# Generation of Inductive Invariants from Register Transfer Level Designs of Communication Fabrics

Sebastiaan J.C. Joosten
School of Computer Science
Open University of the Netherlands
sjj@ou.nl

Julien Schmaltz
School of Computer Science
Open University of the Netherlands
julien.schmaltz@ou.nl

***Abstract*—Communication fabrics constitute a key component of multicore processors and systems-on-chip. To ensure correctness of communication fabrics, formal methods such as model checking are essential. Due to the large number of buffers and the distributed character of control, applying these methods is challenging. Recent advancements in the verification of communication fabrics have demonstrated that the use of inductive invariants provides promising results towards scalable verification of Register Transfer Level (RTL) designs. In particular, these invariants are key in the verification of progress properties. Such invariants are difficult to infer. So far, they were either manually or automatically derived from a high-level description of the design. Important limitations of these approaches are the need for the high-level model and the necessary match between the model and the RTL design. We propose an algorithm to automatically derive these invariants directly from the RTL design. We consider communication fabrics to be a set of message storing elements (e.g. buffers) and some routing logic in-between. The only input required by our method is a definition of when messages enter or leave a buffer. We then exploit this well-defined buffer interface to automatically derive invariants about the number of packets stored in buffers. For several previously published examples, we automatically generate the exact same invariants that were either manually or automatically derived from a high-level model. Experimental results show that the time needed to generate invariants is a few seconds even for large examples.**



## I. Introduction

To gain performance out of more transistors, a major trend in computing is to go parallel [1]. Multi-core processors and multi-processors Systems-on-Chip (MPSoC's) integrate a rapidly growing number of processing and memory units. To cope with the complexity of the integration of all these cores, Networks-on-Chip (NoC's) [2], [3] – also called communication fabrics – emerged as efficient architectures to support on-chip communications between a large number of cores.

NoC's or communication fabrics architectures typically consist of a large number of buffers, which are storing messages at their intermediate hops towards their destination. Control decisions are distributed among these intermediate nodes. The large number of buffers creates a large state-space and distributed control makes the application of abstraction techniques – e.g., localisation – difficult [4]. The application of formal methods to this class of designs constitutes an important challenge.

Recent advancements in the verification of communication fabrics rely on using invariants to speed-up hardware model-checking [5], [6], [4]. For instance, these invariants are key to rule-out false deadlocks or other spurious counter-examples in progress verification. Ray and Brayton [4] write:

> The fact that buffer relations are making proofs one-step inductive for many designs is the key factor that makes our approach scalable.

By 'buffer relations' the authors refer to invariants:

> These seemingly intuitive relations are quite non-trivial to mine from a bit-level implementation of a fabric, unless they are explicitly hinted by the architects.

In the aforementioned studies, invariants are either added manually or automatically extracted from high-level representations of the fabrics. Adding invariants manually is not scalable. Using high-level models require these high-level models to exist and – more importantly – these high-level models must match the Register Transfer Level (RTL) designs. Such abstract models are not always available or easy to create. They are often slightly different from the actual RTL designs due to low-level optimizations.

Our contribution is a new method to automatically extract such inductive invariants directly from the *bit-level RTL* design. Our method is fully automatic and scale to large fabrics. Its main restriction is to require a well-defined interface for message storing elements, e.g. buffers. This interface allows us to decompose designs into a set of buffers and a set of registers (flip-flops) all interconnected by combinatorial logic. The buffer interface precisely defines when a packet enters and when a packet leaves a buffer. From this definition, invariants express the number of packets stored in a buffer as the difference between the number of packets, which have entered a buffer and the number of packet, which have left it. The main components of our approach are a function computing the number of times a wire has been high until the current time and the translation of the usual Boolean connectives to expressions over this function.

## II. Method

To obtain invariants, we focus on the interfaces of buffers. We focus on buffers since they are essential in communication

fabrics, easy to identify, and allow us to compare our results with invariants that have been added manually. We require designers to indicate which Verilog modules are buffers. For each buffer, we generate equalities about the number of packets in that buffer. The expression for the number of packets in a buffer is translated into sums of linearly independent variables. As these linearly independent variables are equal to the number of packets in a buffer, performing Gaussian elimination on this translation gives the desired invariants.

