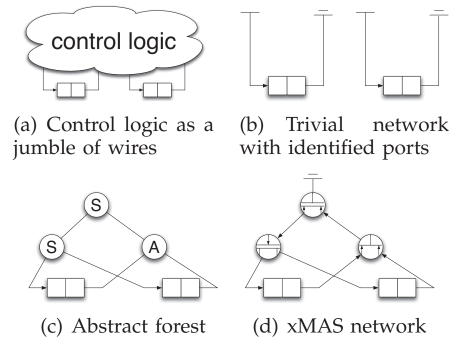


Figure 1. There are eight xMAS components.



(a) Control logic as a jumble of wires (b) Trivial network with identified ports (c) Abstract forest (d) xMAS network

Figure 2. Summary of our method

transferred from the initiator output to the target input if and only if both signals *c.irdy* and *c.trdy* are set to true.

Figure 1 shows the eight primitives of the xMAS language. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Typically, functions are used to convert packet types and represent message dependencies inside the fabric or in the model of the environment. A *fork* duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two outputs are both ready to read. A *join* is the dual of a fork. The function parameter determines how the two incoming packets are merged. A transfer takes place if and only if the two inputs are ready to send and the output is ready to read. A *switch* uses its function parameter to determine to which output an incoming packet must be routed. A *merge* is an arbiter. It grants its output to one of its inputs. The arbitration policy is a parameter of the merge. A *queue* stores data. Messages are non-deterministically produced and consumed at *sources* and *sinks*. Below are the original definitions for fork, join, switch and merge [3]:

- fork: a.irdy := i.irdy ∧ b.trdy, b.irdy := i.irdy ∧ a.trdy, i.trdy := a.trdy ∧ b.trdy.
- join: a.trdy := o.trdy ∧ b.irdy, b.trdy := o.trdy ∧ a.irdy, o.irdy := a.irdy ∧ b.irdy.
- switch: a.irdy := i.irdy ∧ s, b.irdy := i.irdy ∧ ¬s, i.trdy := (a.trdy ∧ s) ∨ (b.trdy ∧ ¬s).
- merge: a.trdy := o.trdy ∧ u ∧ a.irdy, b.trdy := o.trdy ∧ ¬u ∧ b.irdy, o.irdy := a.irdy ∨ b.irdy.

The original description [3] requires s to be a function of the input data, and u to be 'a local state variable that ensures fairness', where fairness means that if o.trdy is high infinitely often, a and b get infinitely many turns (that is: a.trdy and b.trdy will be high).

III. RTL Communication Fabrics

We focus on synchronous RTL: all flops can be perceived as being updated simultaneously, by an implicit clock. While we write RTL in Verilog, we illustrate our approach using an abstraction as a starting point. The disjoint union between sets is written as ⊕, and the set of Boolean values as $\mathbb{B}$.

Definition 1 (Circuit, state, property)**.** *A* circuit *consists of (a finite set of) inputs I and flops F, together with a next-step function $n : (I \oplus F \to \mathbb{B}) \to (F \to \mathbb{B})$. A* state *$\sigma$ of a circuit is a function $\sigma : I \oplus F \to \mathbb{B}$ which assigns a value to every input and every flop. A* property *p in a circuit is a function which, given a state, produces a value $p : (I \oplus F \to \mathbb{B}) \to \mathbb{B}$.*

A communication fabric typically contains a large amount of queues, and a few flops for arbitration. Very often queues are implemented as a separate module. All queue modules are considered black boxes with explicit input and output ports. A queue interface is identified by the following (Boolean) properties:

- r_r The queue is ready to receive.
- r_s The queue is ready to send.
- t_r A packet is added to the queue at the next tick.
- t_s A packet is sent at the next tick.

The queue interface satisfies t_r implies r_r and t_s implies r_s, meaning that whenever a packet is enqueued, the queue must be ready for it, and whenever a packet is dequeued, the queue should have been ready. In most RTL, this is true by design. If not, our method still works when r_r or r_s (or both) are simply defined as true. Together, t_r and r_r give rise to a 'port', as do t_s and r_s. The term *port* is defined as follows:

Definition 2 (port)**.** *A* port *(in a circuit) is an object x for which properties r_x and t_x are defined. We require that $t_x(\sigma) \to r_x(\sigma)$ (for every state σ) and refer to t_x as the transfer property, and to r_x as the implied property.*

IV. Translation of RTL to xMAS

Every queue gives rise to exactly two ports. For each port the transfer property is known, see Figure 2(a). The control logic is modelled as a sink or a source that emits or accepts a packet whenever the routing logic would, see Figure 2(b). As a consequence, each such sink or source depends on a large enough part of the original network such that it mimics the original routing logic. This is not a very satisfactory model of a network: even though this abstraction is sound (the transfer properties are preserved), it does not tell us much about the communication fabric. This structure is roughly the structure we obtain after parsing the Verilog. Hence the step from Figure 2(a) to Figure 2(b) is done using the Centaur Verilog parser [7]. All files corresponding to Figure 2(b) are available online, so installing the parser is optional.

