WickedXmas: Designing and Verifying on-chip Communication Fabrics

Sebastiaan J.C. Joosten
Eindhoven University of Technology
Email: s.j.c.joosten@tue.nl

Freek Verbeek
Open University of The Netherlands
Radboud University Nijmegen
Email: freek.verbeek@ou.nl

Julien Schmaltz
Eindhoven University of Technology
Email: j.schmaltz@tue.nl

***Abstract*—In modern chip architectures, the increase in parallelisation brings about highly complex on-chip communication fabrics. We present WickedXmas, a tool that facilitates the design and formal verification of such interconnects. The tool is based on the language xMAS, which is a high level design language for communication fabrics, originally proposed by Intel. The use of xMAS ensures that many common modelling errors such as unintended loss of data or dangling wires are prevented by construction. Therefore, the major challenge in verifying xMAS models is establishing deadlock freedom. WickedXmas can automatically detect deadlocks or prove their absence. If a deadlock is found, it is presented to the user for further analysis. Experimental evaluation on a range of interconnects shows good performance and scalability of WickedXmas in contrast to verification from scratch, or using existing model checking techniques. Using WickedXmas, a user can draw a communication fabric and formally verify it automatically.**

***Keywords*—Communication Fabrics, Deadlock Detection, Formal Verification Tools**

I. Introduction

The field of computing is experiencing a major change. Improvements in integration capabilities bring more transistors on a piece of silicon. As power dissipation prevents these transistors from running faster, the current trend to gain performance out of these additional transistors is to go parallel [9]. Massively parallel architectures are slowly becoming available to the masses. In this realm of multi-core computing, the communication fabrics connecting the cores become a key component for design and validation. Ensuring correctness of these on-chip networks is crucial to system correctness.

Communication fabrics are complex structures. To achieve performance and to support simultaneous transactions between agents, complex credit-counting mechanisms, virtual channels, or flow-control mechanisms are needed. The verification of communication fabrics is a challenge. Distributed control makes abstraction difficult and the large number of queues needed to store messages at their intermediate hops induces a large state-space. To ease design and verification, researchers at Intel recently proposed xMAS – for eXecutable Micro-Architectural Specifications – as a language to represent micro-architectural models of communication fabrics [6], [7]. This language captures high-level descriptions for designers' intents, easing the understanding of complex designs. From this high-level model, invariants can be derived and input to a hardware model-checker, e.g. ABC [3], to speed-up verification.

In this paper, we present WickedXmas, a toolset for the design and analysis of communication fabrics[1]. Our tool is publicly available and provides designers with a graphical user interface. WickedXmas integrates several analysis techniques, in particular for invariant generation and deadlock verification. Some of these techniques were previously published, but not publicly available. It implements the invariant generation method proposed by Intel [5] and an efficient deadlock detection algorithm developed by Verbeek [14]. Invariants are written in separate files in the SMT format [2]. These invariants can therefore easily be used by related techniques.

A key feature of WickedXmas is *compositional* design. A user can draw subnetworks or design common structures such as scoreboards, virtual channels, processing nodes or routers. WickedXmas enables the user to *parameterize* such designs and save them in such a way that they can be reused to build larger fabrics. Parameters can influence among others the types of packets that are injected, the routing behavior of the switches and even the structure of the subnetwork. This facilitates, e.g., the design of a network in which certain nodes are masters and others are slaves, or the design of a processing node that transmits packets only to a restricted set of destinations. Most importantly, it facilitates the design of *recursive* structures. We illustrate this capability with the design of a Spidergon ring [8]. We provide experimental results on two network elements previously studied in several papers. These show that WickedXmas is able to detect routing deadlocks, message-dependent deadlocks, on-chip synchronization deadlocks, and incorrectly sized credit counters.

Other than WickedXmas, there is no publicly available tool that supports xMAS or a similar high-level design language. Poliakov et al. have developed Workcraft, which is a tool that can model and detect deadlocks in a family of concurrent system design languages such as Petri Nets or gate-level circuits [12]. Workcraft, however, cannot directly be used for design and verification of xMAS.

The next section discusses background theory on xMAS and communication fabric deadlocks. Section III presents the architecture of WickedXmas. We explain the usage of WickedXmas in Section IV. Some non-trivial case studies are presented in Section V. We conclude in Section VI.

[1] WickedXmas and the examples in this paper are available at: http://www.cs.ru.nl/~freekver/algo_xmas/index.html

II. Background Theory

The xMAS language can be used to model and verify on-chip communication networks at the microarchitectural level. Consisting of only eight primitives, the language is expressive enough to model many different aspects of communication fabrics, such as message dependencies, virtual channels, cache coherency protocols, master/slave protocols, credit-based flow controls, and broadcasting. We introduce the eight primitives of xMAS and their semantics. We illustrate how xMAS is typically used and how WickedXmas improves this methodology.

