Burns F, Sokolov D, Yakovlev A.

A Structured Visual Approach to GALS Modelling and Verification of Communication Circuits.

***IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016)**

DOI: 10.1109/TCAD.2016.2611508

A Structured Visual approach to GALS Modelling and Verification of Communication Circuits

Frank Burns, Danil Sokolov, Alex Yakovlev

***Abstract*—In this paper a novel Globally Asynchronous Locally Synchronous (GALS) modelling and verification tool is introduced for xMAS circuits. The tool provides a structured environment for GALS in which organisation of the modelling and verification enables it to handle a variety of implementation tasks facilitating a process which would otherwise be difficult for the end user. The tool provides verification techniques at different levels. A new unfolding algorithm is presented that uses Structured Occurrence nets. A novel representation for deadlocks is introduced using deadlock relations enabling the causality of local and global deadlocks to be visualised. This helps in the investigation of total or partial system shutdown. In particular, the approach enables the visualisation of point-to-point causality of problems occurring between different parts of the system which are more difficult to analyse. In addition different types of deadlock related to the synchroniser can be detected. The work presented here provides structured visualisation capability facilitating the analysis of complex communication systems.**

I. Introduction

WHILST there has been a lot of interest in researching new architectures for GALS [1][2], there have been few attempts at providing modelling solutions for GALS communication. Thus, modelling of GALS from specifications has been limited to hardware description languages such as Verilog, VHDL [3] or synchronous programming languages such as C or ESTEREL [4]. Specialist verification languages that have been introduced for GALS include GRL [5] and process calculi [6] but these languages tend to be used at a higher level of abstraction than hardware. A graphical tool has been developed in [7] but the models here are also used at a higher level i.e. they are not used for circuit deadlock analysis. Although the techniques are higher level they offer better modelling of things like protocols. The work in [8] is more similar in the sense that different formats are interchangeable allowing different tools to be linked which is a useful approach to take but is centered on co-simulation rather than verification or deadlock analysis.

Hardware models for communication logic in the past have relied on standard languages, e.g. Verilog, which require a significant amount of ”glue logic” to connect communicating primitives together. This kind of modelling tends to be unwieldy and non-intuitive. xMAS [9][10][11] represents a significant improvement in the representation and modelling of communication systems. It provides a set of graphical communication primitives which are more natural, i.e. they are closer to the hardware, and their higher level of abstraction enables them to be easily understood.

Although xMAS model checking has been covered extensively at the Boolean level for purposes like deadlock checking [12][13][14][15][16] little work has been done using net level models such as Petri nets. In [17] basic techniques for GALS synthesis to Circuit Petri nets [18] for xMAS were presented offering some distinct advantages: they are well suited to the visualisation of distributed models of local machines in terms of concurrency and for verification they capture a complete knowledge in the unfolding hence providing a representation of the full causality. In [17] an additional xMAS synchroniser primitive was introduced to provide a synchronisation wrapper for synthesising a range of ”glue” solutions e.g. asynchronous, mesochronous, etc. Basic techniques for GALS verification were also presented including unfolding to occurrence nets.

The focus of this paper is on a novel structured visual approach to GALS modelling and verification. It provides a platform which integrates existing and advanced GALS techniques into a unified environment for the analysis and visualisation of complex communication systems. The intention of such an environment is to make it easier to gain visual insight into the causality of complex structural problems that may arise, such as deadlocks in GALS systems. The modelling and verification flow is shown in Fig. 1.

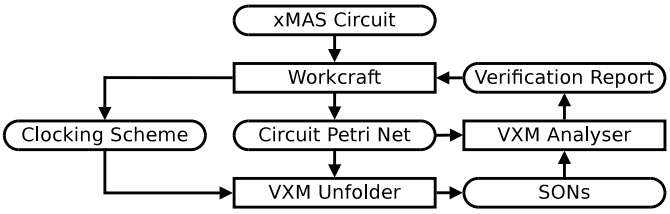


Fig. 1: xMAS tool flow.

Modelling is undertaken via a tool called WORKCRAFT [19]. WORKCRAFT is a framework for a variety of plugins that aid in the visualisation of different graphical interpretations but which are linked [20]. This enables translation and cross-visualisation between different models at different levels, e.g. xMAS, Circuit Petri-nets, and, therefore, the analysis of lower level models can be related back to the original model graphically. Inside WORKCRAFT xMAS is translated to Circuit Petri-nets and then a novel unfolding algorithm is deployed using an unfolder (VXM Unfolder) from Circuit Petri-nets to Structured Occurrence Nets (SONs) [21]. The causality of deadlocks can be difficult to analyse if significant parts of a GALS system become disabled particularly those using intricate feedback. SONs were designed to enable visualisation of such complex behaviours which are more difficult to analyse and they are amenable for the modelling of structural links between modules. This is useful particularly when one wants to

investigate point-to-point effects or how far the occurrence and effect of a problem extends between different modules.

