Scalable Liveness Verification for Communication Fabrics

Sebastiaan J.C. Joosten and Julien Schmaltz
School of Computer Science, Open University of The Netherlands
Institute for Computing and Information Sciences, Radboud University Nijmegen
Eindhoven University of Technology, The Netherlands

***Abstract*—In the realm of multi-core processors and systems-on-chip, communication fabrics constitute a key element. A large number of queues and distributed control are two important aspects of this class of designs. These aspects make decomposition and abstraction techniques difficult to apply. For this class of designs, the application of formal methods is a real challenge. In particular, the verification of liveness properties is often intractable. Communication fabrics can be seen as a set of queues and flops interconnected by combinatorial logic. Based on this simple but powerful observation, we propose a novel method for liveness verification. Our method directly applies to Register Transfer Level designs. The essential aspects of our approach are (1) to abstract away from the details of queue implementations and (2) an efficient encoding of liveness properties in an SMT instance. Experimental results are promising. Designs with hundreds of queues can be analysed for liveness within minutes.**

I. Introduction

Going parallel is the major trend to gain performance out of more transistors [5]. Communication fabrics – also called Networks-on-Chips – have become a key component in the design and verification of multi-core architectures [1], [6]. Formal guarantees about the correct behaviour of communication fabrics is key to ensure system correctness. These on-chip communication networks constitute a class of systems characterised by a large number of queues and distributed control. The former induces a large state-space. The latter prevents the application of abstraction techniques – e.g., localisation [12]. In particular, verifying liveness of communication fabrics is a very hard problem [3].

In this context, recent results rely on the use of high-level models to extract invariants, which are then used to improve hardware model checking [3], [7], [12]. These approaches provide promising results. Still, they require a high-level model, which is not always available or does not directly correspond to the actual Register Transfer Level (RTL) design.

We propose a novel approach for liveness verification of RTL designs of communication fabrics. Our approach does not require a high-level model and scales up to designs with hundreds of queues. Similar to all related works, our method is sound but incomplete. The central ideas of our technique are an abstraction from the details of queue implementations, an abstraction from timing details, and the expression of liveness as the average values of wires along lasso runs. Another important part of our approach is to use recent results showing that many inductive invariants can be derived automatically [9]. These invariants are very similar to those generated from high-level models, e.g., by Chatterjee and Kishinevsky [3]. We then describe a network and liveness properties as a satisfiability modulo the theory of reals and integers (SMT) problem.

978-3-9815370-2-4/DATE14/©2014 EDAA

Queues are replaced with uninterpreted functions and a set of *queue properties*. These properties capture essential characteristics of any queue. Our SMT encoding distinguishes three behaviours of a wire: the initial value until the start of the lasso, its value at the start of the lasso (we call this time now), and its *average* value over the lasso. Using these average values, liveness of a wire means that its average value is not 0. If a wire is not 0 on average, it is 1 infinitely often. To relate all values, we add several properties to the SMT instance. In doing so, we abstract away from irrelevant timing details. The soundness of our approach follows from the correctness of these individual properties. Experimental results show that our method can prove liveness of realistic and large designs. Our method fails to prove some artificial cases. Section III-B gives an analysis on when this occurs, and on how one could circumvent this. Our contributions are summarised as follows:

- We propose a novel liveness verification approach dedicated to communication fabrics.
- We provide experimental results showing its benefits and limitations.
- The implementation of our approach together with examples are publicly available at: `genoc.cs.ru.nl`

II. Principles

A. A simple example

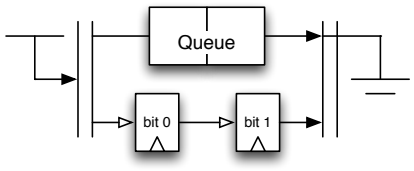


Fig. 1: A queue and two flops.