After showing an example of an interface, we give a high-level description of the invariants we generate. We show in Section II-A that it sufficies to know equalities about a cumulative function called $s$. In Section II-D, we zoom in on the properties of $s$ essential to obtaining these equalities. We finish with an overview of the algorithm in Section II-E.

### A. A simple example

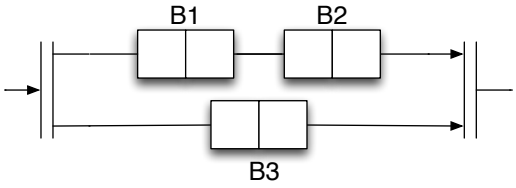


Fig. 1. A very simple network

Figure 1 shows a micro-architectural model of a simple network. It is a network with one input and one output. At the input, a *fork* element duplicates packets, which are offered to buffers B1 and B3. Buffer B1 is connected to B2, and the packets in buffers B2 and B3 are combined at the output by a *join* element. From the RTL implementation of this network, we automatically derive the following invariant:

$$num(\mathrm{B1}) + num(\mathrm{B2}) = num(\mathrm{B3})$$

Such an invariant seems obvious from the micro-architectural model. It can indeed be automatically inferred from such a model [5]. The challenge is that this nice architectural structure is not directly available from the RTL design. Extracting this invariant directly from the hardware design is much more difficult [4].

The basic idea of our method is to define variables $enter(x)$ and $exit(x)$. Variable $enter(x)$ represents the number of packets that have entered buffer $x$. Variable $exit(x)$ represents the number of packets that have left buffer $x$. We assume that at time 0, there is a reset and all buffers are empty. Therefore, the number of packets currently in buffer $x$ up to some global time $N$ is expressed as the difference between $enter(x)$ and $exit(x)$.

Regarding the example in Figure 1, we can express the following equalities:

$$num(\mathrm{B1}) = enter(\mathrm{B1}) - exit(\mathrm{B1})$$
$$num(\mathrm{B2}) = enter(\mathrm{B2}) - exit(\mathrm{B2})$$
$$num(\mathrm{B3}) = enter(\mathrm{B3}) - exit(\mathrm{B3})$$

The three equalities above are simplified by performing a translation of $enter(x)$ and $exit(x)$. After this translation, we perform a Gaussian elimination to eliminate all variables except for $num(x)$. A key aspect of the translation is to identify the following equalities:

$$enter(\mathrm{B1}) = enter(\mathrm{B3})$$
$$enter(\mathrm{B2}) = exit(\mathrm{B1})$$
$$exit(\mathrm{B3}) = exit(\mathrm{B2})$$

These equalities play a central role in the derivation of the invariants.

### B. Well-defined interfaces

To fully express $enter(x)$ and $exit(x)$, we use global time $N$ to count the number of events when a packet is entering or leaving buffer $x$. This implies that for every buffer an expression can be defined to indicate when a packet is entering, and another one to indicate when one is leaving a buffer. Providing these expressions is the only manual requirement for the RTL designer. The rest of our method is fully automatic.

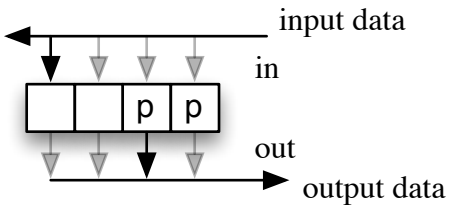


Fig. 2. A four place circular buffer

As an example, we consider a 4-place circular buffer, where each place is 4 bits wide (Figure 2). Note that our method is fully transparent to the number of places, and almost transparent to their width (we only look at the bits that are used for routing). This buffer has a two bit register `in` to keep track of where the packet entering next should be placed. It has a two bit register `out` to keep track of which packet is offered at the output side.

A transfer occurs when both input-ready and output-ready wires are high. The interface of this buffer is composed of an input-ready and an output-ready wire, and a number of data wires. An input-ready wire indicates that there is a packet ready to be transmitted (on the data wires). An output-ready wire indicates that a buffer is ready to receive.