A multi-level analyser (VXM Analyser) is used for verification. For this a novel formalism based on blocking/idle deadlock relations is introduced which describes how deadlocks in different parts of the system relate to each other. This representation, which is derived from the SONs model, enables more detailed structural visualisation of the deadlocks and their causality throughout the GALS communication system. Via feedback through WORKCRAFT it enables direct and indirect deadlocks related to synchroniser handshake and latency errors to be analysed. It enables structural analysis of deadlocks to be carried out across communication links to reveal the following details: vulnerable parts of the system which are susceptible to shutdown; point-to-point causes of deadlock from one local module to another (using querying); multiple original causes of deadlocks and their visualisation in a single instance or snapshot and difficult to detect deadlocks that are hidden or masked by other deadlocks.

The main contributions of this work are:

- A workflow for structured visual modelling and verification of GALS communication circuits.
- A new xMAS primitive for synchronisation of GALS modules.
- Automated translation of xMAS models to a Circuit Petri net representation using prioritisation.
- A novel unfolding algorithm of Circuit Petri nets to Structured Occurrence nets driven by synchronisation policy.
- A new approach to deadlock analysis based on deadlock relations.
- Analysis of deadlocks related to synchronisers.

The paper is organised as follows. In section II the xMAS model together with our WORKCRAFT tool are described. In section III we describe the modelling approach which includes translation from xMAS to Circuit Petri nets and modelling for synchronisation. In section IV novel verification procedures are introduced together with an algorithm for unfolding the Petri nets into Structured Occurrence nets for detailed deadlock analysis. In section V experimental results are provided, followed by conclusions in section VI.

II. xMAS MODELLING

A. xMAS Primitives

xMAS models are based on a set of communication primitives which have inputs and outputs and which can be glued together according to the equations which define them. The benefit of the equations is there is a clear distinction between the transformation applied to data and the logic coordinating the movement of data. There are eight communication primitives altogether and these are depicted in Fig. 2.

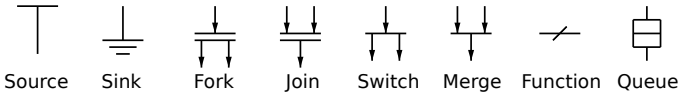


Fig. 2: xMAS primitives.

The Source and the Sink primitives are used for inputting and outputting information in the form of packets or tokens. These are the ports of the xMAS model which allow the model to be interfaced to its environment. The equations governing the Source and Sink are shown below

```
Source:
o.irdy = oracle or pre(o.irdy and not o.trdy)
o.data = e
Sink:
i.trdy = oracle or pre(i.trdy and not i.irdy)
```

The Source is parameterised by a constant expression e : α. Each cycle, it non-deterministically attempts to send a packet e through its output port o : α. The signals $irdy$ and $trdy$ stand for initiator ready to send and target ready to receive. In the equations **pre** is the standard synchronous operator that returns the value of its (Boolean) argument in the previous cycle and the value zero in the first cycle. This is to ensure persistency of the $irdy$ signal regardless of the oracle. The Source and the Sink have a number of different types of operation:

- $eager$ - always ready to send or receive packets;
- $dead$ - never ready to send or receive packets;
- $non-deterministic$ - the value of the oracle is set randomly.

The Fork and Join primitives are the basic synchronisation primitives. The equations governing the Fork and Join are shown below:

```
Fork:
a.irdy = i.irdy and b.trdy  a.data = f(i.data)
b.irdy = i.irdy and a.trdy  b.data = g(i.data)
i.trdy = a.trdy and b.trdy
Join:
a.trdy = o.trdy and b.irdy
b.trdy = o.trdy and a.irdy
o.irdy = a.irdy and b.irdy
o.data = h(a.data, b.data)
```

A Fork coordinates the input i and outputs a, b so that a transfer only takes place when the input is ready to send and the outputs are ready to receive. A Join primitive operates as the inverse of the fork in which the roles of the $irdy$ and $trdy$ signals are reversed.

The Switch and Merge primitives are used for routing and selection of packets or tokens through the xMAS circuit. The Switch primitive is governed by the following equations:

```
Switch:
a.irdy = i.irdy and s(i.data)
b.irdy = i.irdy and not s(i.data)
a.data = i.data  b.data = i.data
i.trdy = (a.irdy and a.trdy) or (b.irdy and
b.trdy)
```

Informally, the Switch applies s to a packet x at its input, and if $s(x)$ is true, it routes the packet to port a, and otherwise it routes it to port b.

The Merge primitive is used for modelling arbitration by selecting one packet among multiple competing packets.

```
Merge:
a.trdy = mg and o.trdy and a.irdy
b.trdy = not mg and o.trdy and b.irdy
o.irdy = a.irdy or b.irdy
o.data = a.data if mg and a.irdy
```

```
b.data if not mg and b.irdy
```

Requests for a shared resource are modelled by sending packets to a merge, and a grant is modelled by the selected packet. A local Boolean state variable mg is used to ensure fairness [9].

