Automatic Generation of Hardware Checkers from Formal Micro-architectural Specifications

Alexander Fedotov and Julien Schmaltz
Eindhoven University of Technology
a.fedotov@tue.nl, j.schmaltz@tue.nl

***Abstract*—To manage design complexity, high-level models are used to evaluate the functionality and performance of design solutions. There is a significant gap between these high-level models and the Register Transfer Level (RTL) implementations actually produced by designers. We address the challenge of bridging this gap, namely, relating abstract specifications to RTL implementations. An important feature of our proposed approach is to support *non-deterministic* specifications. From such a non-deterministic model, we automatically compute a representation of its observable behaviour. We then turn this representation into a System Verilog checker. The checker is connected to the input and output interfaces of the RTL implementation. The resulting combination is given to a commercial EDA tool to prove inclusion of the traces of the implementation into the traces of the specification. Our method is implemented for the formal micro-architectural description language (MaDL) – an extension of the xMAS formalism originally proposed by Intel – and exemplified on several examples.**

I. Introduction

Interconnects play a crucial role in the correctness and performance of modern Multi-Processors Systems-on-Chips (MPSoC's). The significant number of queues induces a large state space and the distributed control hinders the application of localisation techniques [1]. To combat this challenge, researchers have explored techniques to formally model and analyse *abstractions* of interconnect implementations. In particular, techniques based on the xMAS language proposed by Intel have attracted a lot of interest: modelling [2], verification of safety [3] and liveness properties [4], [5], performance evaluations [6], [7], or asynchronous designs [8], [9].

These analyses are performed on *abstractions*. Establishing a relation between these abstract models and Register Transfer Level (RTL) implementations is the challenge tackled by this paper. A previous approach was proposed by Joosten and Schmaltz [10]. The authors proposed to automatically extract xMAS models from RTL designs. A limitation is that this approach only discovers the basic xMAS primitives. We consider the Micro-architectural Description Language (MaDL), an extension of xMAS with new primitives easing the modelling but making their discovery even more challenging. We propose the generation from MaDL models of *checkers* that can be directly connected to RTL designs. These checkers are used to prove trace inclusion between an RTL design and a MaDL model. The main challenge is that MaDL specifications are *non-deterministic*. There are many possible implementations and a single checker derived from a model encodes all these possibilities.

Our contributions are (1) a method to represent the visible behaviour of MaDL models, (2) a method to turn this representation into a System Verilog checker, and (3) several experimental examples, including a virtual channel and a re-order buffer.

II. Background and Overview

MaDL is an open-source project[1] about a description language and associated analysis techniques. The language originates from xMAS – for eXecutable Micro-Architectural Specifications – proposed by Intel [2]. In contrast to xMAS, MaDL has a textual syntax, recursive data types, loops and parameters. It allows the definition of macro blocks for compositional modelling. Its verification engines implement invariant generation [3], deadlock analysis [4], [5], deadlock reachability [11], and the generation of System Verilog prototypes. We introduce the concepts of MaDL relevant to this paper. More details can be found in the MaDL GitHub pages.

1) Channels, transfers, and persistency: Primitives are connected via typed channels. A channel connects exactly two primitives called the *initiator* and the *target*. A channel consists in three signals:

- `irdy`: high when the channel contains valid data, that is, the initiator is ready to transfer;
- `data`: the data contained in the channel;
- `trdy`: high when the target of the channel is ready to accept data.

When both `irdy` and `trdy` are asserted, the data is transferred from the initiator to the target.

Persistency means that every primitive commits to a transfer, namely, when `irdy` is asserted it must remain high until `trdy` is asserted.

2) Data types and colours: MaDL supports the following data types: constant, enumeration, and struct. The following code snippet declares several constant types, making up enumeration types, used in a struct type.

```
const dst0, dst1, req, rsp;
enum dst_t {dst0;dst1;};
```

[1] https://github.com/MaDL-DVT/madl-dvt

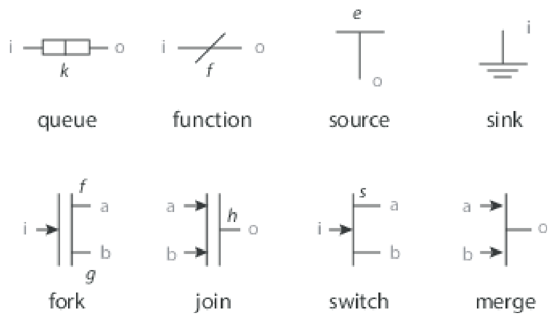

Figure 1. Original primitives of the xMAS language. Italicised letters indicate parameters. Grey letters indicate ports.

```
enum trans_t {req;rsp;};
struct pkt_t { dst : dst_t; type: trans_t; };
```

We often refer to a value of a packet type using the term *colour*. In the above example, a packet with `dst = dst0` and `type = req` is one colour.