Expression $\mathrm{in.irdy} \wedge \mathrm{in.trdy}$ defines the event of a packet entering a buffer. Note that in our buffer $\mathrm{in.trdy}$ equals $\neg$`full`. Expression $\mathrm{out.irdy} \wedge \mathrm{out.trdy}$ defines the event of a packet leaving a buffer. Variables $enter(x)$ and $exit(x)$ are expressed in terms of an event counter – noted function $s$ – as follows:

$$enter(x) = s(x.\mathrm{in.irdy} \wedge x.\mathrm{in.trdy}, N)$$
$$exit(x) = s(x.\mathrm{out.irdy} \wedge x.\mathrm{out.trdy}, N)$$

Every time a packet is injected (input is ready and output is ready), `in` is raised by one. When a packet is taken (output is ready and output is ready), `out` is raised by one. When the two are equal, the buffer is either full or empty. To distinguish between the two cases, a one bit register `full` is set to high if `in` is raised to be the value of `out` while no packet is taken. On the input, `trdy` is defined as the negation of `full`. If output-ready is low, no packet enters the buffer, even if a packet is taken from the turn of that cycle (so latency increases if the full capacity of the buffer is used).

Our method requires the definition of these events for all buffers of the design. Note that these definitions may not be the same for all buffers. In our experimental results, these are the expressions and the buffer implementation we used.

### C. Semantics of function $s$

The bit-level structure of the network is known in terms of values such as input wires and buffer interface wires, and gates such as AND, OR, XOR and NOT. The value of a wire $w$ at time $t$ is given by function $v(w,t)$, which returns 0 for false (low) and 1 for true (high). Time is indicated by natural numbers (starting at time 0). To obtain the number of 1 values of some wire $w$ up to some time $t$, we can accumulate $v$ to obtain function $s$:

$$s(w,t) = \sum_{i=0}^{t-1} v(w,i)$$

Keeping these semantics of $s$ in mind, allows the reader to verify the soundness of the translation of $s$.

### D. Translation of function $s$

In this section, we translate $s(x,N)$ into a set of variables. We show in Theorem 1 (see next sub-section) that the variables used at the end of the translation are linearly independent. Taking a variable for every occurence $s(x,N)$ does not suffice to obtain the correct inequalities, since it would not identify $s(x,N) = s(\neg\neg x,N)$ by itself. That is: $s(x,N)$ and $s(\neg\neg x,N)$ are still linearly dependent. The proposed translation does identify equivalent terms.

We first focus on properties of $v$. The reader may verify that:

$$\begin{aligned} v(0,t) &= 0 \\ v(\neg x,t) &= v(1,t) - v(x,t) \\ v(x \vee y,t) &= v(x,t) + v(y,t) - v(x \wedge y,t) \\ v(x \text{ XOR } y,t) &= v(x,t) + v(y,t) - 2 \cdot v(x \wedge y,t) \end{aligned}$$

Note that we did not write 1 for $v(1,t)$. For now, the reader may consider this as a slightly strange notation. To translate AND gates (or $\wedge$), we introduce operation $\otimes$. This operation is like a product and the following equations hold:

$$\begin{aligned} v(x \wedge y,t) &= v(x,t) \otimes v(y,t) \\ (a+b) \otimes c &= a \otimes c + b \otimes c \\ c \otimes (a+b) &= a \otimes c + b \otimes c \\ 0 \otimes b &= 0 \\ b \otimes 0 &= 0 \end{aligned}$$

Note that these equalities applied from left to right will translate any Boolean expression into a sum in which every term is a constant times a list $v(a_0,t) \otimes \cdots \otimes v(a_n,t)$. Such lists separated by $\otimes$ can be thought of as sets of constants (with no duplicates, since $v(x,t) \otimes v(x,t) = v(x,t)$). Suppose $x$ is some expression translated this way:

$$v(x,t) = \alpha v(a_0 \wedge a_1 \wedge \ldots, t) + \beta v(b_0 \wedge b_1 \wedge \ldots, t) + \cdots$$

By summing over all values of $t$ from 0 to $N$, we obtain:

$$s(x,N) = \alpha s(a_0 \wedge a_1 \wedge \ldots, N) + \beta s(b_0 \wedge b_1 \wedge \ldots, N) + \cdots$$

Hence, for every remaining term $s(c,N)$, $c$ is a conjunct of independent wires. We refer to these final terms as *primitives*. These primitives are the variables in our final Gaussian elimination.