To illustrate our approach, consider the example in Figure 1. It shows a small network consisting of a source injecting packets, two single-bit flops, a queue, and a sink consuming packets. Whenever a packet enters the queue, the first bit is raised. If the first bit is high, and the second is low, the bits swap values at the next clock cycle. A packet leaves the queue

if it exists, the sink is available, and the second bit is high. As a packet leaves the system, the second bit is set to zero. Should the first bit be high, the system is considered full and no packets can enter the queue.

Our method can be applied with different queue implementations, but we focus on one specific queue implementation in this paper. Its interface is shown in Figure 2. For our imple-

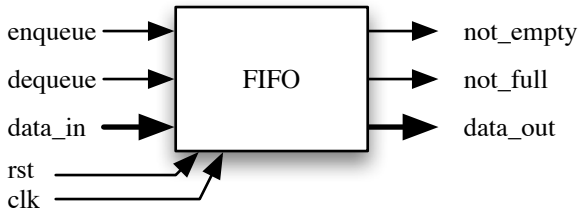


Fig. 2: Queue interface.

mentation, a high `enqueue` wire puts the packet available on `data_in` in the queue if and only if the queue is not full. Similarly, the `dequeue` wire may take a high value while the queue is empty without changing the queue state.

Consider the liveness property stating that packets leave the queue infinitely often, formally expressed as:

$$\square\lozenge\texttt{not\_empty} \wedge \texttt{dequeue}$$

In this example, we also have the following equivalences:

$$\square\lozenge\texttt{enqueue} \leftrightarrow \square\lozenge\neg\ \texttt{bit 0}$$
$$\square\lozenge\texttt{dequeue} \leftrightarrow \square\lozenge\ \texttt{bit 1}$$

Put more generally, liveness of the entire system is implied by packets flowing in the queues. We focus on the property that any packet in any queue is eventually able to leave.

The proof of this property is by contradiction, i.e, we are looking for a counter-example. Such a counter-example is an infinite run of the system in which from some time T wire `not_empty` or wire `dequeue` is always zero. Such infinite runs are represented by finite runs with a *lasso* (See Theorem 9 in [2]). For this proof to work, we assume fairness of all sinks and sources, i.e., the source offers a packet infinitely often, and the sink is ready to accept infinitely often.

The core idea of our approach is to express the fact that 'a wire is stuck at zero' as 'the average value of the wire over the lasso is equal to zero'. We develop an SMT encoding of this property together with relevant information about the network. Our encoding is an over approximation as we only encode properties of the network which do not depend on timing. For instance, we add to the SMT encoding the invariant that the sum of the two bits equals the number of packets in the queue. In this example, this invariant is sufficient to prove liveness of the queue. Until now, our technique has been able to prove liveness of many realistic examples.

B. Runs and lasso's

We briefly justify the fact that infinite runs of communication fabrics are represented by finite runs with a lasso. The argument is completely similar to the one used to justify bounded model checking (e.g. Theorem 9 of [2]). Since there are only finitely many possible states of the network, some of the states are in the run infinitely often (after time τ). Since the sink and the source are fair, the events of offering and accepting packets occur infinitely often as well. This means that we pick two times (after time τ), say time n and time $n+l$, such that:

- the state of the network at time n is identical to the state at time $n+l$
- between time n and time $n+l$, a packet is offered at least once, and the sink is ready to accept at least once

From this run, we construct a lasso run as follows. Up to time $n+l$, our constructed lasso run is identical to the original run. At time $n+l$, all inputs start behaving exactly as they have at time n, such that the state at time $t+l$ is exactly the same as at time t for $t \geq n$. Since $n \geq \tau$, our new run is also a counter-example.