The Function primitives are used for representing functions. The xMAS equations for the function are shown below.

```
Function:
o.irdy = i.irdy    o.data = f(i.data)
i.trdy = o.trdy
```

In xMAS storage is implemented by queues. The equations for the queue are shown below.

```
Queue:
hd = if (o.irdy and o.trdy) then inc(pre(hd))
     else pre(hd)
tl = if (i.irdy and i.trdy) then inc(pre(tl))
     else pre(tl)
where inc(x) = if x=k-1 then 0 else x+1
      o.irdy = not qempty  i.trdy = not qfull
For j = 0 to k-1
  memj = if (i.irdy and i.trdy and j=pre(tl))
         then i.data else pre(memj)
```

The queue is characterised by a non-negative integer k that indicates the capacity of the queue. It has one input port i which is connected to the target end of a channel that is used to write data into the queue. Likewise the output of the queue is connected to the initiating end of the channel that reads data out of the queue. The elements in the queue are stored in an array called mem of size k. These are indexed by head (hd) and tail (tl) pointers used for reading and writing.

B. GALS Asynchronous Primitive

In addition to the standard xMAS symbols for all the basic primitives an asynchronous synchronisation primitive has been added. The primitive is used for inserting asynchronous "glue" components in communication channels that cross clock domains. The interface signals are defined using the xMAS format so that it can be interfaced to other xMAS primitives. A diagram showing the synchronisation primitive is shown in Fig. 3.

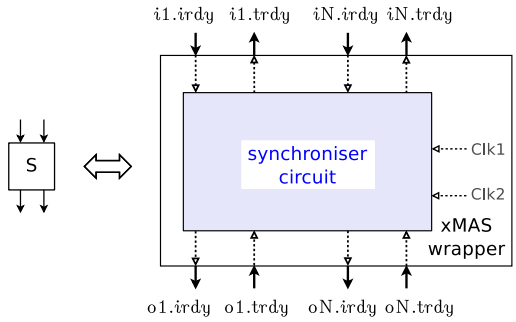


Fig. 3: xMAS synchronisation primitive.

A synchronisation primitive is used for communication between two islands. The synchronisation primitive accepts a variable number of send signals, $i1.irdy \,..\, iN.irdy$, from the incoming primitives from one island and returns the required number of receive signals, $i1.trdy \,..\, iN.trdy$. Similarly

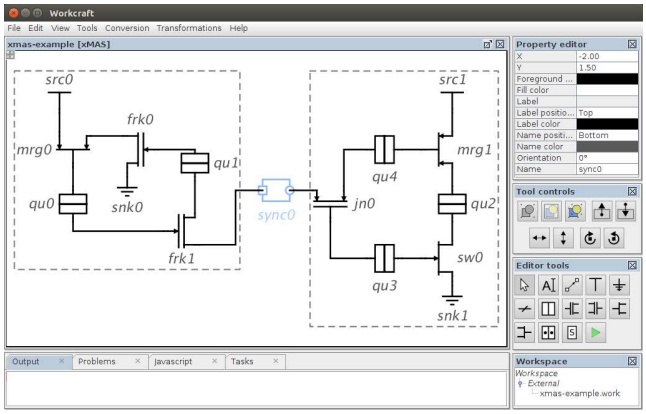


Fig. 4: WORKCRAFT tool - xMAS module.

it communicates with the target island by issuing the required number of send signals, $o1.irdy \,..\, oN.irdy$ and by accepting the required number of receive signals, $o1.trdy \,..\, oN.trdy$. The new asynchronous primitive is generic and incorporates a number of synchronisation schemes. A black box is used to house the specific implementation style used for synchronisation, which is designed to accommodate different GALS implementation styles: asynchronous, mesochronous, pausible clocking, etc. [22].

C. xMAS entry using WORKCRAFT

The tool that we use for graphical entry of xMAS diagrams is called WORKCRAFT. WORKCRAFT supports numerous models: Petri net, digital circuit, dataflow structures [23], occurrence nets, etc. and their interactivity. The framework has a plug-in driven architecture and supports run-time scripting making it a flexible and expandable environment. Its underlying Java technology provides cross-platform operation. WORKCRAFT provides an integrated framework which allows visual editing of xMAS models, their simulation, conversion into Circuit Petri nets and verification by external model checking tools.

Fig. 4 shows a screen dump of the WORKCRAFT tool in which the xMAS plug-in module is shown using graphical entry. Graphical entry allows for construction of the xMAS models using traditional mouse drag and drop. A selection of xMAS components is presented in the Editor tools panel on the right. Property settings are available in the xMAS module for assigning various model properties including: source initialization, setting the sizes of queues, assigning functionality to functions and setting the modes of operation i.e. (0) dead, (1) eager or (2) non-deterministic. A tool control panel is also provided for the organization of nodes into modules. This facility provides for the selection and grouping of xMAS components into distinct groups which is used for the creation of GALS modules. Menus are provided for Tools at the top which relate to different GALS tasks such as synchroniser selection and verification.

