View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Radboud Repository

Radboud Repository
Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/135061

Please be advised that this information was generated on 2018-07-07 and may be subject to change.

Inference of channel types in micro-architectural models of on-chip communication networks

Bernard van Gastel and Freek Verbeek
School of Computer Science
Open University of the Netherlands
and
Institute for Computing and Information Sciences
Radboud University Nijmegen
{bvg,fvb}@ou.nl, {b.vangastel,f.verbeek}@cs.ru.nl

Julien Schmaltz
Department of Mathematics and Computer Science
Eindhoven University of Technology
j.schmaltz@tue.nl

***Abstract*—In the multi-core era, on-chip communication networks are key to system correctness and performance. To deal with their growing complexity, micro-architectural models capture the intent of architects and provide means for formal analysis. However, the analysis of such micro-architectural models is restricted to non-scalable and/or very specific approaches. We present a novel scalable approach to support the symbolic channel type inference of large micro-architectural models described in the xMAS language proposed by Intel. We define an algorithm that computes all possible messages that can occur in a communication channel, treating their payload symbolically. These results can be used for further analysis such as verifying absence of misrouting, deriving inductive invariants and deadlock detection. We illustrate our approach on a Spidergon network developed at STMicroelectronics.**

I. Introduction

Communication networks are crucial to the overall correctness and performance of modern Multi-Processors Systems-on-Chips (MPSoC's). When the number of interconnected system elements increases, performance of bus-based architecture degrades [1]. Networks-on-Chips (NoC's) have emerged as solid high performance communications architectures. Recently, Intel proposed a language – called xMAS, for eXecutable Micro-Architectural Specification – to capture the intent of architects [2], [3]. These models are executable and also amenable to formal verification. It is possible to extract high-level information about a Verilog design from its xMAS abstraction. This information can then be used to improve model checking the Verilog description [4], [5], [6]. Efficient deadlock verification on large xMAS models was demonstrated in [7], [8]. Although many concrete simulation techniques exist, they lack scalability because they simulate a NoC with clock cycle precision, and are therefore not applicable to communication-centric SoCs [9]. We place ourselves in the context of the scalable verification at the level of xMAS models.

Our main contribution is to extend the analysis of xMAS models with the inference of channel types. Our approach is based on a symbolic propagation algorithm. This effectively computes the typing information of all channels, i.e., for each channel it computes the set of packets that can possibly traverse this channel. We define two symbolic types, enumeration and interval range. Every component/packet/function in the network is modelled in terms of sets and operations on sets of these symbolic types. Due to this modelling, a symbolic packet can be split and eventually end up in multiple sinks with different payloads. Our network representation closely matches the formal semantics of xMAS networks as described in [10], in order to make it feasible to use this approach in a generic formal proof. We implemented our algorithm in C++. Starting from a representation of an xMAS network (including the description of what kind of packets each node may inject), our procedure is fully automatic. We demonstrate the application and scalability of our algorithm to the Spidergon NoC from STMicroelectronics.

978-1-4799-6016-3/14/$31.00 ©2014 IEEE

II. xMAS and Spidergon

A. The xMAS language

An xMAS model is a network of primitives connected via typed *channels*. A channel is connected to an *initiator* and a *target* primitive. A channel is composed of three signals. Channel signal $c.irdy$ indicates whether the initiator is ready to write to channel c. Channel signal $c.trdy$ indicates whether the target is ready to read from channel c. Channel signal $c.data$ contains data that are transferred from the initiator output to the target input if and only if both signals $c.irdy$ and $c.trdy$ are set to true. Figure 1 shows the eight primitives of the xMAS language. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Typically, functions are used to convert packet types and represent message dependencies inside the fabric or in the model of the environment. A *fork* duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two

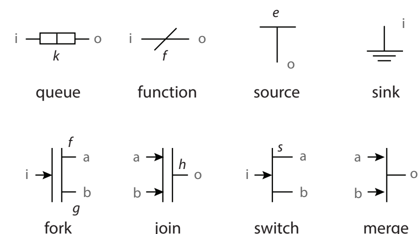


Fig. 1: The eight xMAS primitives.

outputs are both ready to read. A *join* is the dual of a fork. The function parameter determines how the two incoming packets are merged. A transfer takes place if and only if the two inputs are ready to send and the output is ready to read. A *switch* uses its function parameter to determine to which output an incoming packet must be routed. A *merge* is an arbiter. It grants its output to one of its inputs. The arbitration policy is a parameter of the merge. A *queue* stores data. Messages are non-deterministically produced and consumed at *sources* and *sinks*. A source or sink may process multiple packet types.