To get a network, we link ports together using xMAS-like elements similar to the merge, join, switch and fork. The xMAS merge and switch are placed into a category we call 'arbiters'. The xMAS join and fork are of the category 'synchronisers'. Linking ports together using these two elements for each link, gives rise to a graph in the form of a forest (a set of trees, with an arbiter or synchroniser at each branch, and a port at each leaf). This forest has arbiters and synchronisers at every branch, a single port at the root, and queue inputs and outputs at the leaves, Figure 2(c). Section IV-A gives the translation from identified ports to such a forest. We place an eager source or sink at each root, and replace the arbiters and synchronisers with their respective components, Figure 2(d). Some arbiters become switches, and synchronisers become forks and joins. This is an orientation step, given in Section IV-B.

We begin with an informal illustration of how synchronisers and arbiters are created. Our first example has ports u and v, and an oracle P modelling packet injection. After parsing the Verilog, assume we obtain the following equations:

$$r_u = B3.\text{trdy} \qquad t_u = \texttt{AND}(B3.\text{trdy}, B1.\text{trdy}, P)$$
$$r_v = B1.\text{trdy} \qquad t_v = \texttt{AND}(B1.\text{trdy}, B3.\text{trdy}, P)$$

The transfer properties are equivalent in every state, that is, $t_u(\sigma) \leftrightarrow t_v(\sigma)$. We create synchroniser s with the transfer property of u and as implied property the conjunction of the implied properties of u and v. This yields a new port s:

$$r_s = \texttt{AND}(B3.\text{trdy}, B1.\text{trdy}) \quad t_s = \texttt{AND}(B3.\text{trdy}, B1.\text{trdy}, P)$$

If u and v are both input ports, the synchroniser is refined into a fork. Figure 3(a) shows the result. If u and v would be output ports, the synchroniser would be refined into a join.

Our second example (Figure 3(b)) has two ports and policy A deciding which port to give the turn. Packets are accepted when property P holds. After parsing the Verilog, assume we obtain the following equations:

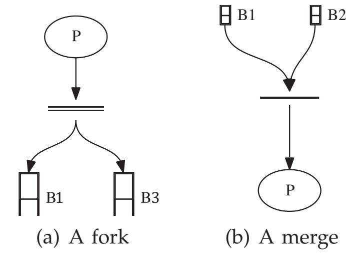
$$r_u = B2.\text{irdy} \quad t_u = \texttt{AND}(B2.\text{irdy}, \texttt{OR}(A, \texttt{NOT}(B1.\text{irdy})), P)$$
$$r_v = B1.\text{irdy} \quad t_v = \texttt{AND}(B1.\text{irdy}, \texttt{NOT}(\texttt{AND}(A, B2.\text{irdy})), P)$$

A SAT solver detects that $t_u \wedge t_v \Rightarrow \bot$. We therefore add an arbiter a with $t_a = t_u \vee t_v$. If u and v are output ports, we refine the arbiter into a merge. If u and v would be input ports then A would be a routing function and we would refine the arbiter into a switch.

A. From ports to a forest

Creating a synchroniser takes two ports and creates a new one. To indicate the new synchroniser, we write a tuple $S\langle x, y\rangle$, where x and y indicate the combined ports (i.e. either queue ports, synchronisers or arbiters). When we create an *arbiter*, there is a switching function s, or an arbitration policy u. A property t will indicate when x should get a turn. An arbiter is written as $A\langle x, y, t\rangle$.

Let x and y be ports. There are two cases in which we combine these ports.