III. NET MODELLING OF XMAS CIRCUITS

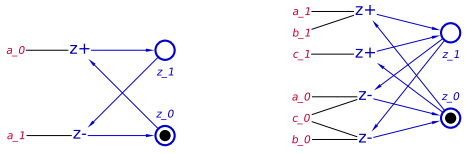
In this section a method of direct translation from the xMAS primitives to Circuit Petri nets is presented together with the

required execution semantics which is based on a method that uses prioritisation. Global communication is achieved by providing a selection of synchronisation primitives which are modelled using Circuit Petri nets, adjusted for synchronisation using prioritisation.

A. Circuit Petri net translation

Representing a logic circuit built from level-based components (i.e. logic gates) is achieved by so-called Circuit Petri nets [3]. A Circuit Petri net is a specific type of signal transition graph, in which each signal t is associated with two places t_0 and t_1 representing its two logic states. The enabling/firing semantics of the labelled $t+$ and $t-$ Petri net transitions, "corrected" through the appropriate labelling mechanism, adequately represents either AND or OR conditions in the logic. The actual "guards" for these transitions are formed by using read-arcs from the Petri net places associated with the state of the input signals to the gate. Historically, the transitions of input signals are drawn in red colour, output signals in blue and internal signals in green.

Fig. 5 shows simple examples for basic logic gates: an inverter and an AND-OR gate. The inverter has an input signal a with two possible states a_1 and a_0. These are connected to the inverting transitions $z-$ and $z+$ which are connected directly to the output states z_1 and z_0 . The correct signal changes are made depending on the corresponding transitions that are enabled and fired.



(a) Inverter. (b) AND-OR gate.

Fig. 5: Circuit Petri net examples.

Circuit Petri nets require an execution policy to enable the transitions to execute in the proper order for the circuit to function correctly. For this a parallel execution policy is employed in which enabled transitions are fired in parallel. Logic reduction is used to increase the natural parallelism within the representation. In this way the net representation of basic gates can be collapsed to effectively form combinations of gates. Fig. 5b shows the net representation of an AND-OR gate $z =$ `(a` **and** `b)` **`or`** `c`.

1) Primitive Translation: A description of the technique for translating xMAS primitives into Circuit Petri nets is now given for the communication control signals. The translation follows closely the xMAS primitives and is logically derived by reduction (optimization) from the xMAS equations. Each net primitive is comprised of basic signal nets (loops) corresponding to the variable assignments in the primitive equations and internal and external connections which provide the links. In addition external control signals are introduced by the system. The following diagrams provide the translation for some of the key primitives.

Fig. 6 shows the translation of the basic source primitive that is comprised of two signals, an internal $oracle$ and an output signal o_irdy. The $oracle$ and o_irdy signals are connected by internal read-arcs which allow the o_irdy signal to be enabled and disabled by the $oracle$. The resetting of the o_irdy signal depends on the external connection o_trdy. which is connected by a read-arc to the incoming signal from the receiving primitive that the source is sending to. The additional signal s is used for setting the mode for the oracle: dead (s_0 is marked), eager (s_1 is marked) or non-deterministic (both s_0 and s_1 are marked). The sink has a similar structure to the source in which the role of $irdy$ and $trdy$ signals are reversed.

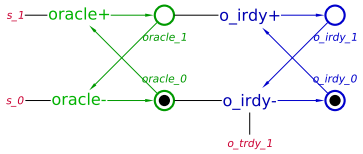


Fig. 6: Circuit Petri net translation - Source.

The fork, in Fig. 7, is comprised of three signals, a_irdy, b_irdy and i_trdy. External connections are shown to the ready i_irdy signal and the two incoming $trdy$ receive signals. The external receive signals are crossed as to align with the semantics of xMAS. The join net has a similar structure to the fork in which the roles of the $irdy$ and $trdy$ signals are reversed.

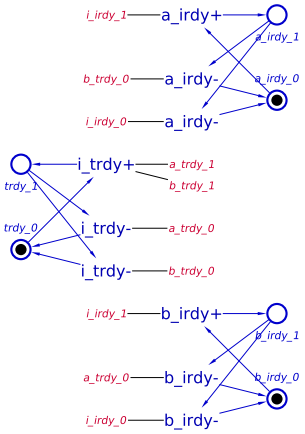


Fig. 7: Circuit Petri net translation - Fork.

Fig. 8 shows the translation of the basic switch primitive. The switch primitive is comprised of four signals, sw, a_irdy, b_irdy and i_trdy. The sw signal is connected to the internal $irdy$ signals by read arcs which are used to determine the selection of the a_irdy or b_irdy signals. The sw signal is data dependent and is sensitive to changes that occur in external data signals $s(i.data)$.

The merge primitive which is not shown uses a fair arbitration policy.