Instead of writing $s(\ldots,N)$, we write the corresponding set of independent variables:

$$s(x,N) = \alpha\{a_0,a_1,\ldots\} + \beta\{b_0,b_1,\ldots\} + \cdots$$

Note that the above is a linear equality, with sets in the places where you would normally expect the variables. The empty set will correspond to value $N$.

We can apply this translation directly on $s$. Where previously $\otimes$ could be thought of as a product, it does not have any semantics on $s$. Therefore the operation $\otimes$ must be read as a purely syntactic intermediary. Despite its lack of semantics, the translation is still sound due to its counterpart on $v$.

Our rewrite system is as follows:

$$\begin{aligned} s(0,N) &= 0 \\ s(1,N) &= \{\} \\ s(\neg x,N) &= \{\} - s(x,N) \\ s(x \vee y,N) &= s(x,N) + s(y,N) - s(x,N) \otimes (y,N) \\ s(x \text{ XOR } y,N) &= s(x,N) + s(y,N) - 2 \cdot s(x,N) \otimes (y,N) \\ s(x \wedge y,N) &= s(x,N) \otimes s(y,N) \\ (a+b) \otimes c &= a \otimes c + b \otimes c \\ c \otimes (a+b) &= a \otimes c + b \otimes c \end{aligned} \tag{1}$$

When we are done rewriting in this way, we can get rid of $\otimes$ by interpreting $s(x,N)$ as $1 \cdot \{x\}$ and applying:

$$c_1 \cdot S \otimes c_2 \cdot T = c_1 \cdot c_2 \cdot (S \cup T)$$

To summarize our approach, let's look at a small example, the translation of the term $\neg(\neg x \wedge \neg y)$:

$$\begin{aligned} s(\neg(\neg x \wedge \neg y),N) &= 1 \cdot \{\} - s(\neg x \wedge \neg y,N) \\ &= \{\} - (\{\} - \{x\}) \otimes (\{\} - \{y\}) \end{aligned}$$

Since $\{\} \cup \{\} = \{\}$, $\{\} \cup \{x\} = \{x\}$ and so forth:

$$\begin{aligned} &= \{\} - \{\} + \{y\} + \{x\} - \{x, y\} \\ &= \{y\} + \{x\} - \{x, y\} \end{aligned}$$

From the above, one can verify that the translation correctly identifies $s(x \vee y, N)$ with its De Morgan dual.

The translation given by our rewrite system (1) is exponential in terms of logic depth, which is bounded in most practical applications.

### *E. An algorithm for finding inductive invariants*

Rewrite system (1) allows us to express $enter(x)$ and $exit(x)$. We can, in fact, define $num(x)$ directly:

$$num(x) = enter(x) - exit(x)$$

This yields one equation per buffer. Note that $num(x)$ denotes the number of packets in a buffer *at* time $N$, while $enter(x)$ and $exit(x)$ denote the number of enter and exit events *up to* (not including) time $N$. It may seem counter-intuitive, but this translation, which yields many variables and few equations, generates all possible inductive invariants that can be expressed in terms of $num(x)$. This is the statement of the key theorem below.

*Theorem 1:* In a network with fully independent wires, the system of equations obtained by translating $num(x) = enter(x) - exit(x)$ for every buffer, implies all linear equalities in terms of $num(x)$.

*Proof:* We prove the Theorem by contradiction. Assume there is a linear equality in terms of $num(x)$, which is not implied from the system of equations. Say this equality is $a_0 num(x_0) + a_1 num(x_1) + \cdots = v$. Here $v$ is a constant, which we can argue to be equal to a factor of $N$: if not, take an assignment of wires up to time $N$, and repeat the same assignment. Now sum to $2N$, and the expression (in terms of the original $num()$ values) reads: $2a_0 num(x_0) + 2a_1 num(x_1) + \cdots = v'$, so $v' = 2v$. The only constant for which this happens is $N$ (or 0).[1]

Replacing the occurrences of $num(x)$ with the definitions yields the equation of primitives: $b_0 \cdot c_0 + b_1 \cdot c_1 + \cdots = 0$ (with $b_i \neq 0$). Here $c_0$ are conjuncts (and possibly $N$ for the empty set). This equation is not implied from the original system of equations, so we know that the left hand side of this equation is not empty. (If this translated to $0 = 0$, the original equality would be implied.) Now we derive a contradiction to $b_i \neq 0$, assuming independent wires.