3) Primitives: The basic primitives of MaDL are the ones originally proposed in the xMAS language. Their graphical representation is pictured in Figure 1. We briefly sketch their semantics.

A *Source* non-deterministically injects a packet in the network. A *Sink* non-deterministically consumes a packet. Sources and Sinks are assumed to be fair. They always eventually inject or consume packets. A *Fork* consumes the input packet and produces two output packets. This happens if and only if both outputs can accept a new packet. A *Join* consumes two packets and produces one packet. As generally done in related works [4], [5], a join has a control input and a data input. When both inputs have a packet, the packet on the data input is propagated to the output. A *Merge* non-deterministically arbitrates when its two inputs have a packet. Arbitration is left abstract but is assumed to be fair. Each input always eventually is granted. A *Switch* routes the input packet to one output. The decision solely depends on the data contained in the input packet. A *Function* modifies its input packet. A *Queue* stores and forwards packets following a First-In-First-Out (FIFO) policy.

MaDL also adds new primitives. In particular, it adds a non-deterministic demultiplexer, called `LoadBalancer` and a complex sorting primitive called `MultiMatch`. A `MultiMatch` has n outputs each controlled by a match input. It also has m data inputs. At each cycle, a predicate between each data input and each control input is evaluated. If the predicate holds, the data input is forwarded to the output controlled by the match input. Otherwise, the data input stalls. The experimental section shows the construction of a re-order buffer using these two primitives.

A. Running example

The basic syntax of MaDL consists in statements declaring and connecting channels. These statements have the following syntax:

```
chan <outs> := <Primitive> (<ins>);
```

Note that in this example, channels in list "ins" must be declared somewhere else.

Example. *We consider a simple network composed of a source injecting packets with colour* `red`*. Packets are sent to both queues q_0 and q_1. The merge non-deterministically takes one packet from either queue and forwards it to the sink. The corresponding MaDL code is the following:*

```
const red;
chan to_q0, to_q1 := Fork(Source(red));
chan q0_out := Queue(1,to_q0)[q0];
chan q1_out := Queue(1,to_q1)[q1];
Sink(Merge(q0_out, q1_out));
```

This example will further be used as a running example.

III. Interface Behaviour

A. Interface Actions

The interface of a MaDL model consists in all sources and sinks. Let Src be the set of all sources of a given model. Let Snk be the set of all sinks of a given model. For any $s \in \mathrm{Src} \cup \mathrm{Snk}$, let C_s be the set of colours produced by source s or consumed by sink s. A source $s \in \mathrm{Src}$ either stays idle – represented by an Idle action – or requests the injection of a packet with colour $c \in C_s$ – represented by an Inject_c action. A sink $s \in \mathrm{Snk}$ either rejects packets of any colour – represented by a Reject action – or is ready to consume a packet with colour c – represented by a $\mathrm{Consume}_c$ action. The complete set of interface actions is the cross-product of all source and sink actions. The formal definition of actions is as follows:

Definition 1. *(Actions) For any $s \in Src$, let $R_s = \{x \mid x = Idle \vee x = Inject_c, c \in C_x\}$ be the set of* ***source actions*** *of s. For any $s \in Snk$, let $N_s = \{x \mid x = Reject \vee x = Consume_c, c \in C_s\}$, be the set of* ***sink actions*** *of s. Let $R' = \prod_{s \in Src} R_s$ be a cartesian product of sets of actions of all sources from Src. Let $N' = \prod_{s \in Snk} N_s$ be a cartesian product of sets of actions of all sinks from Snk. Then, the set of* ***global actions*** *A is defined as $R' \times N'$.*