The queue primitive has the most complex representation. In Fig. 9, a diagram is shown for a queue of size 2. The queue uses a one hot representation. This is used for the head and tail pointers which are shown at the top and bottom of the queue body. The body of the queue comprises a number of slots, one for each queue entry. Each input slot is connected

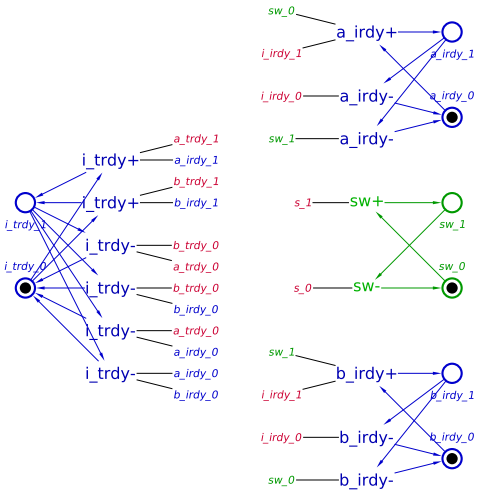


Fig. 8: Circuit Petri net translation - Switch.

to corresponding head and tail pointers. Each slot is also connected to the ready and receive signals (top). When a slot is full the $irdy$ signal is activated. When all the slots are full the $trdy$ signal is deactivated. The corresponding head and tail signals ensure that the correct slots are activated when the external $irdy$ and $trdy$ signals are activated. Together with the execution semantics this ensures the correct loading/unloading mechanism for the queue. The structure of the queue is generic across tl, mem and hd and can be scaled in size by adding additional sections.

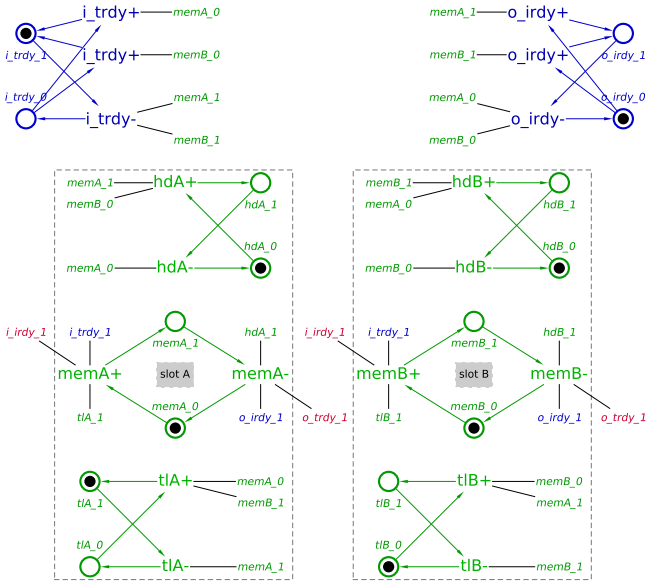


Fig. 9: Circuit Petri net translation - 2-stage Queue.

The Circuit Petri net translator is implemented inside our WORKCRAFT tool. The translator accepts a JSON (data-interchange format) representation of the xMAS model and translates it into a Petri net representation. For translation each primitive is generated as shown in the examples in Fig. 6 to Fig. 9. The net primitives are connected by a process which links together all external connections. A data line is added later automatically to include data signals. Data is treated as signals e.g. 0 .. 1. The signals pass through the data CPNs of the respective gates similar to the examples shown in Fig. 5.

2) Execution semantics: The translation of Circuit Petri nets from xMAS require prioritisation for them to operate effectively. A net executed in maximal parallel mode will not be executed properly according to the xMAS semantics [9]. Communication problems where this becomes a problem include the source and sink. In addition the firing of queues must be synchronised correctly otherwise the transfer of data may not occur in the intended order. For this reason the Petri nets are prioritised to generate the correct order for firing. Generally prioritisation of Petri nets adds priorities to transitions, whereby a transition cannot fire, if a higher-priority transition is enabled (i.e. can fire). For prioritisation, we use a system based on labelled Petri nets.

Formally a priority system is a pair of the form (Σ, Π) where Σ is the base non-prioritised system (Labelled Petri net) and Π a priority specification. For our base system the transition labels are divided into sets to distinguish the type of transitions corresponding to the basic xMAS primitives. Thus, our prioritisation system is one for whom Π is a binary relation on the actions of Σ based on the transition labelling type. For example, $Ta = Tb$ denotes that the set of transitions of label type Ta have equal priority to those of label type Tb. $Ta > Tb$ denotes that the set of transitions of label type Ta have a higher priority than those of label type Tb.

There are a number of rules around which prioritisation of transitions need to be made. In general, if the queues become enabled and other communicating transitions are enabled simultaneously, then the queue transitions should be stalled to allow the remaining communicating transitions to fire. Therefore, queue loading must be given a lower priority than communication signals which connect queues. Formally:

$$Tqload < (T - Tqload - Toracle) \tag{1}$$

where T is the set of all transitions, $Tqload$ is the set of transitions associated with loading/unloading of the queue slots and $Toracle$ is the set of source and sink oracle transitions.

Proposition 1: There is a logical equivalence between the CPN primitives and xMAS primitives [equivalence is by truth table]

Proposition 2: If $\zeta = enabled(Tqload)$ then each $t \in \zeta$ must fire in the same step (clock step) as they have a unique common priority level.

Lemma1: If $t \in \zeta$ fires in a single clock step $\equiv$ (read/write) [32] and $[\chi = (T - Tqload - Toracle)] > Tqload$ then the execution semantics of (1) are equivalent to the clocked xMAS execution semantics.