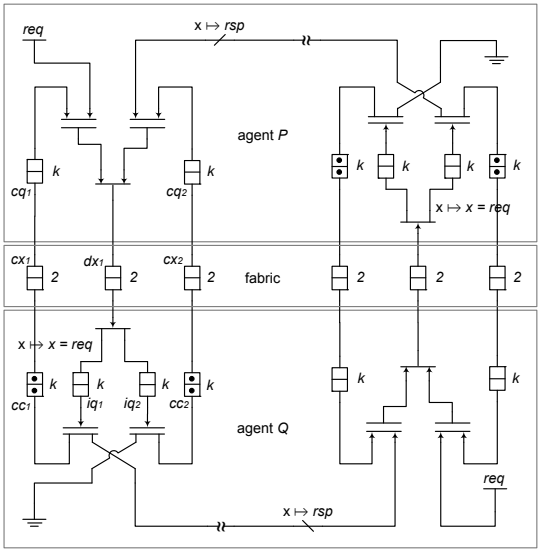
A. The xMAS language

An xMAS model is a network of primitives connected via channels. A channel is connected to an *initiator* and a *target*. Each channel consists of three *signals*. Channel signal $x.irdy$ indicates whether the initiator is ready to write to channel x. Channel signal $x.trdy$ indicates whether the target is ready to read channel x. Channel signal $x.data$ contains data that is transferred from the initiator output to the target input if and only if both signals $x.irdy$ and $x.trdy$ are set to true.

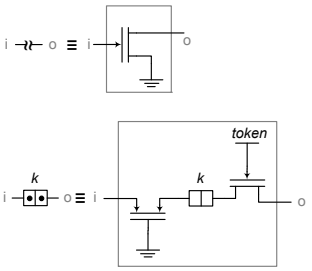
Figure 2 shows the eight primitives of the xMAS language. A *queue* stores data and is the state holding element. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Typically, functions are used to convert message types and represent message dependencies inside the fabric or in the model of the environment. Messages are non-deterministically produced and consumed at *sources* and *sinks*. A source may process multiple message types. We assume sources and sinks are fair, i.e., they will always eventually be ready to transmit (in case of the source) or receive (in case of the sink). A *fork* duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two outputs are both ready to read. A *join* is the dual of a fork. The function parameter determines how the two incoming packets are combined. A transfer takes place if and only if the two inputs are ready to send and the output is ready to read. A *switch* uses its function parameter to determine to which output an incoming packet must be routed. It models deterministic routing; adaptive routing is not yet supported. A *merge* is an arbiter. It grants its output to one of its inputs. We assume that merges are fair, e.g., round-robin or FIFO.

A *configuration* represents the current occupation of queues, i.e., the current state. Configurations are updated when messages are produced, consumed, or moved to one or more next queues.

Example 1: Consider the xMAS model in Figure 1. Two agents are communicating via a fabric. Each agent injects requests at a source. A join synchronizes the source with a credit counting mechanism, ensuring that at most k requests can be enroute in the network at all times. The stream of injected requests is merged with the stream of responses that have been generated as result of receiving requests from the other agent. The combined stream flows to the communication fabric, i.e., queue dx_1. Arriving at the other agent, the streams are split: ingress queue iq_1 receives requests, queue iq_2 receives the responses. When packets leave these queues to be sent back as a response or to be consumed, the forks ensure that tokens are sent back for correct bookkeeping by the credit counters.



(a) Two Agents



(b) Macro's

Fig. 1: Example of network [11]

The credit counters are modelled in a *macro block*, which can be considered as an open xMAS model, i.e., an xMAS model with unconnected in- and outputs.

B. Deadlocks

One of the properties most hard to verify for xMAS fabrics is the absence of *deadlocks*. A deadlock is a configuration in which at least one packet is enroute towards its destination but is permanently blocked. Since both sinks and merges are fair, deadlocks occur because of design flaws in the combinatorial logic between queues. Some packet is stored in a queue and ready to be transmitted, but since its targetted queues are permanently unable to accept it, the packet is stuck. We therefore detect *dead queues*, i.e., queues that contain a packet that is permanently blocked.

We use Linear Temporal Logic (LTL) to formally define deadlocks [1]. LTL provides the $\mathbb{G}$lobally operator to express that some property is always true, and the $\mathbb{F}$inally operator which expresses that some property is eventually true. The channel that is connected to queue q is referred to with $q.out$.