The execution semantics of an xMAS network consists in a combinatorial and a sequential part. The combinatorial part updates the values of channel signals. The sequential part is the synchronous update of all queues according to the values of the channel signals. A simulation cycle consists of a combinatorial and a sequential update. A sequential update only concerns queues, sinks, and sources. We denote these primitives as *sequential primitives*. Other primitives are denoted as *combinatorial*.

For each output port o, signal $o.irdy$ is set to true if the primitive can transmit a packet towards channel o, i.e., port o is ready to transmit to its target. For each input port i, signal $i.trdy$ is set to true if the primitive can accept a packet from input channel i, i.e., the target of channel i is ready to receive. In a sequential primitive, the values of output signals depend on the values of the input signals and an internal state. Queues accept packets only when they are empty. A source and a sink produces or consumes a packet according to an internal oracle modelling non-determinism.

B. The Spidergon NoC

Spidergon is a Network-on-Chip developed at STMicroelectronics [11]. The basic architecture consists of 8 nodes connected in a ring with across links (see Figure 2). A popular routing algorithm for this network is *across first*. The idea is that if a packet needs to use across links to minimise the travel distance an across hop is always performed first.

Figure 3 shows a micro-architectural model of a Spidergon router. The router has 4 inputs and 4 outputs. Packets are coming from either other routers or from input A, i.e., the local input. Switches are used to route packets towards their respective output ports. Arbiters are used to handle conflicting requests for output ports.

Different kinds of cores can be connected to the routers (see Figure 3). For sake of an example that includes message dependencies [12] we have a setup with masters and slaves. Masters inject requests and consume responses. Slaves transform requests into responses. For a topology of N nodes, a master injects packets that contain the following fields:

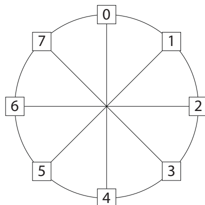
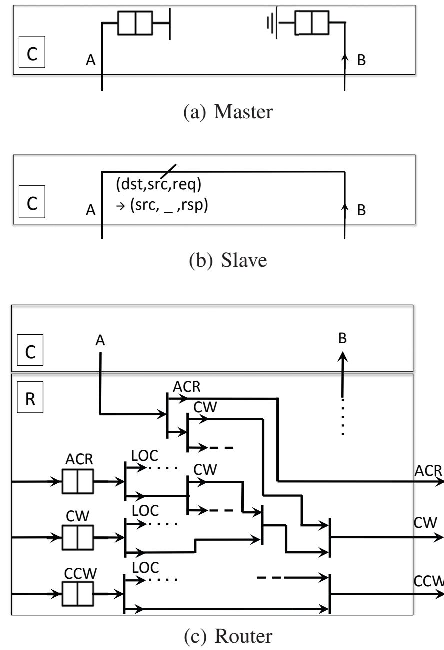


Fig. 2: Spidergon Topology



(a) Master

(b) Slave

(c) Router

Fig. 3: Spidergon Router with ACR (Across), CW (Clockwise) and CCW (Counterclockwise) incoming and outgoing connections, and two types of local nodes (master and slaves).

- dst An integer ranging from 0 to N that represents the destination of the packet. This destination is always a slave.
- src An integer ranging from 0 to N representing the original injection point of the packet. This is used to send the packet back after arriving at its destination.
- $colour$ Either request (when it is injected) or response (when the packet has visited its destination and is returning to its original injection point).
- $payload$ Some 32 bit integer.

In the remainder of this paper, we will use the Spidergon design as a running example. We will show how one can define the routing functions within the switches and specify at sources what kind of packets can be injected. We will use our algorithm to compute all typing information for configurations going from 8 to 1024 nodes.

III. Specification of packets

A simple expression based language is needed to express the intent and effect of primitives. Packets consist of a number of fields. Each field is either an integer (e.g., dst and src in the Spidergon example) or an enumeration (e.g., the colour). There are two kinds of expressions supported: *matching expressions* and *modifying expressions*. Matching expressions are used in xMAS-sources and xMAS-switches to determine which packets are injected. Modifying expressions are used in the xMAS-function to express how a packet is altered.