Example. *Consider the running example. The set of sources is $Src = \{src0\}$. The set of colours possibly injected at that source is $C_{src0} = \{red\}$. The source therefore has two possible actions: either it is idle or it tries to inject a* `red` *packet. The possible actions at the source are the following:*

$$R_{src0} = \{Idle, Inject_{red}\}$$

Similarly, the sink either consumes a `red` *packet or rejects any packet. The possible actions at the sink are the following:*

$$N_{snk0} = \{Reject, Consume_{red}\}$$

The set of global actions is then the cross product of the source and sink actions. This defines the following set:

$$\begin{aligned} A = \{&(Idle, Reject),\\ &(Idle, Consume_{red}),\\ &(Inject_{red}, Reject),\\ &(Inject_{red}, Consume_{red})\} \end{aligned}$$

B. Action Behaviour

The behaviour of the interface action is represented by a Non-Deterministic Finite Automaton. We now (1) define the notion of states, (2) define possible actions in a state, and (3) define the state update function.

1) NFA states: The state is defined by the states of the queues and the states of the sources. Each queue is represented by an ordered list of the colours stored in the queue. The head of the list corresponds to the head of the queue.

Sources need to be *persistent*. This implicit assumption in MaDL needs to be explicit in the NFA to properly characterise possible legal actions. Persistency of sources means that if a source tries to inject a packet with a given colour – that is, executes an action Inject_c for some colour c – the source is obliged to keep trying to inject this colour until it succeeds. To reflect this, a source is either in a state "free" – where it is free to inject any colour or to remain idle – or in a state "next c" expressing the fact that the source is committed to inject colour c.

The global state of the NFA is defined by the product of the queue states and the source states.

Definition 2. *Given a set of sources Src, for all $s \in Src$, the set of source states of s is defined as $S_s = \{Free\} \cup \{Next_{c_1}, Next_{c_2}, ..., Next_{c_n}\}$, where $c_1, c_2, ..., c_n$ are the colours that can be injected by s. Given a set of queues Q and a set of sources Src, for any $q \in Q$, let S_q be the set of states of q, and for any $s \in Src$, let S_s be the set of states of s. Then, the set of all global states is defined as follows:*

$$S = \prod_{q \in Q} S_q \times \prod_{s \in Src} S_s$$

In the initial state of the NFA, all sources are free and all queues are empty.

Example. *Consider the running example. The states of the source are either idle or committed to colour* `red`*:*

$$S_{src0} = \{Free, Next_{red}\}$$

Given the set of queues $Q = \{q_0, q_1\}$, we define the set of states of q_0 and q_1 as $S_{q_0} = S_{q_1} = \{\varepsilon, red\}$, where ε denotes an empty queue. Finally, the set of global states is the cross product of source and queue states:

$$\begin{aligned} S = \{&(Free, \varepsilon, \varepsilon), (Free, red, red), (Next_{red}, red, red), \\ &(Free, \varepsilon, red), (Free, red, \varepsilon), (Next_{red}, \varepsilon, red), \\ &(Next_{red}, red, \varepsilon), (Next_{red}, \varepsilon, \varepsilon)\}. \end{aligned}$$

2) NFA transition relation: To define the transition relation, we need to specify (1) the possible transitions and their labels, that is, the possible actions in the current state and (2) the state update, that is, the new occupancy of the queues and the new state of the sources.

Persistency creates on global actions the constraint that if a source is in state "next c", the local action at that source must be a re-try of sending colour c, namely, action Inject_c. If a source is free, then a source can try to inject any of the colours defined by its type or choose to remain idle.

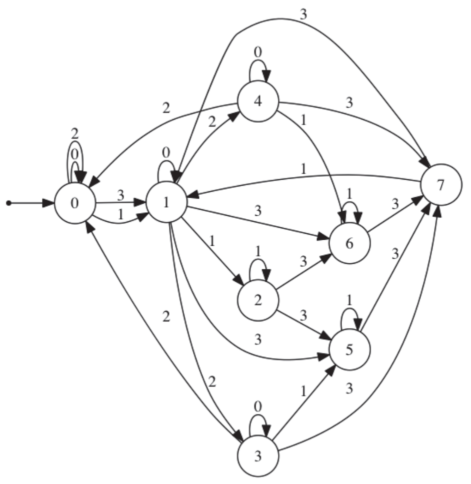

Figure 2. NFA of the MaDL model of the running example.