(a) A fork (b) A merge

Figure 3. Output of our method

1) If the transfer properties are mutually exclusive: $t_x(\sigma) \wedge t_y(\sigma) \rightarrow \bot$ for all states σ. In this case, we create an arbiter $a = A\langle x, y, t_x\rangle$. We turn a into a port by choosing $r_a(\sigma) = r_x(\sigma) \vee r_y(\sigma)$ and $t_a(\sigma) = t_x(\sigma) \vee t_y(\sigma)$.
2) If the transfer properties are equivalent: $t_x(\sigma) \leftrightarrow t_y(\sigma)$ for all states σ. In this case, we create a synchroniser $s = S\langle x, y\rangle$. We turn s into a port by choosing $r_s(\sigma) = r_x(\sigma) \wedge r_y(\sigma)$ and $t_s(\sigma) = t_x(\sigma)$.

The reader can verify that the arbiter and a synchroniser, added in the way noted above, preserve $t(\sigma) \rightarrow r(\sigma)$.

To decide whether expressions like $t_x(\sigma) \wedge t_y(\sigma) \rightarrow \bot$ are the case, we use a SAT procedure. Using SAT instead of model checking means that we might overlook some cases: we may erroneously decide that $t_x(\sigma) \wedge t_y(\sigma) \rightarrow \bot$ does not always hold. In that case we fail to create an arbiter (or a synchroniser in the case of $t_x(\sigma) \leftrightarrow t_y(\sigma)$). It is easy to come up with artificial cases where this happens. In our designs (and in xMAS generated Verilog in general), registers are driven independently, and a SAT solver gives exactly the same results as a model checker. We did not come across any designs where a model checker provides more insight than a SAT solver, but this is most likely due to the fact that we used xMAS generated Verilog.

Starting out with a set of ports, we combine them using one of the two methods illustrated above. By making as many combinations as possible, we reduce the number of remaining ports as much as possible. For this reason, we give priority to the creation of synchronisers.

After these steps, it may be possible that a port has an implied property which is equivalent to the transfer property. In case the two differ, we add a sink (or a source) that accepts (or provides) a packet in states for which the transfer property is high. This ensures that every port x satisfies $r_x(\sigma) \leftrightarrow t_x(\sigma)$. Such sinks or sources are denoted as $P\langle t\rangle$ (for 'port'), satisfying $r_{P\langle t\rangle} = t$. In other words: the sink or source is ready to yield a packet at precisely the moment a transfer occurs. We end up with an algorithm for combining ports that gives precedence to adding synchronisers (Algorithm 1, lines 1–17).

After running the algorithm, we obtain a set of xMAS objects with one open end. We discuss what to do with this open end after introducing function orientPortToGraph. Note that the last few lines of Algorithm 1 set r and p in a way that do not effect the output of the algorithm, but they are used in the correctness proof

Data: Set of ports C, maps t and r on C
Result: Reduced set of ports C'

```
1  repeat
2      for x, y ∈ C with x ≠ y do
3          if t[x] ⇔ t[y] then
4              remove x, y from C, add S⟨x, y⟩ to C;
5              let r[S⟨x, y⟩] = AND(r[x], r[y]);
6              t[S⟨x, y⟩] = t[x];
7          end
8      end
9      for x, y ∈ C with x ≠ y do
10         if t[x] ∧ t[y] ⇒ ⊥ then
11             remove x, y from C, add A⟨x, y, t_x⟩ to C;
12             let r[A⟨x, y, t_x⟩] = OR(r[x], r[y]);
13             let t[A⟨y, y, t_x⟩] = OR(t[x], t[y]);
14             break;
15         end
16     end
17 until Nothing changed;
18 Create a new set C';
19 for x ∈ C do
20     if r[x] ⇔ p[x] then
21         add orientPortToGraph(x) to C';
22     else
23         let r[P⟨t[x]⟩] = t[x]; t[P⟨t[x]⟩] = t[x];
24         let r[S⟨x, P⟨t[x]⟩⟩] = t[x]; t[S⟨x, P⟨t[x]⟩⟩] = t[x];
25         add orientPortToGraph(S⟨x, P⟨t[x]⟩⟩) to C';
26     end
27 end
```

Algorithm 1: Reducing ports

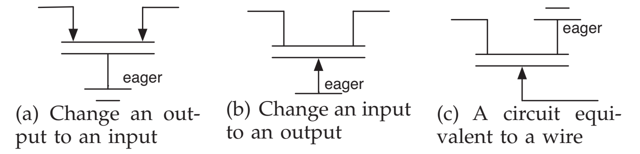
provided in the next section of this paper. By connecting the leaves that correspond to the same buffer (one is an input, the other an output), we obtain an undirected graph. To turn this graph into an abstract network, all edges need to get a direction. This turns synchronisers into forks and joins, and arbiters into switches and merges. That step is done by orientPortToGraph, which we discuss now.