For example, the following matching expression (automatically generated) corresponds to the xMAS-switch connected to the local input A (see Figure 3). This switch decides whether the incoming packet is routed across. We consider the fourth node in the network, meaning that packets destined for

0, 1, and 7 should be routed across. The following expression generates two intervals: $[-1 \,..\, 1]$ and $[7 \,..\, 9]$. As the dst field signifies a node number (which in the example is between 0 and 7), this expression only matches for nodes 0, 1, and 7, as desired.

```
(dst > 4 ? dst >  6 : dst > -2) &&
(dst > 4 ? dst < 10 : dst <  2)
```

As a second example, the following modifying expression is used by a slave to transform requests into responses:

```
dst := src, colour := colour with {req: rsp}
```

This expression yields a new packet with as destination the original source. The new colour is obtained by providing the old colour to a mapping. This mapping turns requests into responses.

We support the following syntax for matching expressions, as described by this BNF-grammar:

⟨*expr*⟩ ::= ⟨*enum-match*⟩ | ⟨*integer-match*⟩
  | '(' ⟨*expr*⟩ ')'
  | '!' ⟨*expr*⟩
  | ⟨*expr*⟩ '?' ⟨*expr*⟩ ':' ⟨*expr*⟩
  | ⟨*expr*⟩ ⟨*logical-op*⟩ ⟨*expr*⟩

⟨*logical-op*⟩ ::= 'and' | '&&' | 'or' | '||'

⟨*enum-match*⟩ ::= ⟨*variable*⟩
  | ⟨*variable*⟩ 'in' '{' ⟨*enum-contents*⟩ '}'
  | ⟨*variable*⟩ 'not' 'in' '{' ⟨*enum-contents*⟩ '}'

⟨*enum-contents*⟩ ::= ⟨*label*⟩ | ⟨*label*⟩ ',' ⟨*enum-contents*⟩

⟨*integer-match*⟩ ::= ⟨*variable*⟩
  | ⟨*variable*⟩ ⟨*compare-op*⟩ ⟨*constant*⟩
  | ⟨*variable*⟩ 'in' '[' ⟨*constant*⟩ '..' ⟨*constant*⟩ ']'
  | ⟨*variable*⟩ 'not' 'in' '[' ⟨*constant*⟩ '..' ⟨*constant*⟩ ']'

⟨*compare-op*⟩ ::= '<' | '<=' | '>=' | '>'

⟨*constant*⟩ ::= ⟨*integer*⟩
  | ⟨*constant*⟩ ⟨*constant-op*⟩ ⟨*constant*⟩
  | '(' ⟨*constant*⟩ ')'

⟨*constant-op*⟩ ::= '+' | '-' | '*' | '/' | '%' | '^'

For definition of modifying expressions we support another syntax, as they express another kind of intent. The syntax for these expressions is described by the following BNF-grammar:

⟨*expr*⟩ ::= ⟨*field-definition*⟩
  | ⟨*expr*⟩ ',' ⟨*field-definition*⟩

⟨*field-definition*⟩ ::= ⟨*variable*⟩ ':=' ⟨*value-expr*⟩

⟨*value-expr*⟩ ::= ⟨*variable*⟩ | ⟨*integer*⟩ | '(' ⟨*value-expr*⟩ ')'
  | ⟨*value-expr*⟩ ⟨*arithmetic-op*⟩ ⟨*value-expr*⟩
  | ⟨*value-expr*⟩ 'with' '{' ⟨*substitution-def*⟩ '}'

⟨*arithmetic-op*⟩ ::= '+' | '-' | '*' | '/'

⟨*substitution-def*⟩ ::= ⟨*label*⟩ ':' ⟨*label*⟩
  | ⟨*label*⟩ ':' ⟨*label*⟩ ',' ⟨*substitution-def*⟩

If a substitution is defined on an expression, a special label '_' can be defined, which is the default (fail-over) case of the substitution.

IV. Symbolic representation of packets

The basic principle of our symbolic computation is to represent a set of concrete packets in a significantly smaller set of symbolic packets. The computation of all paths for concrete packets is effectively done by propagating their symbolic representations. Fields of the integer kind are symbolically represented by intervals, fields of the enum kind are represented by abstract enumeration labels.