Definition 1: A *deadlock* is a configuration in which some queue q is ready to send data, but is permanently unable to transmit it.

$$\mathbf{Dead}(q) = \mathbb{F}(q.out.irdy \wedge \mathbb{G}\,\neg q.out.trdy)$$

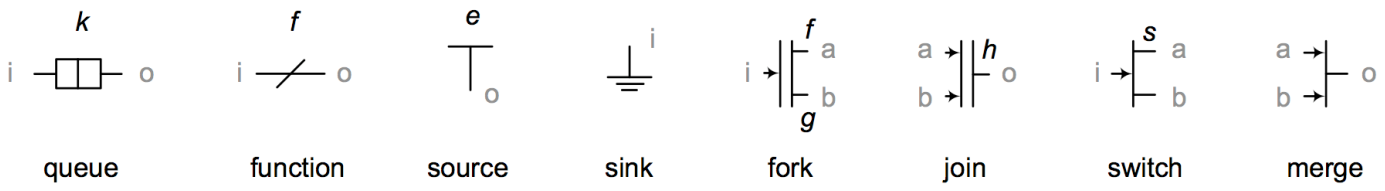


Fig. 2: Eight primitives of the xMAS language [7]. Italicized letters indicate parameters. Gray letters indicate ports.

Eventually, a configuration is reached in which $q.out.irdy$ is high, meaning that queue q contains a packet. However, $q.out.trdy$ is permanently low, meaning that the packet will never leave the queue. Such a situation can arise due to starvation or a – possibly local – deadlock. Since we assume fair merges, no starvation can occur. Therefor, this definition identifies deadlocks.

Example 2: We have modelled the communication fabric in Figure 1 in WickedXmas, but with oversized credit counters. Due to this oversizing, the network has a deadlock. This deadlock occurs when the source of agent P injects requests into queue dx_1 which are then routed to queue Qiq_1. The oversized credit counting mechanism allows at most 10 requests in these two queues. Since queue Qiq_1 has size 9, it is possible that queue Qiq_1 is full and queue dx_1 contains 1 request. This request can block a response that wants to travel from $dx1$ to queue Qiq_2. In such a configuration, queue dx_1 is full and cannot accept incoming packets. If the exact same happens at the right-hand side of the fabric, a deadlock occurs. The deadlock configuration is the following assignment of packets to queues:

$$\begin{array}{rclrcl} dx_1.req & = & 1 & dx_2.req & = & 1 \\ dx_1.rsp & = & 1 & dx_2.rsp & = & 1 \\ Piq_1.req & = & 9 & Qiq_1.req & = & 9 \end{array}$$

Note that this deadlock is prevented by correct credit counting mechanisms: if at most 9 requests are allowed in queues dx_1 and Qiq_1, then a configuration in which a request in dx_1 permanently blocks responses would not be possible.

C. Usage of xMAS

Currently, xMAS is used for design and verification of microarchitectural communication fabrics at Intel's Strategic CAD Labs. At Intel, designers manually build a representation of an xMAS diagram in C++ [6]. The C++ description is compiled down to a hardware language (synchronous, single clock, edge-triggered Verilog [4], [5]). The Verilog is model checked using ABC. Inductive invariants are generated from the C++ description which are used for the verification of the model. Their purpose is to facilitate the model checking by providing speed-up and hints for verification.

WickedXmas has three major advantages on this methodology. First, it provides completely automatic verification and compilation to a representation in C++. This saves significant time, as the main part of the modelling effort is in sketching the diagram and converting it to C++ [6]. During this manual process, modelling bugs can occur such as demux conditions which are not mutually exclusive, build errors, or dangling channels. Even though these are often easily corrected, a graphical editor such as WickedXmas is much better tailored for designing models in the graphical language xMAS. Any drawing in WickedXmas is correct by construction when it comes to these kinds of modelling bugs.

Secondly, WickedXmas does not verify on a low-level of abstraction such as Verilog. It directly verifies the xMAS model using a dedicated algorithm [14]. This approach leads to better scalability.

Thirdly, WickedXmas is able to visualize deadlocks in the communication fabrics. The deadlock described in Example 2 and its visualization have both been produced automatically by WickedXmas. The tool finds deadlocks and shows the user which queues are filled with which kinds of messages. This graphical feedback can help in understanding and evaluating the design.

III. Tool Architecture

A network designer only interacts with a graphical user interface for WickedXmas. To verify that a network contains no deadlocks, several modules are executed sequentially. Upon verification, the network is stored into a single file. The network is then inspected, and an executable to perform deadlock detection is generated, along with the network information it needs. Should any deadlock configurations be found, one such configuration is presented to the user. Figure 3 presents an overview of this architecture of WickedXmas.

A. User interface

The user interface to WickedXmas is an executable in which networks can be composed out of primitives and macros.