Let $c_i$ be a smallest conjunct, that is, for no $j$ every $c_j \subset c_i$. Now let $N = 0$, and assign a value of 1 to all wires in $c_i$ at $t = 0$, while assigning 0 to all other wires. This implies that $c_i = 1$ and $c_j = 0$ for $j \neq i$. Filling in these assignments of $c$ into our equation of primitives gives us: $b_i 1 = 0$, so $b_i = 0$, which is a contradiction. ∎

[1]Here we consider $N$ to be a constant. If you don't, $v = 0$ is the only remaining equality, since $v = N$ would not be a linear equality, and the proof still works.

The main limitation of this theorem lies in the assumption that the analysed network consists of fully independent wires. Hardware typically has a state, and even for the buffers, the number of packets that can leave typically depends on the history. To give an artificial example, consider two buffers that never receive any input, say B1 and B2. From B1, we accept all available packets, while from B2, we do not. Our analysis derives $num(\mathrm{B2}) = 0$, but it will not derive $num(\mathrm{B1}) = 0$ or $num(\mathrm{B1}) = num(\mathrm{B2})$ because it does not recognise that no packets could ever leave B1. In practice, however, such situations are rare to communication fabrics. In Section III-D, we present an example of this which arises for buffers with data dependencies.

Using the previous observations, our algorithm can be summarized as follows:

1) For every buffer, the hardware designer gives an expression indicating when a packet enters, and when one leaves.
2) Generate the system of equations with an equation for every buffer $x$:

   $$num(x) = enter(x) - exit(x)$$

   Translate $enter(x)$ and $exit(x)$ into primitives according to Equations (1)
3) Perform Gaussian elimination to obtain all equalities between $num(x)$.

### *F. Data dependent buffers*

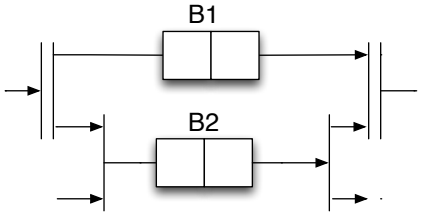


Fig. 3. A network with message dependency in B2

We are also interested in the number of packets of a certain type. To see when this might be useful, consider Figure 3. In this figure, buffer B2 is preceded by an arbiter, and has a switch at its output. We call the packets that are switched upwards to be of 'type A'. The other packets are of 'type B'. Depending on the particular configuration of this network, B2 might contain as many packets of type A as the number of packets in buffer B1. Such an invariant could be necessary to prove progress of this network.

We handle these invariants by adding a different kind of buffer:

$$\begin{aligned} num_A(x) &= enter_A(x) - exit_A(x) \\ enter_A(x) &= s(x.\mathrm{in.irdy} \wedge x.\mathrm{in.trdy} \wedge A, N) \\ exit_A(x) &= s(x.\mathrm{out.irdy} \wedge x.\mathrm{out.trdy} \wedge A, N) \end{aligned}$$

Note that the above is only one equation: $enter_A(x)$ and $exit_A(x)$ are only there for readability. It is not necessary to

add this equation of every possible type $A$ and every buffer $x$. After translating all terms, we inspect every conjunct occurring in this translation. In the case that data wires of the output of B$i$ occur in a conjunct, we add the equation above, with $A$ defined as the conjunction of all the data wires of the output of B$i$. Adding this equation causes a translation of $s$ again, which may introduce new conjuncts. For this reason, we iterate the process until we reach a fixed-point.