Formally, we consider a lasso run of a network that is described in terms of some *input bits* X. By input bits, we mean all bits that are relevant to the current state. This includes input wires, flop values, and possibly the output of some 'black box' components. We describe the run as follows:

1) For every input bit $x \in X$ and time $t \in \mathbb{N}$, $v(x,t) \in \mathbb{B}$ describes the value of bit x at time t. When we combine input bits to create some new value w (e.g. $w = \texttt{x}_1 \ \wedge\ \texttt{x}_2$), we write $v(w,t)$ for its value at time t (for our example: $v(w,t) = v(x_1,t) \wedge v(x_2,t)$).
2) The run has a lasso of l clock ticks, starting at time n. That is: $v(x,t) = v(x,t+l)$ for $t \geq n$ (and for all wires).

Note that $v(x,t)$ is not a variable that is calculated anywhere in our approach, nor is it one that occurs in the final SMT encoding. Instead, we will relate all variables of the SMT encoding to v, such that any run gives an assignment to the SMT variables. Soundness of our approach will follow from checking every property added to the SMT instance against its interpretation according to v.

C. Encoding liveness as averages

To describe the counterexample, we need to say something about the long term behaviour of the property "the queue is not empty and the `dequeue` signal is high", i.e. $\texttt{not\_empty} \wedge \texttt{dequeue}$. Such a property is described in terms of the input wires X. In this example, the property requires that the `not_empty` wire is high, the sink is high (i.e. accepts), and `bit 1` is high. We define the following property specific variables:

T_w A *persistency variable* for a property w. Boolean T_w is true iff: $v(w,t) = 1$ for $t \geq n$.

F_w We write F_w as a shorthand for $T_{\neg w}$ (which is true iff $v(w,t) = 0$ for $t \geq n$).

a_w An *average value* to relate several properties in the lasso, this is a fraction:

$$a_w = \sum_{t=n}^{n+l-1} \frac{v(w,t)}{l}$$

We can now express 'never w', by 'F_w'. In our example, we aim at disproving $F_{\texttt{not\_empty}\wedge\texttt{dequeue}} = F_{\texttt{not\_empty}\wedge\texttt{bit 1}\wedge\texttt{sink}}$. To prove that this results in a contradiction, we use several other properties of the network.

We take the reader through several properties that apply to networks in general by investigating the example in Figure 1. To distinguish the general properties added to all networks from those specific to the example, we number all properties which end up as assertions in the final SMT instance. The first properties directly follow from the definitions of T, F, and a.

$$T_w \leftrightarrow (a_w = 1) \quad (1) \qquad F_w \rightarrow \neg v(w,n) \quad (4)$$
$$F_w \leftrightarrow (a_w = 0) \quad (2) \qquad T_w \rightarrow v(w,n+1) \quad (5)$$
$$T_w \rightarrow v(w,n) \quad (3) \qquad F_w \rightarrow \neg v(w,n+1) \quad (6)$$

To express Equations 3 to 6, we use a description of the network state at time n. For this purpose we use the following variables to describe the state of the network:

c_x A *current value* for every input bit x. This is an SMT Boolean: $c_x = v(x,n)$.

In the SMT instance, we express Equations 3 and 4 in terms of c. For $v(w,n+1)$, in many cases this can be expressed in terms of the previous state, thus using c. If $v(w,n+1)$ depends on the value of input wires at time $n+1$, this is not possible. In that case, we simply omit Equations 5 and 6.

D. Relating average values

For our example, using that F_w implies $\neg v(w,n)$, we see that either $\neg c_{\texttt{not\_empty}}$ or $\neg c_{\texttt{bit 1}}$ or $\neg c_{\texttt{sink}}$. We also use that we are in a lasso. This implies that if `bit 1` is raised, it must also be lowered at some point for the state of the network to return to the current state. Indeed, for flop x with driving wire d we know:

$$a_d = a_x \qquad \text{if } d \text{ drives flop } x \qquad (7)$$

In addition to relating T and F to averages, we also relate averages amongst themselves. In particular:

$$a_u \leq a_v \qquad \text{if } u \rightarrow v \qquad (8)$$
$$a_u + a_v \leq a_w + 1 \qquad \text{if } u \wedge v = w \qquad (9)$$