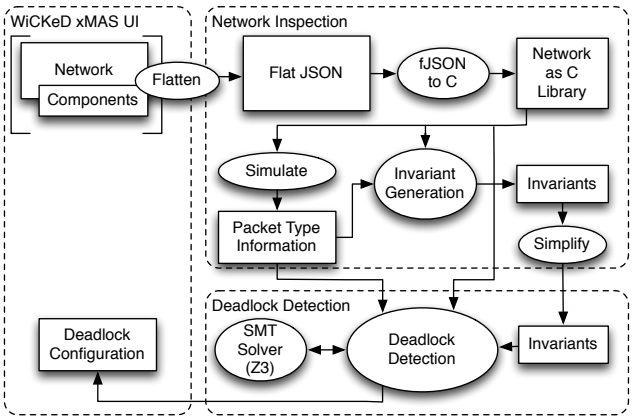


Fig. 3: Tool Architecture

Each macro can be stored into a separate file, to enable modular design. Upon starting the verification process, a single file is generated containing a flattened description of the network. This file, like the original network, is stored in a JSON structure (for: Java Script Object Notation). We call this structure 'flat' since it fully describes the network in terms of primitives, that is: it does not contain any macros.

In the user interface, a designer can click through the various steps of the deadlock detection. Should a deadlock be found, its configuration is presented to the user. More details can be read in Section IV.

B. Network inspection

The first step to detect deadlocks from a flattened description of a network, is performing an analysis on the network. In contrast to the user interface, these analysis tools are cross-platform, and written in various languages. The tasks performed are:

- Parsing the flattened network, generating C++ code.
- Performing a simulation on the network to obtain packet type information.
- Generating invariants that hold for the network.
- Simplifying these invariants.
- Finding deadlocks.

To parse the JSON structure, we use a JavaScript file, which is executed through phantomjs. The output of this file is C++ code describing a network, in which xMAS functions become C++ functions. This allows a compiler to perform optimization on these functions when they are compiled into several executables.

The first executable generated performs a simulation that assigns packet types to sources and queues. This information on packet types is stored in a separate file, which is used for generating invariants and finding deadlocks. It is obtained by a forward simulation without any assumptions on timing. Initially, the algorithm assumes no packets can arrive at any queue. Starting at the sources, for every packet type it can emit, its path is followed through the network to the next queue. That queue is then updated: the packet which just arrived there is added to the list of packet types that may arrive at that queue. The algorithm is repeated for all queues and corresponding types, until a fixed point is reached. Since we only allow for a finite number of packet types, we are guaranteed to reach this state.

The generation of invariants occurs in two steps. In the first step, invariants concerning all message types are generated, according to the technique described in [4], [5].

Example 3: Consider again the network in Figure 1. Four credit counting mechanisms are regulating the flow of packets arriving at the agents. Credit counter $cc1$ ensures that *invariably* at all times the number of requests in queues dx_1 and Qiq_1 is less than or equal to k. This is an invariant that is found automatically:

$$dx_1.req + Qiq_1.req \leq k$$

Another invariant, that is induced from the typing information, states that invariably queue dx_1 contains requests or responses, whereas queue Qiq_1 contains requests only:

$$\begin{aligned} dx_1 &= dx_1.req + dx_1.rsp \\ Qiq_1 &= Qiq_1.req \end{aligned}$$

These invariants rule out the deadlock described in Example 2.

During and after invariant generation, the set of found invariants is simplified. This simplification may cause the deadlock detection to perform its analysis independently of packet types, reducing the time it takes to verify a network. The initial generation of invariants requires knowledge about the network, which is obtained from the C++ library. The simplification part does not require this knowledge. It is a precompiled Haskell program which returns the invariants in SMT2 format, which allows us to parse it more easily.

The final executable generated with the C++ description of the network is that of the deadlock detection. This executable uses the previously derived packet type information, along with the generated invariants.

C. Deadlock detection

The deadlock detection is an implementation of the algorithm described in [14]. This algorithm may find many deadlock configurations. In generating these configurations, it ignores most invariants. For each such deadlock configuration, the SMT solver Z3 [10] is queried to determine whether that configuration is actually in compliance with the generated invariants. If deadlocks exists, it is guaranteed that the tool will find one. If none of the deadlock configurations can occur, the network is guaranteed to be live. Should there be a deadlock configuration which cannot be excluded by the invariants, it is returned for closer inspection by the network designer. The designer may then decide to fix his network, or add (unverified) properties about the network in the invariants file manually in order to exclude the configuration found.

IV. Tool Usage

Figure 4 shows the main interface of WickedXmas. We detail the actions involved in drawing and making a complete design.

Defining packet types: The first step in designing a communication fabric is defining the contents of the packets. WickedXmas assumes homogeneous packet types. Each packet consists of one or more fields. Consider, for example, a 10×10 2D mesh with request/response packets. Each packet contains the destination coordinates, the message type (i.e., request or response) and the coordinates of the source. To provide this information to WickedXmas, the *packet type window* is opened. For each field a line is inserted of the following form:

$$f < n$$

Here f is the name of the field and n represents its range. A request (response) is modelled by a packet whose type is 0 (1). This example will be used as running example in this section.