To manipulate symbolic packets, we assume the following common operations on *fields*:

- *Intersection* of two sets.
- *Difference* of set a and b, defined as $a \cap \bar{b}$. This can be computed without having the means to take the complement of a set.
- *Subset*. Check if a set is a subset of another set.

Note that the union operation is not explicitly listed, as we can represent the union of two symbolic packets by associating these two symbolic packets with the same channel independently of each other.

These operations are used for the manipulation of symbolic packets, by applying them in a field-wise fashion. We assume that each packet-field has a unique *label*. For each field operation $a \Box b$ ($\Box$ is intersection or difference) and for each label l of the first argument, if the label is also present in the second argument, perform $a_l \Box b_l$. If the label is not present, the resulting field for label l in the result is equal to the field in the first argument a.

Consider again the Spidergon example. For an 8 node Spidergon, the source and destination fields will be in a range of 0 to 7. The payload can be any value between 0 and $2^{32} - 1$. All possible concrete packets of these fields, can be represented in the following singleton set containing one symbolic packet:

$$\left\{ \begin{array}{ll} dst \rightarrow [0 \,..\, 7] & colour \rightarrow \{request, response\} \\ src \rightarrow [0 \,..\, 7] & payload \rightarrow \left[0 \,..\, 2^{32} - 1\right] \end{array} \right\}$$

We now describe the field operations that are specific for each field kind.

A. Enumerate kind

This kind is the standard enumeration type, where the labels are kept symbolic. The only operation defined on this kind is a mapping operation, converting one value in another one. We denote an enumerate field by $\{a, b, c, d\}$.

B. Interval range kind

Intervals have a lower bound l and an upper bound h, both represented as an integer for which $l \leq h$. A number of integer arithmetic operations are supported on these types, such as addition and subtraction on intervals, and comparisons with a concrete number, like the operators greater than, less than, etc.

Multiplication is supported, but as it can result in multiple noncontinuous regions it is prone to state explosion. An interval field is denoted by $[l \mathinner{..} h]$.

The addition of two bounds is defined by adding the lower bounds together for the new lower bound, and doing the same for the upper bound. Formally, we have the following definition:

$$[a \mathinner{..} b] + [c \mathinner{..} d] = [(a + c) \mathinner{..} (b + d)]$$

The unary minus operation is defined as:

$$-[a \mathinner{..} b] = [-b \mathinner{..} -a]$$

Combining these two definition, subtraction is defined as follows:

$$[a \mathinner{..} b] - [c \mathinner{..} d] = [(a - d) \mathinner{..} (b - c)]$$

Multiplication results in multiple nonadjacent intervals, and is defined as follows:

$$[a \mathinner{..} b] \times [c \mathinner{..} d] = \forall_{i \in [a \mathinner{..} b]} \forall_{j \in [c \mathinner{..} d]} [(i \times j) \mathinner{..} (i \times j)]$$

Division is defined by:

$$[a \mathinner{..} b] \,/\, [c \mathinner{..} d] = \left[\left\lfloor \frac{a}{d} \right\rfloor \mathinner{..} \left\lceil \frac{b}{c} \right\rceil \right]$$

As division by zero is not defined on integers, we also consider division by an interval that includes zero an error as it should not occur.

C. *Symbolic semantics for* XMAS

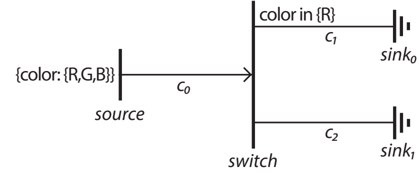
Queues, merges, forks and *sinks* do not modify packets, i.e. all packets going into the primitive propagate to all the output channels unmodified. XMAS sinks only receive packets, so no special care is needed for sinks. Regarding the remaining primitives, they can either filter packets (XMAS-switch), combine two packets together (XMAS-join) or modify the packets (XMAS-function). We now detail the semantics of these primitives.

xMAS-switch: A switch has a switching condition associated with it. This condition describes a set of packets that should be routed to output a. Since a symbolic packet signifies a set of possible values, a symbolic packet sw can be used to represent the switching condition.