To recognise equalities like $u \rightarrow v$ and $u \wedge v = w$, we use the rewrite system from [9], namely:

$$a_{\neg p} = a_{true} - a_p$$
$$a_{p\vee q} = a_p + a_q - a_p \otimes a_q$$
$$a_{p\wedge q} = a_p \otimes a_q$$

The reader may verify that the system above reduces to a system with only $+$ and $\otimes$. The operator $\otimes$ is not introduced, as it should be treated as intermediate syntax. We eliminate $\otimes$ by using:

$$(a_x + a_y) \otimes a_z = a_x \otimes a_z + a_y \otimes a_z$$
$$a_z \otimes (a_x + a_y) = a_x \otimes a_z + a_y \otimes a_z$$
$$a_x \otimes a_y = a_{x\wedge y}$$

If these were averages over a single time unit ($l = 1$), $\otimes$ can be interpreted as multiplication, and the equations would be in the 'Arithmetic sum-of-product format' considered in [10]. To see that for other values for l, note that the values for a in their rewritten form are linear equations. Doing the rewriting first (on v, using multiplication for $\otimes$), and then taking the average, will result in the same equations as taking the average first, and then rewriting (using $\otimes$ as uninterpreted syntax).

This allows us to write down a linear expression in place of every a_w, such that each resulting term a_c has just a conjunction of input bits as c. The averages can then more easily be related amongst themselves using Equations 8 and 9.

We related the values of T and F to the averages a, and the averages amongst each other. For our example:

$$a_{\texttt{not\_empty}} + a_{\texttt{bit 1}\wedge\texttt{sink}} \leq a_{\texttt{not\_empty}\wedge\texttt{bit 1}\wedge\texttt{sink}} + 1$$

A property of the queue is that if no packets leave in the lasso, $a_{\texttt{not\_empty}}$ is either 0 or 1. This implies that for $a_{\texttt{not\_empty}\wedge\texttt{bit 1}\wedge\texttt{sink}} = 0$ to hold, two situations can apply:

- $a_{\texttt{not\_empty}} = 0$: the queue remains empty while the bits are high (otherwise a packet could enter)
- $a_{\texttt{not\_empty}} = 1$: the queue remains full, and both bits are low (since $a_{\texttt{bit 1}\wedge\texttt{sink}} = 0$)

To exclude these two situations, we use the following inductive invariant. Suppose q denotes the number of packets in the queue, f_0 denotes the value of `bit 0`, and f_1 that of `bit 1`:

$$q = f_0 + f_1$$

Such invariants can be found automatically [9].

To relate the invariant to our instance, we need to define integers to describe the state:

q_i A *state variable* for every queue i, which is an integer indicating the number of packets in queue i at time n.

f_i A *state variable* for every flop i, which is an integer indicating its output at time n.

For flops, this state variable is related to our instance via:

$$c_x \leftrightarrow (f_i = 1) \qquad (10)$$
$$\neg c_x \leftrightarrow (f_i = 0) \qquad (11)$$

E. Queue properties.

For every queue i of size s, we used the following properties:

$$0 \leq q_i \leq s \qquad (12)$$
$$q_i = 0 \leftrightarrow \neg c_{\texttt{not\_empty}} \qquad (13)$$
$$q_i = s \leftrightarrow \neg c_{\texttt{not\_full}} \qquad (14)$$
$$T_{\texttt{enqueue}} \rightarrow T_{\texttt{not\_empty}} \qquad (15)$$
$$T_{\texttt{dequeue}} \rightarrow T_{\texttt{not\_full}} \qquad (16)$$
$$F_{\texttt{enqueue}} \rightarrow \left(c_{\texttt{not\_empty}} \rightarrow T_{\texttt{not\_empty}}\right) \qquad (17)$$
$$a_{(\texttt{enqueue}\ \wedge\ \texttt{not\_full})} = a_{(\texttt{dequeue}\ \wedge\ \texttt{not\_empty})} \qquad (18)$$

Equation 18 states that for every packet entering the queue in the lasso, it must also leave, for the queue to return to its original state.