Proof: Given χ = communication transitions which execute between clock steps (read/writes) - similar to the $transitionsislands$ of [32]. If $\chi > Tqload$ then $Tqload$ cannot fire until all χ have fired even if they are enabled. Therefore, χ will fire in a different step to $Tqload$. If proposition 1 and proposition 2 hold it follows due to the alternating firing of χ and $Tqload$ that the execution semantics of (1) are equivalent to the xMAS execution semantics.

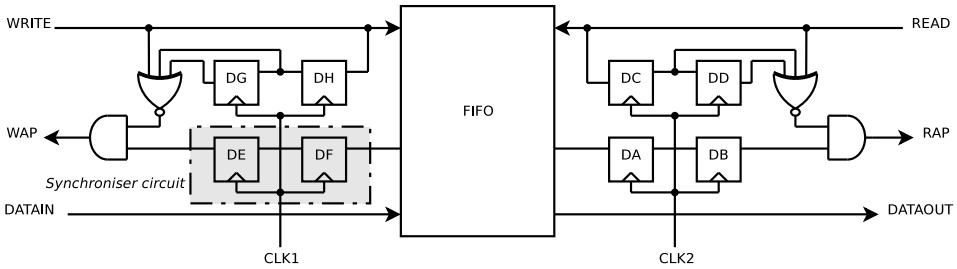


Fig. 10: Asynchronous synchronisation.

In addition the source and sink oracles must occur as a multiple of queue transfers. For this the following relation is required:

$$Toracle < (T - Tqload - Toracle) \tag{2}$$

which assigns the oracles a lower priority than all signals apart from the queue: if the source and sink are eager they are activated once at the start via the system control signals (see Fig. 6); if they are non-deterministic then according to (2) they can only be activated when the communication signals other than the queue have already been activated.

Lemma 2: If each $t \in enabled(Toracle)$ fires in a single clock step and $[T - Tqload - Toracle] > Toracle$ then the execution semantics of (2) are equivalent to the xMAS execution semantics.

Proof: the proof follows from lemma1 with $Tqload$ switched with $Toracle$ i.e. the same must hold for the sources and sinks as holds for the queues.

Finally a specific mapping $\Pi = \Pi_{QO}$ is required which relates queue to source and sink oracles:

$$\Pi_{QO} = \begin{cases} Toracle = Tqload & \text{if eager} \\ Toracle \leq Tqload & \text{if non-deterministic} \end{cases} \tag{3}$$

Here Π_{QO} represents a priority mapping relation between the queue and the source and sink oracles. The first part of the expression operates in a similar manner to (2) with regards initialisation i.e. the oracles are only activated once at the start if they are eager. The second part of the expression in (3) dynamically sets the prioritisation of the oracle to be less than or equal to the queue depending on the non-deterministic setting that is generated by the system for each oracle.

Theorem 1: If Case 1: $Toracle = Tqload$ (the setting of the oracle is eager) or Case 2: $Toracle \leq Tqload$ (the setting of the oracle is non-deterministic), then for either case, Case 1 or Case 2, the execution semantics must be equivalent to the xMAS execution semantics.

Proof: Case 1: The proof follows from lemma 1 and lemma 2. Case 2: $IfToracle < Tqueue$ then each $t \in Toracle$ for which this holds must be inactive when any $\tau \in Tqload$ is active. This means that the χ islands between the inactive $Toracles$ and queues themselves will become inactive until a dynamic change of $Toracle$ occurs to $[Toracle = Tqload] \equiv$ Case 1. This behaviour is equivalent to the non-deterministic execution sematics of xMAS.

B. Synchroniser modelling

For modelling synchronisers [24] in WORKCRAFT the user connects the communicating GALS modules by means of synchronisation primitives and subsequently from a selection menu chooses the implementation style for each synchroniser. This enables the user to make a decision with regard the internal details based on the GALS style that is required. The GALS style is chosen from a selection of available GALS implementation schemes [25]

The basic synchroniser schemes provided by the tool are as follows: $asynchronous$ - an implementation based on the use of synchronisers to transfer signals arriving from an outside timing domain to the local timing domain e.g. two flip-flops to synchronise signal with local clock; $mesochronous$ - an implementation in which clocks are derived from the same source and the bounds on the frequencies of communicating blocks are exploited to meet the timing requirements; $pausible$ - an implementation based on ring oscillators in which each locally synchronous block generates its own clock with a ring oscillator.

An implementation style that is provided for the asynchronous scheme is shown in Fig. 10. The implementation in Fig. 10 uses a FIFO and synchroniser circuits to transfer signals between the global timing domain and the local timing domain. In this implementation the FIFO buffer handshake signals may be asserted at any time relative to the transmitter or receiver clocks. The implementation uses two flip-flops to synchronise a signal with the local clock. To account for the synchronisers delay, the wait signal generated by the gates prevents the transmitter from sending until the FIFO buffer status following the previous write operation has propagated through the synchroniser.