The effect of the XMAS-switch on packets in the input channel can be expressed using set operators, specifically the intersection and difference operators. The intersection of p with sw will yield the resulting packets that will be propagated to the channel connected to the a output: $p \cap sw$. Calculating which packets will propagate to the b output can be done by taking the intersection of p with the complement of sw: $input \cap \overline{sw}$.

The switching condition is described using the syntax for matching expressions as described in Section III. We show by example how we derive a symbolic packet sw from such an expression. The expression `a <= 10` describes the field a interval $[0 \mathinner{..} 10]$. Combining this restriction with another can be expressed by `a <= 10 && a >= 5`, which will yield the intersections of $[0 \mathinner{..} 10]$ and $[5 \mathinner{..} \infty]$, and result in the interval $[5 \mathinner{..} 10]$. The conditional expression can also be expressed as a series of set operations: we define the semantic of `c?a:b` as $(c \cap a) \cup (\overline{c} \cap b)$, which results in two symbolic packets stored in the associated channel (as the union operation is not explicitly defined).

Example 1: A switch routes packets based on the colour of the packet. From the matching expression `colour in {R}` a symbolic representation is derived: $\{colour \rightarrow \{R\}\}$. If the symbolic packet $\{colour \rightarrow \{R, G, B\}\}$ is located in c_0, the effect of propagating the packet can be calculated using set operators. The resulting type in c_1 will be the intersection of the input packet with the symbolic representation of the switching condition: $\{colour \rightarrow \{R, G, B\}\} \cap \{colour \rightarrow \{R\}\}$, resulting in $\{colour \rightarrow \{R\}\}$. Likewise, the resulting type in channel c_2 will be $\{colour \rightarrow \{R, G, B\}\} \cap \overline{\{colour \rightarrow \{R\}\}}$, resulting in $\{colour \rightarrow \{G, B\}\}$.



xMAS-function: The *function* primitive applies a function on a packet going through the primitive. Although the function is defined over concrete packets, the same function can be applied to symbolic packets. The effect of a XMAS-function is specific to a kind of a field.

Example 2: A network containing a function with as function body: `result := x + y` (with `result`, `x`, `y` of the interval kind) transforms concrete packets going into the function. We can also calculate the effect of the function symbolically by using the symbolic semantics on intervals. If the function is applied to a symbolic packet $\{x \rightarrow [0 \mathinner{..} 16], y \rightarrow [8 \mathinner{..} 32]\}$, the result will be combining the two intervals. The resulting packet will be $\{result \rightarrow [8 \mathinner{..} 48]\}$.

xMAS-join: A join primitive combines the available symbolic packets of the two inputs into one packet, with each field prefixed with `a_` or `b_` to make clear from which input channel the field came. This is a somewhat different semantics as opposed to the original XMAS paper [3], as we split their join in our join and a normal XMAS-function. This eliminates the need to separately handle the function part of the XMAS-join, so we do not need to define any additional syntax for joining expressions.

xMAS-source: An expression on a XMAS-source primitive signifies which symbolic packets might be inserted at this point, which constitute a set of possible values that can be injected. These expressions are also described using the syntax for matching expressions, the same as for the XMAS-switch.

V. Type inference algorithm

The type inference algorithm (see Algorithm 1) is based on iteratively propagating a symbolic packet from a channel to the next channel, connected by a primitive. Propagation continues until a fix point has been reached, where no new inference can be performed. The algorithm outputs for each channel the set of symbolic packets describing which concrete packets can occur at the channel.

Algorithm 1: The basic propagation algorithm

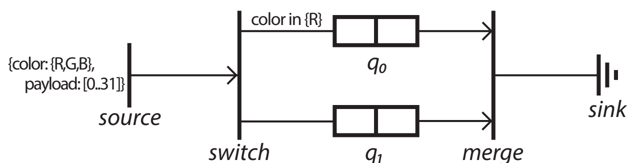
```
inject source types into channels;
while not all types in the network are marked as propagated do
    forall the channels in the network do
        normalise types in channel;
        forall the types in the channel do
            if type is not marked as propagated then
                propagate;
                mark type as propagated;
```

A normalisation procedure is required to reach a fix point. If packets are added to a channel, all the available packets in the channel are normalised. The normalisation consists of two parts: *eliminating* symbolic packets that are already contained in other symbolic packets, and *combining* two symbolic packets together if possible. Both parts of the normalisation are essential.