Some networks contain data dependencies. Instead of looking at all possible packets in brute force, we form different expressions for several data types. We consider a packet to be of a certain type, if some set of its bits is high. To determine what types to consider, we find out what data bits occur in the averages found among Equation 18. Each average variable $a_{x_1 \wedge x_2 \wedge \cdots}$, containing some data-bits in its conjunction (among the x's) constitutes a type of packet. For these types, we add a variable:

- d_i state variable indicating the number of packets for a particular data type on some queue.

We write `dto` to indicate that the output data has the type we care about, and `dti` for input. These properties hold for all queues:

$$(d_i = q_i) \rightarrow (q_i = 0 \vee v(\texttt{dto}, n)) \quad (19)$$
$$(d_i = 0 \wedge c_{\texttt{not\_empty}}) \rightarrow \neg v(\texttt{dto}, n) \quad (20)$$
$$\neg F_{\texttt{enqueue} \wedge \texttt{not\_full}} \wedge F_{\texttt{dti}} \rightarrow d_i = 0 \quad (21)$$
$$\neg F_{\texttt{enqueue} \wedge \texttt{not\_full}} \wedge T_{(\texttt{enqueue} \wedge \texttt{not\_full}) \rightarrow \texttt{dti}} \rightarrow d_i = q_i \quad (22)$$
$$F_{\texttt{enqueue} \wedge \texttt{dti}} \wedge c_{\texttt{not\_empty}} \wedge v(\texttt{dto}, n) \rightarrow T_{\texttt{not\_empty} \wedge \texttt{dto}} \quad (23)$$
$$a_{(\texttt{enqueue} \ \wedge \ \texttt{not\_full} \ \wedge \ \texttt{dti})} = a_{(\texttt{dequeue} \ \wedge \ \texttt{not\_empty} \ \wedge \ \texttt{dto})} \quad (24)$$

F. Summary

In the final SMT instance, we defined the following variables:

- q_i Integer indicating the number of packets in queue i at time n.
- f_i Integer indicating the flop output at time n.
- c_x Boolean denoting $v(x, n)$.
- d_i Integer indicating the number of packets with a certain kind of data in a queue at time n.
- T_w Boolean T_w, true iff: $v(w, t) = 1$ for $t \geq n$.
- F_w Boolean shorthand for $T_{\neg w}$.
- a_w Real: *average value* arising wherever T or F is created.

In our implementation, the first three variables can be defined for the entire network. The last four variables are defined where needed: they occur in the property we are trying to disprove, or in a property of one of the other variables. These properties are exactly the properties which have been numbered in this section. Next to these, we add invariants on q, f and d, and we add the negation of the property that all queues are live.

III. Experimental results

A. Verification Flow

Figure 3 shows the different steps of our approach. The input to our method is an RTL description of the on-chip network architecture using the Verilog hardware description language. The Verilog file is parsed and interpreted using a parser developed by Centaur Technology [8]. These files are translated to one EMOD module. During this translation, queues are identified by special tags, and their inner working is hidden as a black box. The rest of the design is identified as combinatorial logic and flops. An SMT instance is created from our definition of liveness, and from properties that hold for the combinatorial logic, flops, and queues.

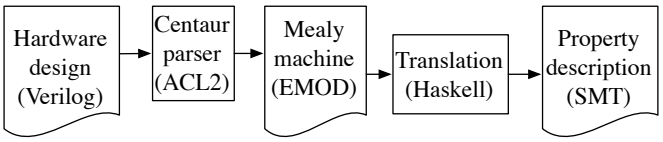


Fig. 3: Verification Flow.