## III. Experimental results

All software and examples used in this section are available at the following address:

`http://genoc.cs.ru.nl/research/memoCODE/`

For our experimental results, we draw on examples from papers that prove liveness through model checking. In all of these papers, inductive invariants were a necessity, and were added manually, or through inspection of a high-level description of the same model. This section shows that we are able to deduce the inductive invariants automatically.

### A. Tool flow

We created Verilog modules for the networks by interpreting the figures shown here. Using ACL2 code from Centaur Technology [7] and a bit of custom code, these models are bit-blasted while preserving buffer information. The resulting structure is called an E-module. This translation step is such that the E-module does not contain any cyclic dependencies (it generates flops, X values, or possibly even errors in case the RTL does contain them). The E-module is parsed and analysed using a Haskell implementation of the method described here. Where execution times are stated, these times include parsing the bit blasted E-module and returning the invariants. The time needed to parse Verilog input files is not included.

In case the reader wants to try these experiments himself, we have made the E-modules and our Haskell implementation available from the previously mentioned website. For readers who would like more details on the networks, we also included the original Verilog modules.

### B. Virtual channels with buffer

From the network described in Figure 1, we obtained exactly the desired invariant. We tested our approach on various credit-flow systems from [4]. Figure 4 shows the configuration of a buffered virtual channel. Our in- and outputs are modeled as a buffer with X values at the unused end, so this design has 11 buffers. The buffer labeled as 'Buffer' can contain two types of packets: those for B3 and those for B6. The last bit of the packet data is used to indicate this. The resulting invariants were:

```
B1 + buffer + B3 = B0 + buffer_[3]
B4 + B5 + buffer_[3] = B6
```

The prefix `_[3]` indicates the number of packets in which bit 3 (the last bit) was set. In the first invariant, `buffer - buffer_[3]` can be read as: packets in which bit 3 was not set. These two invariants were exactly the invariants required in the analysis in [4]. Generating these invariants took 0.03 seconds on one 1.8 GHz i7 core for our implementation.

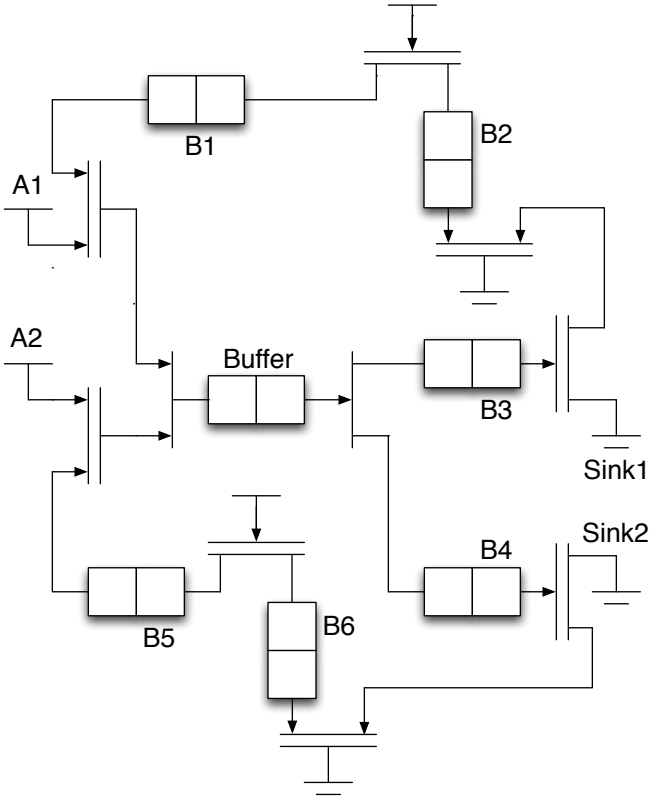


Fig. 4. Virtual channels and a message dependent buffer

If we replace the buffer with a wire, we get a model of a virtual channel without any data dependency. Here we get exactly the invariants as in [4] again.