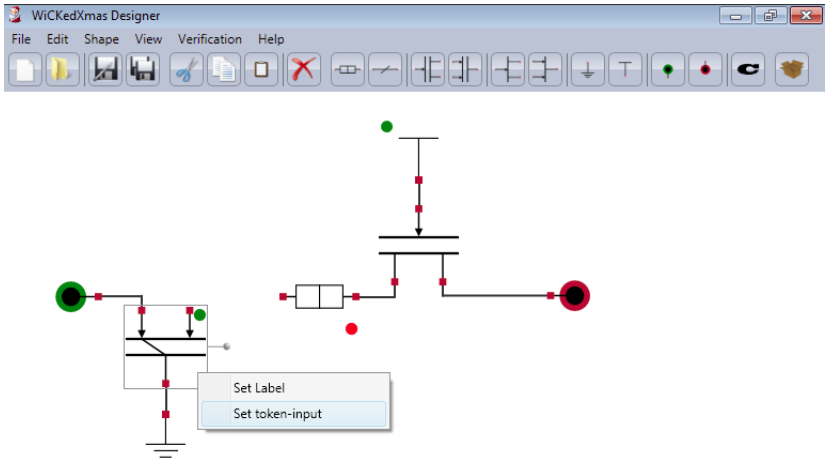


Fig. 4: The user interface

Adding components: The tool allows for placing, rotating and dragging components as the user would expect. Also, copy/paste behavior is implemented. Aside from the regular components, there is a specific 'input' and 'output' component, to allow for the creation of macro blocks. All these components are available both from the 'Shape' menu and from the toolbar.

Some components require extra information. For example, sources require information about the type of packet that is injected and queues require a known queue size. All components that need extra information feature a bullet alongside it. When the bullet is red, the corresponding component still requires more information. By right clicking the component, fields can be edited. When the bullet is green, the component is 'done'.

All components can be named by given them a label. This is optional, but naming the queues is recommended for readable feedback.

Connecting components: The layout of components is given by the user, by dragging and placing them. The layout of connecting wires is automatic, only a start and endpoint have to be given. The tool prevents connections from outputs to outputs or inputs to inputs.

Verifying the design: A design is complete only when 1.) the bullets of all components are green and 2.) all components are connected. When the design is complete, a user can click through the process of deadlock verification. The first step verifies that the design is complete. It then creates a JSON file containing a flattened description of the network as explained in Section III-A. Step 2 generates all verification tools. If the user wants to inspect the invariants that have been generated, it can execute Step 3. Inspecting the invariants is optional, but it can be very useful to get insight into the design. The final step runs deadlock detection. Should there be a deadlock, this information will be presented to the user.

The major part of the design effort is in adding components. We provide details on how the user can finish each component:

queue: When adding a queue, the user must provide its size. Currently, this has to be a natural number. Our verification tools can deal with parametric queue sizes, but before we allow this some technical details in the tool flow have to be dealt with.

function: A function component requires the user to specify the packet transformation that is executed by the function. The transformation takes as input a packet p and outputs a packet ret. Initially, packet ret is a copy of p. If nothing is entered, the function will be the identity function. The user can edit the fields of packet ret. Revisiting the 2D mesh example, let's say that a function transforms an incoming request to a response and sends it back to the source node. The user can specify this transformation as follows:

$$ret\_Xdst = p\_Xsrc; ret\_Ydst = p\_Ysrc; ret\_type = 1;$$

A language of basic compare operators and mathematical functions is supported.

source: When adding a source, information must be added on the set of packets that are inject by that source. WickedXmas accepts a set comprehension for this. The set of all packets can be accessed through variable *PacketDomain*. The following line must be inserted when adding a source:

$$\{p \text{ in PacketDomain } \mid f(p)\}$$

Here f is predicate that returns true if and only if the given packet is injected at the source. Reconsider 2D-mesh with requests/response. Say we want to model that the source of node $(0,0)$ injects packets that are destined for all nodes in the other columns (i.e., not for the left-most column) and that are not responses. WickedXmas requires the following specification:

$$\{p \text{ in PacketDomain } \mid \\ \text{p\_Xsrc} == 0 \ \&\& \ \text{p\_Ysrc} == 0 \ \&\& \\ \text{p\_Xdst} > 0 \ \&\& \ \text{p\_type != } 1\}$$

Besides "and" and "not", the logical "or" can be used as well.

sink: We assume sinks to be fair. Therefore, a sink requires no extra information from the user.

fork: Without loss of generality, we assume forks without functions on the output channels [11]. Therefore, forks require no additional information.

join: For efficient verification, we assume that one of the inputs of a join is a token that is consumed by the join and one input is the packet that is to be transferred [11], [14]. The user has to specify which is which. Figure 4 shows a join in which the left input transmits packets, whereas the right input consumes tokens.

switch: A switch requires the specification of its routing logic. The user has to provide a function s that takes as input a packet p and returns true if and only if the packet is routed to the upper output channel of the switch. For example, say a switch has to route all packets destined for node $(1, 1)$ and all response packets to the upper output channel. The user adds the following specification:

$$\text{return}(\text{p\_Xdst} == 1 \;\&\&\; \text{p\_Ydst} == 1) \;||\; \text{p\_type} == 1;$$

merge: We assume that merges are fair. Therefore, the arbitration policy that is applied within the merge (e.g., FIFO or round-robin) is irrelevant for verification. A merge requires no extra information from the user.

composite object: In WickedXmas, composite objects are the key components to achieve scalable design. They are the components that implement macros. To create a new composite object, the user just has to save his file. When inserting a composite object, a dialog pops up asking what file to open. An input and output port will be created for every 'In connector' and 'Out connector' object in the original file.