The Centaur translator targets a subset of Verilog. Arithmetic and bitwise operations are supported, as well as non-blocking assignments. A (syntactic) limitation is that no blocks can be used. Transistor-level constructs, real variables, hierarchical identifiers, and multi-dimensional arrays are not supported. After this translation, all wires are assigned Boolean expressions, in which `X` and `Z` values can be regarded as input wires.

We recognise which wires are used for queues, and abstract away from the queues themselves. After a tree of modules is built, each queue in the list of those identified by the hardware designer is syntactically replaced by a function. Flops are treated similarly. Concretely, the EMOD file expresses values using AND, XOR and NOT, but also using queue specific functions. To give an example, the following value could determine whether a transfer occurs in a sequence of two queues:

```
(AND (not_empty Q1) (not_full Q2))
```

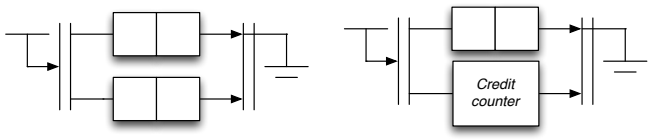
If the original RTL contained any cycles, the translation to EMOD gets rid of them, or fails.

B. Fully automated verification of various networks

We have fully automated the verification of networks using a tool in Haskell. Properties q_i, d_i, and f_i are declared in an SMT instance as integers. The averages a are declared as reals (which are actually fractions). The other variables T, F, and c become Boolean variables. This allows us to add the properties shown in Equations 1 to 24.

To verify a particular network, we start with the liveness property we wish to verify, and add the properties shown in Equations 1 to 24 when needed. By 'when needed', we mean that we do not add the network properties for everything in the network. We merely add the properties introduced by the negation of the liveness property we wish to verify, plus all variables introduced in doing so, until no new variables are introduced by the properties added. This way, our tool focuses on the cone of influence, which allows us to analyse networks with a large data size without any performance penalty.

Since we are merely adding properties to the SMT instance that hold for every run, we can be certain that if the instance yields UNSAT, we have proven the desired property. Hence our encoding is sound. Our encoding is not complete. Although our approach works for all designs we encountered in literature, we artificially constructed some networks for which our approach finds false counter-examples.



(a) Two queues. (b) A credit counter with a queue.

Fig. 4: Two layouts with parallel queues

1) Parallel queues: Figure 4 shows how a queue can be placed in parallel with something else: packets accepted from the source are put in the top queue, and something is added to the bottom component as well. On an abstract level, the two layouts are the same (assuming the credit counter is initially empty). Like the queue, the credit counter accepts tokens until it is 'full'. While full, it prevents packets from entering at the source. Similarly, the top queue is prevented from releasing its packets in case the credit counter is empty. This means that in the unreachable case that the top queue is full, and the bottom half is empty, the system is in deadlock.

These designs illustrate that the invariants added to the SMT instance prevent us from getting false positives. In the case of Figure 4a, the generated linear invariant ensures that this state is not reached. The invariant generated for that network, which suffices to prove deadlock freedom of both queues, and liveness at the input is the following:

$$q_{top} = q_{bottom}$$

For our implementations in Figure 4b, the credit counter is not abstracted as a queue. The internal state is given by the state of some flops in the credit counter. We have two implementations for a credit counter of size 3. In one implementation, we count the number of credits in a unary way (similar to a shift register). Our method finds the following invariant required to prove deadlock freedom:

$$q_{top} = f_0 + f_1 + f_2$$

In a different implementation, we use only two flops, and count binary: packets may enter if either bit is zero. Once again, our method finds the desired invariant:

$$q_{top} = f_0 + 2 \cdot f_1$$

Limitations of invariants: if we make a slight modification, and decide that the credit counter should hold at most two credits, the invariants are not strong enough to express that both flops are never simultaneously high. If we allow a packet to enter (or leave) in such a case, the invariant does not hold for that transition. In this particular case, our method does not find any invariants. A deadlock configuration is found in which the top queue is full, while the credit counter is empty. To strengthen the invariants found by our method, we suggest credit counters to be abstracted the way queues are.