The *elimination step* checks whether a symbolic packet is already described by another symbolic packet. If so, the symbolic packet can be removed with no effect on correctness. A symbolic packet is assumed to be described by another symbolic packet, if for each field in the symbolic packet the other symbolic packet contains a superset of the values allowed by that field.

The second step of the normalisation is the *combination step*. A symbolic packet is combinable if and only if 1.) all fields are equal, except for one field, and 2.) if this one field is combinable independently of the other fields. For enumeration field kinds, two fields are always combinable. For interval fields a and b are combinable if and only if $(a.min \leq b.min \wedge b.min \leq a.max + 1) \vee (a.min \leq b.max \wedge b.max \leq a.max)$, i.e. if one of the bounds of b lies in the interval as represented by a. For example, an interval of $[a\,..\,b]$ and $[(b+1)\,..\,c]$ can be combined into one interval of $[a\,..\,c]$. This step is used to reduce the runtime of the algorithm as it reduces the number of propagation steps needed.

Example 3: We demonstrate the combination step using a small network. In this network, the channel at the source and the sink have the same symbolic type $\{colour \rightarrow \{R, G, B\}, payload \rightarrow [0\,..\,31]\}$. In the input channel of q_0 there are only symbolic packets with $\{colour \rightarrow \{R\}, payload \rightarrow [0\,..\,31]\}$, as the switching condition only matches packets that are of $colour$ R. The remaining packets are routed to the lower route with q_1. The merge propagates all the packets from the channels of the queues to the last channel. The type occurring at the sink is the result of combining the types $\{colour \rightarrow \{R\}, payload \rightarrow [0\,..\,31]\}$ with $\{colour \rightarrow \{G, B\}, payload \rightarrow [0\,..\,31]\}$. The network is listed below.



VI. Implementation

We have implemented the aforementioned algorithms as a single-threaded program called `sym-xmas`[1] in C++11, without any external library dependencies. To easily create XMAS networks we have created a graphical interface to easily draw XMAS networks. The tool chain flow is shown in Figure 4. The network representation in the JSON format is outputted by this graphical tool. This JSON file can be read by our sym-XMAS tool. The file representation of primitives is straightforward, but for the representation of the functions as used in the definitions of the XMAS switch, join and function primitives.

The syntax as defined in Section III is read using a recursive descent parser. For matching expressions a set of possible values is generated by evaluating the expressions during parsing. We also added a constant expression evaluator for convenience of the user, so constant expressions with addition, subtraction, multiplication are supported. For modifying expressions we use an internal representation which applies the operations as defined in Section IV.

Our tool defines on all symbolic packets and field kinds some extra operations (e.g. hash function and printing a string representation) that are not essential to the algorithm but aid in execution speed and the debugging effort. All kinds of typing errors are detected, both conflicting restrictions on fields as well as applying expressions on non existent fields.

We developed our tool to be a common foundation to deploy all sorts of algorithms to analyse various correctness properties. Therefore it is important to maximise the flexibility of the network data structure. Two aspects are important. Foremost, the data structure supports the *visitor* design pattern [13], enabling the decoupling of the algorithm executing on a data structure and the data structure itself. It enables add methods to the data structure without modifying the data structure itself, so a new algorithm or features can be added without modifying all the code depending on this data structure. Secondly, each object of the network can have algorithm specific data structure attached to it. This avoids costly lookups in mappings and enables easier algorithm design. In the future other algorithms, e.g. checking if a network conforms to a given specification, can be based on this foundation.

There are two basic correctness requirements to XMAS networks: syntactic correctness and absence of combinatorial cycles. Before we symbolically infer types we check that an input XMAS network satisfies these correctness properties, otherwise the results of the type inference are not sound. Both are implemented using the data structure and the visitor pattern as mentioned before. The first one is easily checked by ensuring all ports are connected and output ports are only connected to input ports. This is a quick superficial check. The latter requirement is described in the paper introducing XMAS [3]. In short it ensures a stable state can be reached between clock cycles, ensuring the data transferred is deterministic. A combinatorial cycle is a cycle of dependencies of $irdy$ and $trdy$ wires. Absence can be verified using a standard cycle detection algorithm for a directed graph [14].