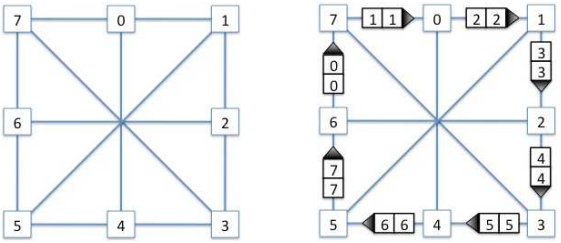
Composite objects can be parameterized. Functions, switches and queue sizes may depend on these parameters. Most importantly, the parameter can be used to decide the layout of a composite object. By clicking 'add tab', a case distinction can be made based on a parameter, effectively allowing a parameter to influence the implementation of an object. We allow composite objects to contain other composite objects, which enables a hierarchical design flow. Most importantly, we allow composite objects to contain *themselves*. The combination of parametric and hierarchical composite objects allows recursive design of communication fabrics.

V. Case Studies

In this section, we show that our tool can verify deadlock freedom of various examples from industry and academia. As mentioned in section III-C, our tool is guaranteed to find a deadlock in case deadlocks exist. To give a sense of the size of the networks, we give the number of queues. Readers interested in the state space of such networks can use 2^{qdw} as an estimate, where q is the number of queues, d the queue depth, and w is the data width.

A. Spidergon

We first consider a case study concerning the Spidergon of STMicroElectronics [8]. The Spidergon is a ring with 'across' channels. Packets move via the shortest path towards their destination (see Figure 5a). Without further modification, the chip suffers from routing deadlocks. A deadlock can occur if each clockwise channel going out of processing node n is filled with messages destined for node $n + 2 \mod p$ with p the total number of processing nodes (see Figure 5b).



(a) Spidergon ring topology (b) A deadlock. Each channel has capacity to store two packets.

Fig. 5: Spidergon of STMicroElectronics

Our aim is to formalize this design in WickedXmas and confirm this deadlock. Then, we will experiment with alternative versions to see how this influences the existence of deadlocks.

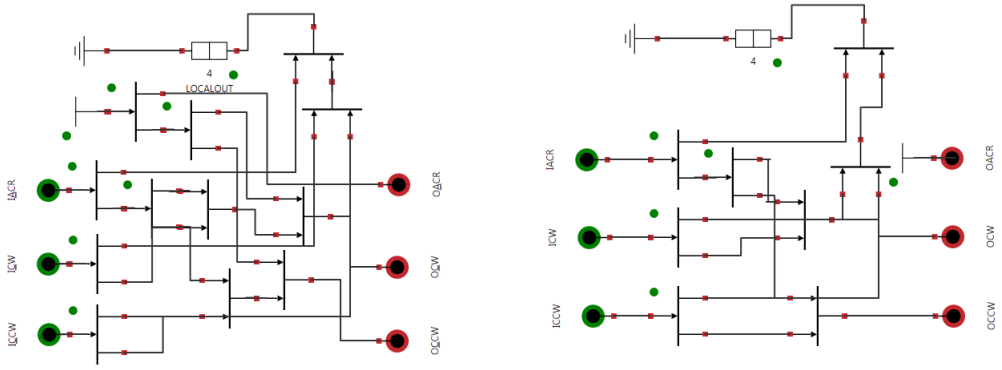
Figure 6a shows the design of the processing nodes. The figures are hard to read, but they are just intended to give an impression of a WickedXmas design. There are three inputs for messages coming in from the across, clockwise, or counter-clockwise direction. Similarly, there are three outputs. Messages are injected by the sources. The combination of wires, switches and merges implements a crossbar, routing messages in the right direction. If messages arrive at their destination, they are sent to the local out queue, where they are consumed by a sink. Note that this design is parametric. The routing logics at the switches base their decisions on the incoming packet p_n, but also on parameter $\$n$ (indicating which node it is) and \$p (the total number of processing nodes). Consider, for example, a newly injected packet at the source. The first switch decides whether the packet has to be routed across. The routing logic is formulated as follows:

$$(p\_n + \$p - \$n)\%\$p > \$p/4 \;\&\&\\ (p\_n + \$p - \$n)\%\$p < \$p - \$p/4;$$

The relative address $(p\_n + \$p - \$n)\%\$p$ is computed. If it is greater than $\$p/4$ (i.e., outside the quadrant reached clockwise) but less than $\$p - (\$p/4)$ (i.e., outside of the counter-clockwise quadrant), the packet is routed across.