2) Buffered virtual channels: Figure 5 is an example from [12]. Packets in the queue labelled as 'Buffer' are identified by a bit, such that packets originating from In1 are routed to B3, and those originating from In2 are routed to B4. The arbiter alternates between accepting packets from In1 and

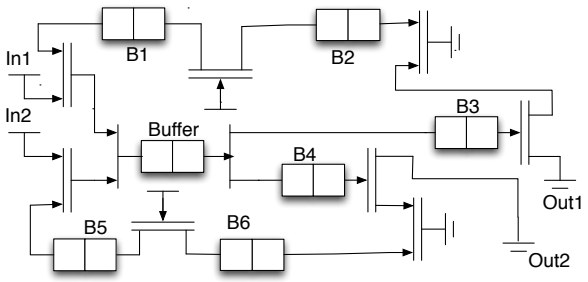


Fig. 5: Buffered virtual channels.

In2 if one is offered. It is persistent in the sense that if the 'Buffer' is full, and the packet from In1 (or In2) is offered, at the next clock tick the packet from In1 (In2) is offered to the Buffer again. The data dependent invariants corresponding to this network are found in this approach, and every buffer in the network can be proven live (provided all sources and sinks are fair).

Limitations of one-step simulation: We can modify the arbiters' behaviour such that it allows packets from In1 more often than the packets from In2. For instance, it can take two packets from In1, and then only one from In2. In this case, the properties presented so far are insufficient to verify that B5 is live (or that packets from In2 are eventually accepted). To prove this, one could add these equations to the SMT instance:

$$T_w \rightarrow v(w, n+2) \quad F_w \rightarrow \neg v(w, n+2)$$

Note that these are similar to Equations 5 and 6. In essence, these equations add a one-step simulation to our analysis. The equations above allow us to perform a two-step simulation. Unfortunately, adding these equations has a severe impact on the performance of the analysis.

3) Other networks and scalability: Next to the networks described above, we have analysed the network in Figure 5 without the queue called Buffer (with its outputs and inputs connected with a wire instead). In all cases, we proved liveness of all queues, with extra queues added at the sources and sinks (thereby verifying their liveness as well), and under the assumption of fairness of sources and sinks.

To illustrate the scalability of our approach, we took the network in Figure 5 (with the buffer), and cloned it several times, connecting the outputs to the inputs (Out1 to In1, Out2 to In2). All queues could be verified to be live. The time it took to verify this property is shown in Figure 6.

The measurements were performed using one core on a 1.8 GHz Intel Core i7-2677M[1], using Z3 as the SMT solver. A network with 25 repetitions containing 279 queues (25 repetitions of 11 queues - for of which are at sinks or sources, plus four queues for the unattached sinks and sources) is analysed in 25 seconds. When investigating a network with 26 repetitions, we aborted the execution after 10 minutes. We consider 280 queues to be a rough limit for networks with medium complexity, at which the current SMT solvers tend to fail for solving the instances.

[1]Using one core increases the base frequency to 2.9GHz

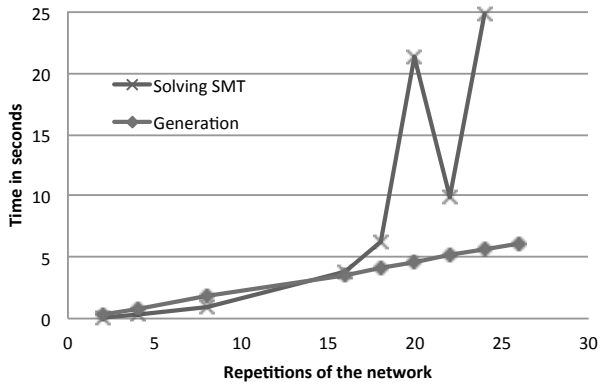


Fig. 6: Execution times for cascaded buffered virtual channels.