B. Orienting the forest

A synchroniser has three ports, and all of them have a transfer simultaneously. For the three connections leading to a synchroniser, at least one must be an input, and at least one an output. Depending on the third, we then create a fork or a join. This means that when we encounter a synchroniser, we first orient the underlying branches. If both branches are inputs, or both are outputs, the synchroniser is fixed. Otherwise, we choose to orient this branch freely.

In some cases, we need to change the direction of an input or output. Figure 4 shows circuits that achieve this. The first circuit consists of a join connected to an eager sink. This connects to an input, and provide an output (with the same transfer property). The second circuit is roughly the same: a fork with an eager source that connects to an output, providing an input.

Function orientPortToGraph reduces the number of places where the direction of inputs or outputs needs to



(a) Change an output to an input (b) Change an input to an output (c) A circuit equivalent to a wire

Figure 4. Different ways of changing the direction of a port

be changed, mainly for esthetic purposes. We show in the next section that the actual direction does not matter for the correctness of our translation. We omit a pseudocode implementation of the orientation procedure, but invite the reader to look at our Haskell implementation.

Our output is a graph of xMAS components that have their parameters instantiated, but they are not necessarily instantiated in a way that was intended. For example: the routing decision of an xMAS switch only depends on packet data. Our approach applies to such functions that may also depend on inputs, flops, or other queues.

V. Resulting Graph Correctness

We show that the resulting graph is equivalent to the network in the original circuit, using the semantics for the xMAS components as presented in Section II.

On the RTL design, we require the different input and output ports together with the corresponding property t to indicate that a transfer occurs, to be known. At the xMAS level, we know all channels and their transfer conditions. We say that an xMAS model and an RTL design are *transfer equivalent* if and only if their transfer conditions are equivalent. *Transfer equivalence* is defined as follows:

Definition 3 (Transfer equivalence)**.** *Let X be a set of queue inputs and outputs, and let f_1 and f_2 be functions that give the transfer property for every element of X. We say that f_1 and f_2 are transfer equivalent if for all elements $x \in X$ and all states σ: $f_1(x)(\sigma) = f_2(x)(\sigma)$.*

Lemma 1. *Let C be the inputs and outputs of the queues in a circuit with the transfer properties t_x for each $c \in C$. Let $t'_c = c.\text{irdy} \wedge c.\text{trdy}$ where $c.$irdy and $c.$trdy follow the xMAS semantics as extracted from C by the proposed algorithm. Then t and t' are transfer equivalent.*

Proof: To show equivalence, we just need to show that $t \leftrightarrow o.\text{irdy} \wedge o.\text{trdy}$ on a queue output o, and that $t \leftrightarrow i.\text{irdy} \wedge i.\text{trdy}$ on a queue input i. We show that the following properties hold for all ports inductively:

Each input corresponding to port c:

$$t_c \to c.\text{trdy} \quad (1)$$
$$c.\text{trdy} \to r_c \quad (2)$$
$$\neg t_c \wedge r_c \to \neg c.\text{irdy} \quad (3)$$
$$t_c \to c.\text{irdy} \quad (4)$$

Each output corresponding to port c:

$$t_c \to c.\text{irdy} \quad (5)$$
$$c.\text{irdy} \to r_c \quad (6)$$
$$\neg t_c \wedge r_c \to \neg c.\text{trdy} \quad (7)$$
$$t_c \to c.\text{trdy} \quad (8)$$

Taken together, these equations imply $t_c \leftrightarrow c.\text{irdy} \wedge c.\text{trdy}$.

The algorithm combines ports, building a tree structure. On each port that remains, the underlying tree

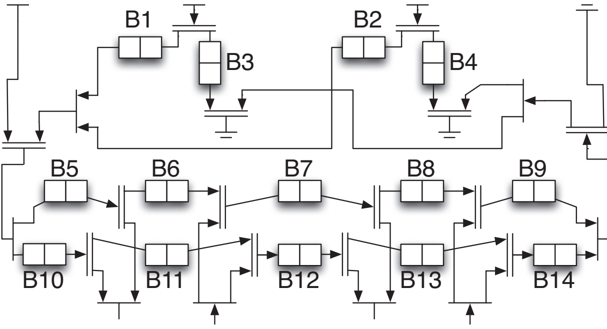


Figure 5. Two entry scoreboard implementation

is connected to an eager sink or source. We perform induction over each such tree structure in two directions. We first do induction over the top equations (queues to trunk), and then - using these equations - over the bottom equations (trunk to queues). Both proofs are by a case analysis over the components, and left largely to the reader due to space limitations. We start with the base cases.