### C. Scalability of the approach

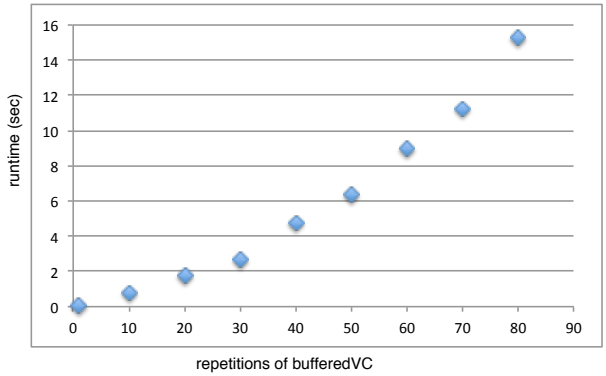


Fig. 5. Runtime for repetitions of Figure 4

To test the performance of our method, we repeated the configuration of Figure 4 several times by connecting the

outputs to the inputs of another design. The result is shown in Figure 5. In this case, each repetition of the virtual channels contains 11 buffers, so the 80 repetition example has 880 buffers. It generated 160 invariants. Also, the invariants were generated as two local invariants per repetition (as we would expect), and not as a linear combination of those invariants.

### D. Scoreboard

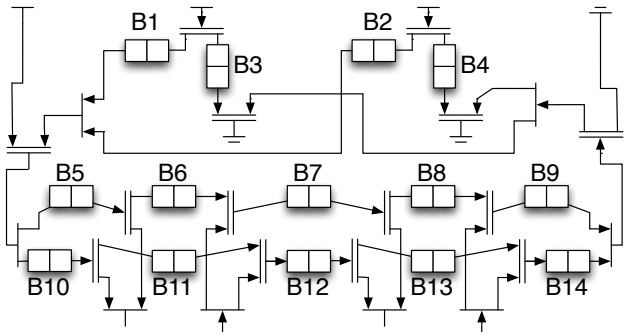


Fig. 6. A 2-entry scoreboard

The two-entry scoreboard in [4] is shown in Figure 6. In this example, the top right switch causes B9 and B14 to have a data dependency. Because of this, all buffers B5 to B14 have data dependencies. Analysing the network took 0.195 seconds. The invariants found were slightly different from the invariant the authors added:

```
B5 + B10 + B6 + B11 + B7 + B12 + B8
   + B13 + B9 + B14 + B2 + B1 = B4 + B3
B1 + B14_[3] + B9_[3] + B13_[3] + B8_[3]
   + B12_[3] + B7_[3] + B11_[3] + B6_[3]
   + B10_[3] + B5_[3] = B3
```

Note that for our network, B10 does not receive any packets with bit `[3]` set. For this reason, we can deduce `B10_[3]=0` Applying this argument iteratively, we obtain `B11_[3]=B12_[3]=B13_[3]=B14_[3]=0` On the other side, we know: `B5_[3]=B5`, and apply this argument to the next buffers. These two arguments suffice to get the same two invariants as used in [4], namely:

```
B10 + B11 + B12 + B13 + B14 + B2 = B4
B1 + B9 + B8 + B7 + B6 + B5 = B3
```

The above invariant would become invalid if we simulate the network from an (unreachable) state in which `B10_[3]=B10=1` This implies that the invariant (which the authors of the paper found by 'manual inspection') is not one step inductive by itself for our implementation. The Verilog implementation used in [4] was not available to us, nor did they publish sufficient code to allow us to replicate their results. The authors mentioned that their one step induction technique failed for this scoreboard. Unfortunately, we could not check whether the invariant we found would give an improvement.

An alternative way of obtaining the above invariant is to change Buffers 5 to 14 such that they are three bits wide instead of four. In this case, no distinction can be made on the last bit `_[3]`.

Even though our method did not find the same equations for the scoreboard, the number of equations found is equal, as well as the buffers occurring in them.

## IV. Discussion

### A. Related Work

Our work closely relates to recent research in the verification of communication fabrics. These studies proposed the use of invariants generated from micro-architectural models to ease safety and liveness verification. Chatterjee *et al.* [5] 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 [5] demonstrated how inductive invariants can be automatically derived from an xMAS model. Such invariants can then be used to improve hardware model-checking of safety properties. Chatterjee *et al.* [6] extended the technique to the verification of deadlock freedom. In this line of work, invariants are automatically generated. Ray and Brayton [4] 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 used to derive properties which are then used for hardware verification. An implicit assumption is that the invariants derived from the xMAS model must match with the RTL description. There is the implicit requirement that the RTL code is strongly related to the xMAS model. Also, there exist useful design components which cannot be expressed using the 8 primitives of the xMAS language [8]. Our method directly applies to the RTL design, removing the need for the high-level model. It is also not limited to the eight primitive of the xMAS language. It accepts any combinatorial logic between buffers and registers to store the states of arbiters. Note that our method automatically derives the exact same invariants mentioned in the aforementioned references.