IV. Related Work and Discussion

Recent advancements in progress verification for communication fabrics are based on micro-architectural models. Chatterjee *et al.* [4] introduced a language – called xMAS – for the description of executable specifications of micro-architectures. This language is restricted to eight primitives with well-defined semantics. Chatterjee and Kishinevsky [3] demonstrated how inductive invariants can be automatically derived from an xMAS model. Such invariants are then used to improve hardware model-checking of safety properties and deadlock freedom [7]. Ray and Brayton [12] proposed a semi-automatic approach dedicated to xMAS models of credit-based flow control systems. They manually add buffer relations as invariants to prune the search. In all these works, a high-level xMAS model is required. An implicit assumption is that the invariants derived from the xMAS model must match with the RTL description. There exist useful design components which cannot be expressed using the 8 primitives of the xMAS language [14]. Even if our method imposes some restriction on the input design, it does not necessitate the existence of an xMAS model and directly applies to RTL designs.

The examples mentioned so far are rather small network components. Verbeek and Schmaltz [13] developed a deadlock detection algorithm for xMAS models and applied it to the verification of large networks. Their work applies directly to the xMAS level. The question of translating their results obtained on xMAS models to RTL designs is still open.

V. Conclusions and Future Work

We presented a novel verification method for liveness properties of RTL designs of on-chip communication networks. Our method abstracts away from queue implementations and timing details. Liveness properties are expressed using the average values of wires along lasso runs. Properties together with a representation of the network are translated to an SMT instance. Experimental results demonstrates the scalability of our approach to fabrics with hundreds of queues. It is sound but incomplete. Our method fails on some artificial examples.

We see two possible ways of improving our method. First, our way of relating T and F values via averages may be less efficient than translating the corresponding properties to functionally reduced AIGs, and then inspecting their structure [11]. This may have a positive influence on the scalability of our approach. Second, instead of writing a large SMT instance, we could try applying the ideas of our approach to an existing bounded model checker. This may speed up the model checker, or put differently, make our method complete (instead of just sound) without imposing a large performance penalty on the networks we could already handle.

Acknowledgments

We thank the anonymous reviewers for their apposite, constructive, and detailed comments. This research is supported by NWO project Effective Layered Verification of Networks on Chips (ELVeN) under grant no. 612.001.108.

References

[1] L. Benini and G. De Micheli. Networks on Chips: a new SoC paradigm. *IEEE Computer*, 35(1):70–78, January 2002.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[3] S. Chatterjee and M. Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In T. Touili, B. Cook, and P. Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*. Springer, July 2010.

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

[5] W. J. Dally. The end of denial architecture and the rise of throughput computing. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages xv–xv. IEEE, 2009.

[6] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of Design Automation Conference (DAC'01)*, pages 684–689, 2001.

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

[8] W. A. J. Hunt and S. Swords. Centaur technology media unit verification. In *Computer Aided Verification*, pages 353–367, 2009.

[9] S. J. C. Joosten and J. Schmaltz. 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*, pages 57–64. IEEE, 2013.

[10] S.-I. Minato and F. Somenzi. Arithmetic boolean expression manipulator using bdds. *Formal Methods in System Design*, 10(2-3):221–242, 1997.

[11] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. Fraigs: A unifying representation for logic synthesis and verification. Technical report, ERL Technical Report, 2005.

[12] S. Ray and R. K. Brayton. Scalable progress verification in credit-based flow-control systems. In W. Rosenstiel and L. Thiele, editors, *DATE*, pages 905–910. IEEE, 2012.

[13] 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*, FMCAD '11, pages 223–231, Austin, TX, 2011.

[14] F. Verbeek and J. Schmaltz. Automatic generation of deadlock detection algorithms for a family of micro architectural description languages. *IEEE International High Level Design Validation and Test Workshop (HLDVT'12)*, November 2012.