We use this design to recursively build a ring (see Figure 7). The base case, for $\$i = 0$, is an "empty ring" where ins outs are simply connected. The inductive case, for $\$i > 0$, consists of three composite objects. Two of them are processing nodes UP and DOWN, which are across of each other. These nodes are, e.g., nodes 0 and 4 in Figure 5a. The third is a ring object with $\$i = \$i - 1$. This ring object will – during flattening – recursively unfold itself to two new nodes (i.e., the next two will be 1 and 5, then 2 and 6, etc.). There are in- and outputs for sending packets clockwise and counter-clockwise. Ultimately, this yields a composite object named *ring* that requires a parameter $\$i$ that dictates how many processing nodes the ring should contain.

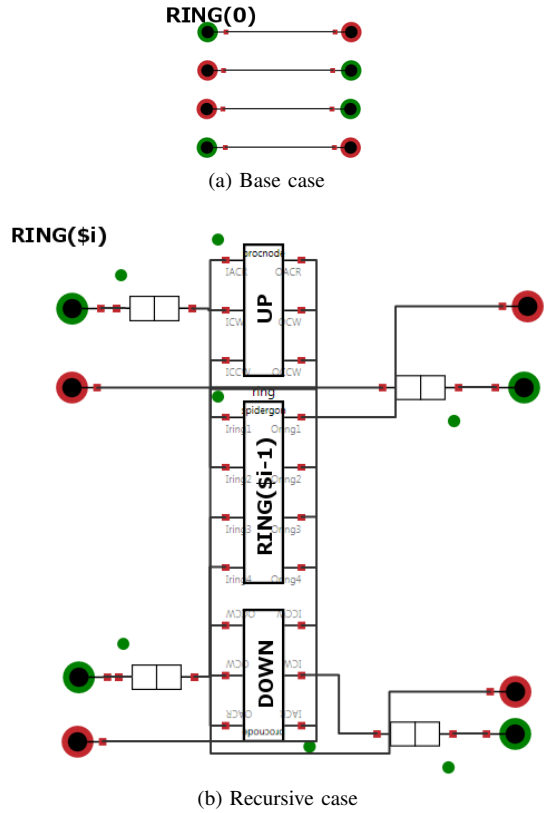
When executing deadlock detection, the deadlock in Figure 5b is found instantaneously. We experiment with different



(a) Processing Node (master)

(b) Processing Node (slave)

Fig. 6: Processing nodes



(a) Base case

(b) Recursive case

Fig. 7: Recursive design of the Spidergon

approaches to resolving this deadlock (see Table I). First we consider two types of processing nodes: masters (see Figure 6a) and slaves (see Figure 6b). Slaves are processing nodes that consume messages, but do no produce any. Masters produce messages for all other nodes. Depending on parameter $\$n$, a node is a master or a slave. We consider the case where the right upper quadrant consists of slaves. This resolves the deadlock for $\$p = 8$ (i.e., when the size of the ring is 8). However, for larger rings the deadlock – surprisingly – reappears.

We then change the behavior of master nodes in such a way that they send out messages to slaves only. This eliminates all deadlocks. WickedXmas is able to prove this for a 64 ring in approximately 15 minutes. A final experiment adds message dependencies to the network: masters send out requests, which when arriving at slaves are transformed into responses and sent back. WickedXmas proves that adding message dependencies does introduce deadlocks to this particular configuration of masters and slaves, but only for rings with more than eight nodes. Adding message dependencies significantly increases running time, since packets become larger (they now store their original source and a message type).

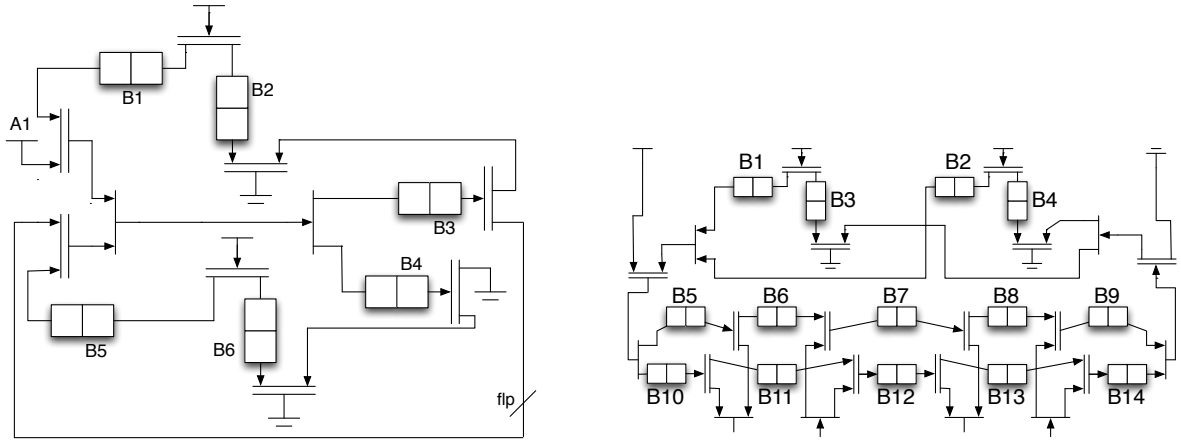
Size	8	16	32	64
Number of queues	32	64	128	256
Verification Time	1 sec.	7 sec.	45 sec.	± 15 min.
Spidergon	×	×	×	×
+ 1 quadrant slaves	✓	×	×	×
+ packets to slaves only	✓	✓	✓	✓
Verification Time	58 sec.	N/A	N/A	N/A
+ requests/responses	✓	×	×	×