These examples are rather small network components. Verbeek and Schmaltz [9] developed a deadlock detection algorithm for xMAS models and applied it to the verification of large networks. Their techniques scale to much larger designs, but is only applicable to xMAS models.

### B. Micro-architectural abstraction

A great potential of our technique is to leverage this scalable approach to RTL designs. Indeed, variables $enter()$ and $exit()$ can be used to extract a micro-architectural model from a design. We turn our attention to Figure 4 again. We may derive our invariants in terms of $enter(x)$ and $exit(x)$, instead of $num(x)$. This allows us to reconstruct the routing logic between buffers on a more abstract level, and signal subtleties such as unintended packet loss. Performing our analysis for the $enter(x)$ and $exit(x)$ values on Figure 4 yields:

```
A1EXIT + A2EXIT = BufferENTR
B3ENTR + B4ENTR = BufferEXIT
B1EXIT = A1EXIT
B4EXIT = sink2ENTR
B2EXIT = TokenSinkENTR
sink1ENTR = B2EXIT
B2ENTR = TokenSrc1EXIT
sink2ENTR = B6EXIT
B6EXIT = TokenSink2ENTR
B5ENTR = B6ENTR
B6ENTR = TokenSrc2EXIT
B1ENTR = B2ENTR
A2EXIT = BuffENTR_[3]
B3EXIT = sink1ENTR
BuffEXIT_[3] = B4ENTR
B5EXIT = A2EXIT
```

From these equations, it seems possible to extract a more high-level structure of the network. Equalities such as `B1EXIT = A1EXIT` would indicate *fork* and *join* elements, while additions such as `A1EXIT + A2EXIT = BuffENTR` would indicate the presence of switches and arbiters. High-level analysis algorithms such as those of Verbeek and Schmaltz [9] could then be applied to this abstraction.

### C. Dealing with registers

The current value $f$ of a flop can be expressed as:

$$\begin{aligned} v(f,N) &= s(f,N+1) - s(f,N) \\ &= s(d,N) - s(f,N) \end{aligned}$$

By $d$ we mean the expression driving the flop, while $f$ is the flop value, so our substitution $s(f,N+1) = s(d,N)$ holds for flops which are initially zero. Adding $cur(x) = s(d(x),N) - s(f(x),N)$ to our analysis can help to deal with things such as credit counters. Future research could investigate how to do this. Currently, we are able to craft RTL designs such that they yield invariants about flops, but this imposes severe restrictions on the RTL design.

## V. Conclusions

We presented a fully automatic method to derive inductive invariants from RTL designs of communication fabrics. Our method requires a precise definition of the events of entering or leaving a buffer. From this definition, the remaining analysis is fully automatic. Experimental results show that we automatically derive roughly the same invariants that were previously either manually or automatically extracted from high-level models (the exact same invariants in most examples). The time to generate the invariants is a few seconds even for large examples. Recent advancements in the verification of communication fabrics all required high-level models. Our approach leverages these advancements to regular RTL designs. This opens up the possibility of studying the scalability of the combination of our invariant generation technique with hardware model-checking techniques for arbitrary RTL designs.

## References


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

[2] L. Benini and G. De Micheli, "Networks on Chips: a new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, January 2002.

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

[4] 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.

[5] S. Chatterjee, M. Kishinevsky, and Ü. Y. Ogras, "Quick formal modeling of communication fabrics to enable verification," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'10)*, 2010, pp. 42–49.

[6] 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.

[7] W. A. J. Hunt and S. Swords, "Centaur technology media unit verification." in *Computer Aided Verification*, 2009, pp. 353–367.

[8] 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.

[9] ——, "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.