These are the base cases: For the queue output c corresponding to port x, we know $r_x = c$.irdy (Equation 6 holds), and for the queue input c, we know $r_x = c$.trdy (Equation 2 holds). By $t_x \rightarrow r_x$, Equations 1 and 5 hold. At the 'trunk' side of the network, $c \in C'$, there is an eager sink or source. If c is an input then c.irdy is high (eager sink), and c.trdy is high if c is an output, thus Equations 4 and 8 hold. For the corresponding port x we know $r_x \leftrightarrow t_x$, so Equations 3 and 7 hold.

The induction step is by case distinction on the xMAS component and the port direction. The proof for the bottom two equations (Equation 3, 4, 7 and 8) depends on that for the top two (Equation 1, 2, 5 and 6), but not vice versa. This holds for all cases, which we omit due to space limitations.

This completes our induction, which implies that $t_c \leftrightarrow c.\text{irdy} \wedge c.\text{trdy}$ for all queue in- and output ports. ■

The proof is independent of choices from the orientation function, so orientation of synchronisers and arbiters does not change whether transfer equivalence holds. In the networks we obtain, we have the liberty to change merges into switches, forks into joins, and vice versa.

VI. Experimental results

Figure 5 shows a network of a two entry scoreboard with two phases. The top half models tokens preventing an overflow of packets entering the bottom half. The switch at the bottom left is configured such that tokens from B1 are used to route data to B5, while tokens from B2 are used in packets for B10. To illustrate our approach, we applied our method to Verilog generated from this xMAS model. In addition to the Verilog description, our method needs a description of the ports of each queue module. In the case of the queues used in our example, the ports are defined in the following way:

```
port
  direction output;
  module xmas_queue;
  ready o$irdy;
  transfer o$trdy && o$irdy;
endport
```

The output of our method is a graph visualised using the graphvis program 'dot' [4].

Figure 6 shows the structure extracted by our method. Properties used at sources and sinks are labeled with P and a number. While our method largely reconstructed the original design, a few subtle differences arise. For instance, B2 is connected to B10 in the extracted model, while B2 is connected to B5 and B10 in the original xMAS model. Because the switch is configured properly, tokens from B2 are actually routed to B10. Our extraction method makes this fact visible. Also, our method focuses on the control logic. The 'ready' signal of data coming from the input in the top left corner is hidden in P1.

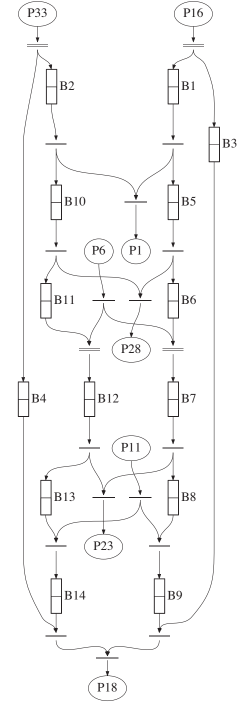


Figure 6. Two entry scoreboard extraction result

Obtaining an xMAS model may provide some insight by itself, but the main reason to want an abstraction, is to be able to perform verification of the design using high level techniques. The method for finding inductive invariants (such as [2]) is directly applicable to our generated graph. Such methods do not require specific xMAS properties like fairness of merges or the restriction that switching functions only depend on message data. Other techniques, like deadlock detection [10], do require such properties. In order to be able to use these techniques, we show that it is feasible to verify fairness properties using standard methods. The approach is as follows: we output a nuXmv [1] description of the remaining flops. We hide the queue implementations (treat them as black boxes, leaving their outputs as undefined variables) in order to reduce the state space. We then formulate the property for a merge $A\langle x, y, t\rangle$ as an LTL sentence expressing that the merge is fair. That is: if y gets turns (so not $\mathbf{G}\,\neg y_t$), and if x requests turns (x_r), then x should get turns as well (x_t). We want the same to hold when we interchange x and y.

$$\mathbf{G}\,\mathbf{F}\,((x_r \rightarrow x_t) \vee \mathbf{G}\,\neg y_t) \wedge \mathbf{G}\,\mathbf{F}\,((y_r \rightarrow y_t) \vee \mathbf{G}\,\neg x_t)$$

The left side of that expression says that eventually x gets a transfer if it is ready, given that y is not blocking. The

right side says the same for y. We check both sides of the expression separately using the model checker nuXmv. The runtime for nuXmv was well under a second. All merges provably satisfy the equation above, which does not come as a real surprise, as they also satisfy this property in the original design.