Definition 3. *Given a global action a, a global state s, action a is a* valid action *if the following condition holds:*

$$s_x = Next_c \rightarrow r_x = Inject_c$$

where $s_x \in S_x$, $r_x \in R_x$.

Except for persistency, actions are unconstrained. In any state, any valid action is possible. The NFA has a transition labelled with this global action in that state.

Example. *Figure 2 shows the states and transitions of the NFA obtained for the running example. In the figure, we use the following labels for action:*

0	$(Idle, Reject)$,
1	$(Inject_{red}, Reject)$,
2	$(Idle, Consume_{red})$,
3	$(Inject_{red}, Consume_{red})$.

Regarding the states, the labels are:

0	$(Free, \varepsilon, \varepsilon)$,
1	$(Free, red, red)$,
2	$(Next_{red}, red, red)$,
3	$(Free, \varepsilon, red)$,
4	$(Free, red, \varepsilon)$,
5	$(Next_{red}, \varepsilon, red)$,
6	$(Next_{red}, red, \varepsilon)$,
7	$(Next_{red}, \varepsilon, \varepsilon)$.

Note, that there are four non-deterministic transitions from state 1. This happens because both queues contain packets, and the merge can decide non-deterministically, from which queue to consume a packet. Also, the NFA is not input enabled. The transitions, that are not allowed according to Definition 3 are absent. Note that this NFA can be reduced as for instance states 4 and 5 are bi-similar. When generating the checker, we basicaly determinize the NFA, and bi-similar states are merged.

3) State update: In a MaDL model, forks and joins ensure that several channels transfer together, that is, they all have their `irdy` and `trdy` signals asserted at the same time. Wouda and Schmaltz formalised this notion of *transfer islands* [11]. A transfer island is a set of channels such that a channel fires – `irdy` and `trdy` are asserted – if and only if all other channels in the island also fire.

Definition 4. *Let M denote the set of channels of a MaDL model. A transfer island is a non-empty set of channels $I \subseteq M$, such that for any $x \in I$, x transfers – $x.\texttt{irdy} \wedge x.\texttt{trdy}$ – if and only if all the channels from $I \backslash \{x\}$ transfer as well.*

We denote the set of all transfer islands of a given MaDL model by I'.

Example. *Consider the running example. Assume output channels are named by appending '_out' to the name of the component. An additional 'a' is used to denote the top and bottom outputs. The set of islands is the following:*

$$I' = \{(scr0{:}red\_out, frk0\_outa, frk0\_outb) \\ (q0\_out, mrg0\_out) \\ (q1\_out, mrg0\_out)\}$$

The first island groups together all channels between the source and the two queues. The other two islands identify the selection by the merge of one of its input.

To manipulate islands, we define the input component of a transfer island. An input component of a transfer island is a component, the output channels of which are in the island. Similarly, an output component of a transfer island is a component, the input channels of which are in the island.

Definition 5. *Given a transfer island x and a component p, let Out_p denote the set of output channels of p, and let In_p denote the set of input channels of p. We call p an* ***input component*** *of transfer island x, if $Out_p \cap I \neq \varnothing \wedge In_p \cap I = \varnothing$. We call x an* ***output component*** *of I, if $Out_x \cap I = \varnothing \wedge In_x \cap I \neq \varnothing$.*

We are ready to introduce the firing conditions for an island. Basically, an island fires when its input sources and the output sinks are non-idle, the input queues are non-empty, and the output queues are non-full. Given a global action, a global state, and an island I, the island is able to transfer if the following holds:

- for all $x \in \text{Src}$, if x is an input component of I, then the action of x is $Inject_c$, where $c \in C_x$,
- for all $q \in Q$, if q is an input component of I, then the state of q is non empty,
- for all $y \in \text{Snk}$, if y is an output component of I, then the action of y is Consume_c, where c is the packet that is transferred to y,
- for all $z \in Q$, if z is an output component of I, then the state of z is not full.

Given a global action and a global state, all queues of all firing islands are updated. This models the synchronous update of all queues. Each queue of a firing island I is updated in the following way:

- for all queues $q \in Q$, if q is an input component of I, then dequeue from the state of q,
- for all queues $q \in Q$, if q is an output component of I, then enqueue a packet into q.