For each synchroniser a net model is provided which is based on the circuit implementation. The net model is derived from the basic circuit using Circuit Petri nets but the synchroniser flip-flops are modelled as prioritized signals. The net model used for a synchroniser circuit which uses two flip-flops is depicted in Fig. 11.

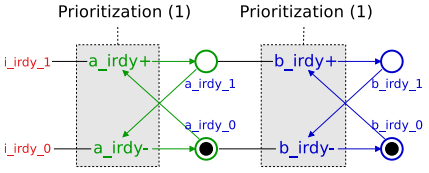


Fig. 11: Synchroniser circuit.

In Fig. 11, for the pair of signals comprising the synchroniser circuit, the transitions have a level of prioritization which is similar to the level that was assigned to the queue loading in (1). This is required in order to give the flip-flops a lower priority than the other communication signals.

The synchroniser is used to synchronise the asynchronous communication signal with the local clock. The synchroniser circuit is designed to protect the communication signal when it synchronises with the clock from metastability errors. If the synchroniser and clock edges arrive too close together the synchroniser can become metastable with a probability which is related to Mean Time Before Failure (MTBF) [24]. For the two-flop synchroniser this could result in the addition of 1-clock cycle to the latency.

To minimize the failures due to metastability in asynchronous signal transfers, a sequence of two or more registers (a synchronization register chain or synchronizer) is typically used in the destination clock domain to resynchronize the signal to the new clock domain. Adding registers allows additional time for a potentially metastable signal to resolve to a known value before the signal is used in the rest of the design. The timing slack available in the synchronizer register-to-register paths is the allowed time available for a metastable signal to settle, and is known as the available metastability settling (resolution) time. Different designs of sychroniser are, therefore, possible based on the available resolution time in clock cycles.

For each implementation style details of the clocking are entered by the user. Inside the tool menus are provided which allow the clocking details to be modified for each synchroniser. Frequencies are set as relative values to reflect changes across module boundaries. The clocking details entered are used later in the unfolding algorithm. Potential synchronisation problems due to metastability are exploited in the unfolding by varying the clock by 1 clock-cycle. This is used as a margin of error for the two-flop synchroniser to investigate the effect of a change in the latency.

IV. Verification

The xMAS models are verified by a process of unfolding to occurrence nets and deadlock analysis.

The existing unfolding algorithms [26], [27], [28] are based on asynchronous semantics of the Petri nets. The notion of prioritisation introduced for xMAS models expressed in Circuit Petri nets requires a fundamentally different unfolding strategy that is driven by policies [29].

The unfolding proceeds in a parallel manner based on the execution semantics of the xMAS model. For normal verification as used in [17] the unfolding proceeds to occurrence nets using basic GALS unfolding. For structured net verification the net is translated from xMAS but the unfolding is made to Structured Occurrence nets rather than Occurrence nets [21]. Structured Occurrence nets are designed for modelling complex systems which can be partitioned into subsystems and are more amenable for modelling the GALS communication structure.

A Structured Occurrence net (SON) is a set of related occurrence nets linked together by specific types of relation [30]. The type of relation determines the class of the SON. Communication Structured Occurrence nets (CSONs) are a basic class of Structured Occurrence nets introduced in [21]. For CSONs The individual occurrence nets are linked together by communication relations in the form of communication channels.

The communication Structured Occurrence nets are limited to 1-safe communication only. 1-safe implies a limit of one token per place in the CPN which is a necessary requirement for CPNs. A Communication Structured Occurrence net is a tuple $CSON = (ON_1, ..., ON_k, P'_0, l'_0, F'_0)$ consisting of k occurrence nets where $k \geq 1$ such that each $ON_i = (P'_i, T'_i, F'_i, l'_i)$ is an occurrence net, P'_0 is a set of channel places linking the occurrence nets together (the communication occurs across the channel places), l'_0 is a labelling of P'_0 and a flow relation $F'_0 \subseteq (T' \times P'_0) \cup (P'_0 \times T')$, where $T' = \bigcup_{i \geq 1} T'_i$.

A. Unfolding

For unfolding the GALS model is mapped to Structured Occurrence nets and the local modules L_N are mapped to ordinary occurrence nets. The GALS unfolding enables mapping to the $CSON$ model by assigning occurrence nets to divisions corresponding to local module boundaries; occurrence nets are generated automatically for each local module and the individual ONs are subsequently connected using communication channels.

The nets are mapped in a specific way to reflect the unique GALS structure. In the unfolding process labelled events are assigned accordingly; all labelled events that belong to each local module L_N are assigned to an occurrence net ON_N and all labelled events that belong to each $s \in S$ are assigned to ON_S. The $irdy$ and $trdy$ control signals corresponding to the ports of the ONs are linked directly to channel places. Similarly, the $irdy$ and $trdy$ control signals of the synchroniser wrapper of Fig. 3 are linked to the corresponding channel places. This is depicted in Fig. 12 in which two local modules L_A and L_B are mapped to ON_A and ON_B respectively and connected to ON_S via channel places (CP).