[1] The source code is available at http://www.open.ou.nl/bvg/sym-xmas

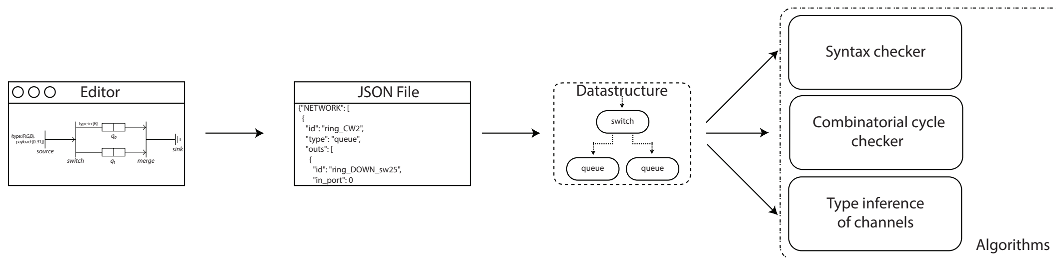


Fig. 4: The tool chain.

VII. Application to Spidergon

Using our `sym-xmas` tool, the symbolic types of channels in the Spidergon NoC were inferred. In our Spidergon example, the nodes of the first quadrant are slaves. All other nodes are masters. All packets have a payload of a 32-bits integer. In order to attain a notion of correctness, we keep the original injection point in the responses. This allows us to validate that the packets that are evacuated from the network are indeed correctly handled by the communication fabric. At the sinks in the network, the types inferred are stated below, with n the concrete value of the node number the sink is located in:

$$\left\{ \begin{array}{ll} dst \to [n\,..\,n] & colour \to \{response\} \\ src \to [n\,..\,n] & payload \to \left[0\,..\,2^{32}-1\right] \end{array} \right\}$$

This expression shows that node n only receives responses destined for n, that were originally injected at that same node. In other words, in the network requests and responses are correctly handled.

Actually during development of our algorithm, the version we analysed contained (without our knowledge) a wrong switching expression for the counter clockwise channel, resulting in packets misrouted to the wrong node. This error was easily detected and corrected.

The experiment was conducted using a single core on a 2 GHz Intel Core i7 2635QM running Mac OS X 10.9.2, with the gcc 4.8.2 compiler.

# Nodes	# Primitives	Time	Memory
4	70	0.008s	1.38 Mb
8	140	0.011s	1.63 Mb
16	280	0.024s	2.23 Mb
32	560	0.050s	3.65 Mb
64	1120	0.226s	7.64 Mb
128	2240	1.739s	19.79 Mb
256	4480	21.693s	65.68 Mb
512	8960	5m22.359s	221.32 Mb
1024	17920	92m31.848s	782.40 Mb

VIII. Discussion

We presented an algorithm that efficiently infers the type of all channels in large networks described in the xMAS language proposed by Intel. By 'channel type', we denote the information about which packet can traverse the channel. This information is key in the analysis of xMAS networks. Our algorithm produces at every sink, the set of packets which can possibly reach that sink. It is then relatively easy to check whether this conforms to the expectation of a designer. Inferring channel types is also needed, e.g., for the verification of deadlock freedom [7]. For large networks, the explicit simulation of all possible values is not feasible. For such networks, our algorithm provides a scalable alternative. An interesting direction is the automatic generation of Register Transfer Level (RTL) descriptions from xMAS networks. In this context, the knowledge of channel types is required to determine the bit-width of all wires.

References

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

[2] 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.

[3] ——, "xMAS: Quick formal modeling of communication fabrics to enable verification," *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.

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

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

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

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

[8] ——, "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] K. Richter, M. Jersak, and R. Ernst, "A formal approach to MpSoC performance verification," *Computer*, vol. 36, no. 4, pp. 60–67, April 2003.

[10] B. van Gastel and J. Schmaltz, "A formalisation of xMAS," in *ACL2*, 2013, pp. 111–126.

[11] M. Coppola, M. Grammatikakis, R. Locatelli, G. Mariuccia, and L. Pieralisi, *Design of interconnect processing units Spidergon STNoC*. CRC Press, 2009.

[12] A. Hansson, K. Goossens, and A. Rădulescu, "Avoiding message-dependent deadlock in network-based systems on chip," *VLSI Design*, 2007.

[13] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*. IEEE, 1998, pp. 9–15.

[14] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.