Given a source $x \in \text{Src}$, its state s_x and its action a, for all packets $c \in C_s$ that s can inject and for all transfer islands $I \in I'$, the way source state is updated as follows.

For all transfer islands $I \in I'$, if:

- $s_x = \text{Next}_c$,
- $a = Inject_c$,
- x is an input component of I,
- I can transfer,

then the successor of s_x is Free.

If there exists an $I \in I'$, such that I cannot transfer and the following holds:

- $a = Inject_c$,
- x is an input component of I,

then the successor of s_x is Next_c.

Given a global state and a global action. Let M be the set of all merges, LB be the set of all loadbalancers, MM be the set of all multimatches, and I_{trans} be the set of islands that can transfer for the given state and action. For all $m \in M$, let In_m be the set of input channels of m. For all $l \in \text{LB}$, let In_l be the set of input channels of l. For all $n \in \text{MM}$, let InM_n be the set of match input shannels of n and InD_n be the set of data input channels of n. If $\text{In}_m \cap I_{\text{trans}} > 1 \vee \text{In}_l \cap I_{\text{trans}} > 1 \vee (\text{InD}_n \cap I_{\text{trans}} > 1 \wedge \text{InM}_n \cap I_{\text{trans}} > 1)$, then we need to consider a set of possible island transfers, such that there are no conflicting transfers (only one input channel transfers at a time). Computing several successor states for a given state and action nondeterminism arises.

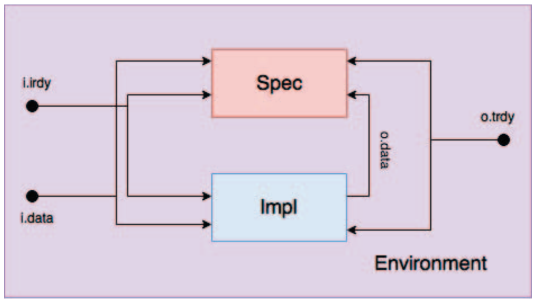

Figure 3. Overall approach

IV. System Verilog Checkers

Given a MaDL specification, we generate a System Verilog checker. In addition to the clock and reset inputs, this checker has the following interface:

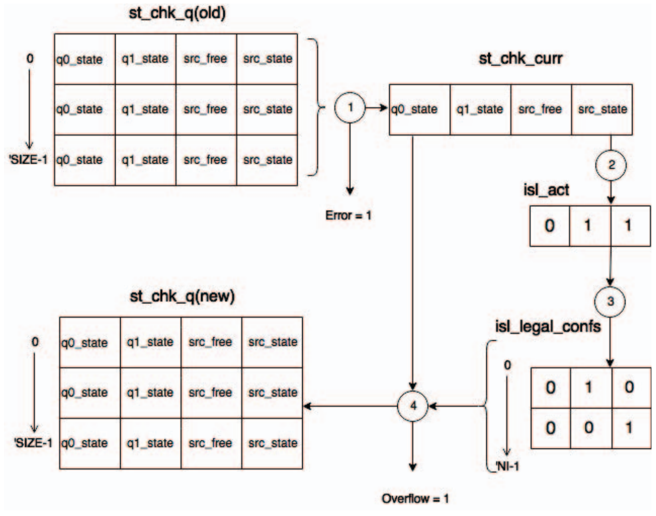

Figure 4. Core checker computations

- for each source $s \in$ Src, a s.irdy input of type **bit**, and a s.data input of the corresponding type;
- for each $k \in$ Snk, a k.trdy input of type **bit**, and a k.data input of the corresponding type,
- two flags named Error and Overflow.

Figure 3 illustrates the connection of the checker – denoted by 'Spec' – to an RTL implementation – denoted by 'Impl'. Both 'Spec' and 'Impl' are driven by the same source and sink actions. The data produced by 'Impl' are fed to 'Spec'. The checker will raise the Error flag when it detects an illegal action. Proving that this flag is always low proves trace inclusion between 'Spec' and 'Impl'. As explained below and because of non-determinism, the checker maintains a queue of all possible current states. Computing the size of this queue is a very difficult problem. The overflow flag indicates an overflow on that queue. The size of this queue is a parameter in the System Verilog description and can easily be adjusted.

It is important to notice that in order our checkers can be processed by formal verification tools, the code must be restricted to the synthesizable subset of System Verilog.