VII. Discussion

Our method also applied to virtual channels examples analysed in related works [3], [9]. Similar to these related works, our experiments are performed on xMAS generated code. One bias is that these models already contain architectural insight. This bias appears in the generation of the Verilog which is done by translating each xMAS module separately. Note that the design is fully flattened before being processed by our method. The latter captures the parts of a design where packets synchronise with each other or where the progress of packets depends on arbitration or switching decisions. The key element is to identify when transfer properties are mutually exclusive or equivalent. In more realistic and larger cases, these properties are larger and more complex. Still, they always involve a relatively small subset of the design and therefore should stay within the capabilities of SAT solvers.

Another bias is that only a subset of RTL languages is used. In our designs, we could automatically rewrite 4-valued logic to 2-valued logic. In interconnects that use buses, this may not be the case, which still forms a challenge.

Additionally, all registers in our designs are independent. As a result, using SAT solvers instead of model checkers suffices. When a single wire is the driving wire for two different registers, this is not the case. We can craft a network in which a synchroniser is not detected automatically, because two queues use two different registers that always have the same value. There are industrial examples in which this situation arises. The straightforward solution is to use model checking instead of SAT solving. Another solution is to pre-analyse the network in the spirit of [8], and identify such registers.

A final bias is that we only produce xMAS networks in which every cycle contains a buffer. In other words: the method fails to recognise structure when applied to RTL circuits with a cycle without a buffer.

In all cases where our method fails to detect structure, it returns a (non-eager) source or a sink at that place. The translation to a model checking problem can help to distinguish between output which is desired, and output in which still some structure is missing. In other words: it is feasible to detect *all* places where structure is missing. Future work should be able to fill all these gaps, by further investigating the cases where they arise.

Thanks to observations at the xMAS level, intended queues can be implemented more efficiently. The trouble is that once we use these observations, the implementation is no longer pure xMAS. Our approach obtains the routing logic between queues, in a way tolerant for several different queue and buffer implementations.

VIII. Conclusion

We presented an algorithm that automatically extracts the micro-architectural structure of RTL descriptions of communication fabrics. We proved that the original RTL circuit is *transfer* equivalent to the extracted micro-architecture. We illustrated our approach on typical examples found in the literature.

Bridging an important gap between unstructured RTL designs of communication fabrics and their micro-architectural structure, our approach provides designers with key insight into their design. More importantly, micro-architectural models play a key role for verification. Until now, the manual construction of an xMAS model was required to apply all these techniques. Our approach removes this barrier and opens up ways to leverage these previous works to tackle the open challenge of directly verifying RTL designs of communication fabrics.

Acknowledgments

This research is supported by NWO project Effective Layered Verification of Networks on Chips under grant no. 612.001.108 and a grant from Intel Corporation.

References

[1] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Computer Aided Verification. pp. 334–342. Springer (2014)

[2] Chatterjee, S., Kishinevsky, M.: Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. Formal Methods in System Design 40(2), 147–169 (Apr 2012)

[3] Chatterjee, S., Kishinevsky, M., Ogras, Ü.Y.: xMAS: Quick formal modeling of communication fabrics to enable verification. IEEE Design & Test of Computers 29(3), 80–88 (2012)

[4] Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz—open source graph drawing tools. In: Graph Drawing. pp. 483–484. Springer (2002)

[5] van Gastel, B., Verbeek, F., Schmaltz, J.: Inference of channel types in micro-architectural models of on-chip communication networks. In: Proceedings of the 22nd IFIP/IEEE International Conference on Very Large Scale Integration (2014)

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

[7] Hunt, W.A.J., Swords, S.: Centaur technology media unit verification. In: Computer Aided Verification. pp. 353–367 (2009)

[8] Joosten, S.J.C., Schmaltz, J.: Generation of inductive invariants from register transfer level designs of communication fabrics. In: Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on. pp. 57–64. IEEE (2013)

[9] Ray, S., Brayton, R.K.: Scalable progress verification in credit-based flow-control systems. In: Rosenstiel, W., Thiele, L. (eds.) DATE. pp. 905–910. IEEE (2012)

[10] Verbeek, F., Schmaltz, J.: Hunting deadlocks efficiently in microarchitectural models of communication fabrics. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. pp. 223–231. FMCAD '11, Austin, TX (2011)