TABLE I: Experimental results with variations of the Spidergon architecture.
✓= deadlock-free, × = deadlock

B. Virtual Channels and Scoreboards

Figure 8a shows a network from a paper by Ray and Brayton [13]. The switch is configured such that packets coming from A1 are routed to B3, and packets coming from the bottom half are routed to B4. The figure shows a configuration in a deadlock situation which can be explained as follows: Buffer B3 is full and permanently blocked, as the arbiter is in a state that gives priority to the top channel. Packets in the top channel, however, are destined for B3, which is full. For this reason, the arbiter will not give the turn to the bottom channel, since it can only do so after a transfer. It is critical to this deadlock that there is a packet in B1, and a packet entering at A1. Should either of these packets be missing, the arbiter cannot give priority to the top channel.

There are two invariants for the network shown in Fig-



(a) Deadlocking virtual channel

(b) Two entry scoreboard

Fig. 8: xMAS examples from [13]

ure 8a, namely:

$$B2 = B1 + B3$$
$$B6 = B5 + B4$$

The deadlock can be prevented by reducing the size of B2, such that B1 is empty whenever B3 is full. In other words: B2 can be as large as B3, but no larger. Our tool finds both invariants. It finds the deadlock if the queue sizes are such that there is one.

A second network from the same paper is a two entry scoreboard, as shown in Figure 8b. In this example, a workload gets assigned a token from the top half of the network. Depending on this token, the packet traverses queues B5 to B9, or queues B10 to B14. There are two phases in which 'work' is done on the packet: one phase happens while the packet is in queue B6 or B11, and the other while it is in B8 or B13. In our tool, the two phases of the scoreboard were modeled by single queues, giving a total of 16 queues. Of the two methods proposed in the paper, only one of them was able to prove what the authors call a 'response property', taking 38 seconds to verify after adding several properties about the network manually. This property is similar to the property we verify in our tool. Our tool verifies the network to be deadlock free in a few seconds.

VI. Conclusion

We presented WickedXmas, a toolset supporting the graphical design of communication fabrics. The toolset includes the generation of inductive invariants and an efficient deadlock checker. The design language is the xMAS language originally proposed by Intel. We illustrated the applicability and performance of WickedXmas on variations of the Spidergon by STMicroElectronics. Once the design was drawn, we could tweak it to study the effects of different layouts of masters and slaves, or adding message dependencies. We also tackled a few tricky examples from the literature.

We envision many possible extension of WickedXmas. Allowing parametric queue sizes and parametric macro's would significantly increase the tools' ability to deal with large repetitive structures such as meshes and rings. Adding support for hierarchical structures could increase scalability. Most of all, we aim at providing compilers from xMAS to Verilog. WickedXmas can then be used to model and verify at a high level of abstraction, but to generate correct low-level hardware designs.

Acknowledgements: This research is supported by an NWO/EW project Effective Layered Verification of Networks-on-Chip (ELVeN) under grant no. 612.001.108. This research is supported by a grant from Intel Corporation.

References

[1] Baier, C., Katoen, J.P.: Principles of model checking. The MIT Press (2008)

[2] Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2010)

[3] Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Computer Aided Verification, pp. 24–40. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2010)

[4] Chatterjee, S., Kishinevsky, M.: Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In: 22nd International Conference on Computer Aided Verification (CAV'10) (July 2010)

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

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

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

[8] Coppola, M., Grammatikakis, M., Locatelli, R., Mariuccia, G., Pieralisi, L.: Design of interconnect processing units Spidergon STNoC. CRC Press (2009)

[9] Dally, W.: The end of denial architecture. Keynote at DAC'09 (2009)

[10] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)

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

[12] Poliakov, I., Khomenko, V., Yakovlev, A.: Workcraft – a framework for interpreted graph models. In: Applications and Theory of Petri Nets, pp. 333–342 (2009)

[13] Ray, S., Brayton, R.K.: Scalable progress verification in credit-based flow-control systems. In: DATE. pp. 905–910 (2012)

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