The main computations performed by the checker are pictured in Figure 4. To represent non-determinism, the checker maintains a queue containing all possible current states. Let us call this queue *st_chk_q*. Each state consists in the following elements:

- for each source: a commitment flag *k_free* that is high when the source is in a Next state and a *k_state* variable representing the injected colour.
- for each queue: a queue state identifier *q_state*.

Instead of representing each queue by an actual queue in the checker, we precompute for each queue all possible state transitions in a case statement. This effectively results in a large increase in the number of gates but a major decrease in the number of flops. This trade-off results in more efficient formal verification.

1) Step 1: The first step is to pick the first state in *st_chk_q*. Let us call this state *st_chk_curr*. It is checked whether the current actions are legal w.r.t. this state. If not, the Error flag is raised and computation stops. Otherwise, the computation of the possible next states is started.

Example. *Consider the running example. It has only one source and therefore only one invalid action. Given a state x and a source s, an action is illegal iff:*

$$\neg x.s\_free \wedge (\neg s.irdy \vee s.data \neq x.s\_state)$$

2) Step 2: The second step is to compute the islands that are possibly active in the current states. This results in a bit vector where each position indicates whether the islands is active or not. Let us call this bit vector *isl_act*.

Example. *Let us recall that the set of islands is* $I' = \{(scr0{:}red\_out, frk0\_outa, frk0\_outb), (q0\_out, mrg0\_out), (q1\_out, mrg0\_out)\}$. *Consider state* $(Free, red, red)$, *and action* $(Inject_{red}, Consume_{red})$. *The first island is inactive, since both queues are full. The second and the third islands are active, since the queues are non-empty, and the source action is* $Consume_{red}$. *Hence,* $isl\_act = 011$.

3) Step 3: In general, not all the active islands in *isl_act* can fire simultaneously. For instance, two islands with a common arbiter can be active in the current state but the arbiter must make a non-deterministic choice. The third step is to extract from the current active islands the set of possible legal island configurations. This is a queue of bit vectors. Let us call this queue *isl_legal_confs*.

Example. *In case* $isl\_act = 011$, *the second and the third islands are active, but cannot fire simultaneously due to the common arbiter. Hence, we split 011 in the following way:* $isl\_legal\_confs = \{010, 001\}$.

4) Step 4: Finally, for each legal configuration a new state is computed and enqueued in a new global state *st_chk_q*. When this queue is full, but a new state should be enqueued, the Overflow flag is raised.

Example. *Again, consider state* $(Free, red, red)$, *and action* $(Inject_{red}, Consume_{red})$. *For these state and action,* $isl\_legal\_confs = \{010, 001\}$. *Thus, we compute two state updates. As the queues are full and the source tries to inject a packet, the next state of the source in each case is "Next". Finally, we obtain two distinct successor states* $st\_chk\_q = \{(Next_{red}, \varepsilon, red), (Next_{red}, red, \varepsilon)\}$.

V. Experimental Results and Discussion

Experiments are conducted using five MaDL examples. `SM` is the running example. `SMC` is the same example but (1) the fork is replaced by a switch and (2) the source injects two colours. `SLB` is composed of a source, a load balancer with three outputs. Each output is connected to a queue. All queues are connected to forks and then a sink. `VC` is a virtual channel. The

name	flops (spec.)	flops (impl.)	time (proof)	time (!ovf)	time (cex)
SM	31	11	0,5	0,5	0,1
SMC	23	11	0,1	2,0	0,1
SLB	101	17	8	3,7	0,6
VC	28	27	0,2	0,1	0,1
ROB	61	18	17,5	17,6	1,9

Table I. Experimental results

implementation uses credit-flow control and corresponds to Figure 3 by Ray and Brayton [1]. The specification simply consists in two independent queues, each one with its own source and sink. There is a large gap in the structure of the two circuits. ROB is a re-order buffer with two inputs and one output and defined as follows:

```
struct pkt { tp: [0:0];};
pred f (p: pkt, q: pkt) { p.tp == q.tp};
bus<2> j;
bus<2> i   := LoadBalancer(Source(pkt));
let j[0]   := Queue(1,i[0]);
let j[1]   := Queue(1,i[1]);
chan q_m   := Queue(1,Source(pkt));
chan o_s   := MultiMatch(f,q_m,j);
Sink(o_s);
```

Packets enter at the source feeding the LoadBalancer. Packet leaves at the output of the MultiMatch in the order given by the second source.

Specifications are obtained by translating the MaDL models into System Verilog checkers. Implementations are obtained by translating the same MaDL models into Verilog, except for VC where two different MaDL models are used. When generating implementations, non-determinism is removed. All merges and load balancers implement a round-robin policy. All queues are circular buffers. The combination of a specification and an implementation are given to a commercial formal verification tool to check if there is an unbounded proof of the safety properties that the Error flag and the Overflow flag are always low. Proof times in the table are only given for the Error flag. Experiments are run on a CentOS 6.8 server with 4x16 AMD Opteron 6276 2,3 GHz processors and 128GB 1600MHz memory. The first two columns give the number of flops of the specifications and the implementations. The next columns give the time to prove trace inclusion, to prove absence of overflows, and to find a counter-example. Errors are injected in implementations by either modifying queue sizes or leaving the data input a free variable. In all cases, an illegal action is correctly and quickly detected.

As expected, the number of flops is larger for specifications than for implementations. This is due to non-determinism. SLB shows that a LoadBalancer introduces more non-determinism than a merge. This is where the increase in the number of flops is the highest (6x). ROB has a LoadBalancer and a MultiMatch, two components with a high degree of non-determinism and complex logic. Predictably, ROB is the hardest example for the verification tool, with a proof time of 17,5 seconds. Note that even if the specification of VC is structurally completely different from its implementation, the verification takes a negligible amount of time.

VI. Conclusion

We presented a method to bridge the gap between high-level specifications and RTL implementations. We turned a state-based *non-deterministic* specification into a hardware checker for trace inclusion. The increase in state holding elements is kept under control. We exemplified our approach on several examples including a credit-flow virtual channel and a re-order buffer.

Acknowledgments

This research is supported by NWO the Dutch Foundation for Scientific Research under grant no. 612.001.302.

References

[1] S. Ray and R. K. Brayton, “Scalable progress verification in credit-based flow-control systems,” in *DATE*, W. Rosenstiel and L. Thiele, Eds. IEEE, 2012, pp. 905–910.

[2] S. Chatterjee, M. Kishinevsky, and Ü. Y. Ogras, “xMAS: Quick formal modeling of communication fabrics to enable verification,” *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.

[3] S. Chatterjee and M. Kishinevsky, “Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics,” *Formal Methods in System Design*, vol. 40, no. 2, pp. 147–169, Apr. 2012.

[4] F. Verbeek and J. Schmaltz, “Hunting deadlocks efficiently in microarchitectural models of communication fabrics,” in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD ’11, Austin, TX, 2011, pp. 223–231.

[5] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, “Verifying deadlock-freedom of communication fabrics,” in *Verification, Model Checking, and Abstract Interpretation (VMCAI ’11)*, vol. 6538, 2011, pp. 214–231.

[6] D. E. Holcomb and S. A. Seshia, “Compositional performance verification of network-on-chip designs,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 33, no. 9, pp. 1370–1383, 2014. [Online]. Available: https://doi.org/10.1109/TCAD.2014.2331342

[7] X. Zhao and Z. Lu, “Heuristics-aided tightness evaluation of analytical bounds in networks-on-chip,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 986–999, 2015. [Online]. Available: https://doi.org/10.1109/TCAD.2015.2402176

[8] F. P. Burns, D. Sokolov, and A. Yakovlev, “A structured visual approach to GALS modeling and verification of communication circuits,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 6, pp. 938–951, 2017. [Online]. Available: https://doi.org/10.1109/TCAD.2016.2611508

[9] F. Verbeek, S. J. C. Joosten, and J. Schmaltz, “Formal deadlock verification for click circuits,” in *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 183–190.

[10] S. J. C. Joosten and J. Schmaltz, “Automatic extraction of micro-architectural models of communication fabrics from register transfer level designs,” in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, 2015, pp. 1413–1418. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757140

[11] S. Wouda, S. J. C. Joosten, and J. Schmaltz, “Process algebra semantics & reachability analysis for micro-architectural models of communication fabrics,” in *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, 2015, pp. 198–207. [Online]. Available: https://doi.org/10.1109/MEMCOD.2015.7340487