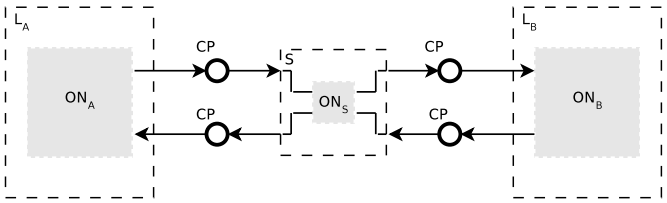


Fig. 12: CSON diagram.

The unfolding algorithm for the structured nets is shown below. The first few lines deal with the initialisation. In the algorithm the SEED refers to the initial conditions. Corresponding to Fig. 12 the prefix is pre-partitioned into two segments ON_L and ON_S. For the unfolding algorithm prioritisation is used to adjust for synchronisation between queues (see section III). All enabled transitions are ordered according to a priority queue based on their priority level. If the priority is higher they appear first in the queue. All transitions with the highest priority are processed first.

As the unfolding proceeds the events are assigned to their partitions respectively; the events e_L of all local transitions of t_A are assigned to ON_L and the events of all communication transitions e_S are assigned to ON_S. Any s $\in$ postset corresponding to $syncIO$ are converted to channel places. Here $syncIO$ refers to synchroniser inputs and outputs. Postset refers to output conditions (places) of events (transitions). Cut-off refers to events which when fired take the net to a marking that has already occurred from a previous firing of an event [26]. For synchronisation of clocking domains the unfolding is continued across the clock domain; the unfolding of the local partitions is configured according to the synchronisation style. For different clock domains the unfolding in each ON_L is set at different rates according to the rate of queue firing. This is set in the unfolding algorithm by controlling the addition of the xMAS queue transitions which are clock sensitive at different rates according to the relative frequencies of the local partitions in which they reside (line 19). This also holds for the signals inside the synchronisers which are clock sensitive.

Algorithm 1 Unfolding algorithm

```
1:  Add the conditions in the SEED to the prefix
2:  Initialise the priority queue q with the events possible in
    the SEED
3:  Initialise the cut-off set to φ
4:  Partition prefix → ON_L, ON_S
5:  while q ≠ φ do
6:      sort the queue in order of priority (highest first)
7:      for each transition t_A in the highest priority set do
8:          if cut-off is detected then
9:              cut-off ← cut-off ∪ e
10:         else
11:             if t_A is a non-sync transition then
12:                 Add corresponding event e_L and postset to ON_L
13:             else
14:                 Add corresponding event e_S and postset to ON_S
15:             end if
16:             if  s ∈ postset corresponds to syncIO then
17:                 convert s to channel place
18:             end if
19:             Check valid frequency sensitive transitions with
                reference to clock domain (queue loading)
20:             Insert new possible events into the queue
21:         end if
22:     end for
23: end while
24: add the postsets of all cut-off events to the prefix
```

B. Deadlock analysis

From the unfolding deadlock analysis proceeds. Deadlock checking is made at the communication level and modular level by analysis of the CSON model and localised occurrence nets. The deadlock checking uses traditional global verification deadlock analysis techniques as are used for general Petri nets. Local xMAS deadlocks [31] are also analysed at this stage. In [31] the concept of a local deadlock was introduced in terms of dead channels in which they define the concept of local deadlock based on sections of the model that become permanently inactive. These types of deadlock are split into two different types: **blocking** where $irdy$ signals become permanently inactive and **idle** where $trdy$ signals become permanently inactive. For GALS modules local deadlock analysis is made from the occurrence net for each module. Here the checks for local deadlock are restricted to queue blocking where each queue is checked in the occurrence net to see if local blocking has occurred in a specific module.

More specifically, in the occurrence net each event contains a record of which step it occurred in. From the set of steps $S \in steps(CSON)$ a record of all cut-off points $c \in C$ is recorded which occurs due to those events firing in step $s \in S_C$ where $S_C \subset S$ corresponds to all of the concluding steps which precede cut-off. Each cut-off c associated with step s corresponds to some marking $m \in M$ which relates to an earlier marking m' in the net which has already occurred due to the execution of events in some prior step s' where $s' \prec s$.

Analysis of local blocking proceeds by analysing the sequence of steps $s'..s$ by checking if any queues are inactive in such a sequence. This is determined by the activation of the labelled events of the queues, in the occurrence net, that are active in those steps from s' to s due to the marking of the corresponding conditions associated with the events. For this to be the case the corresponding condition in (4) must hold.

$$\forall s_l \in steps(s'..s) \;\; \nexists(cond = true) \tag{4}$$

where

$$cond = (i.trdy \wedge i.irdy) \vee (o.irdy \wedge o.trdy) \tag{5}$$

Here, the $irdy$ and $trdy$ signals correspond, respectively, to the send and receive queue signals in Fig. 9. If the condition in (4) holds and there is no escape from the sequence $s'..s$ i.e. to some other path where the condition in (5) holds then the condition specified in (4) is used to validate local blocking of the queues. A diagram depicting this in an example snippet is shown in Fig. 13.

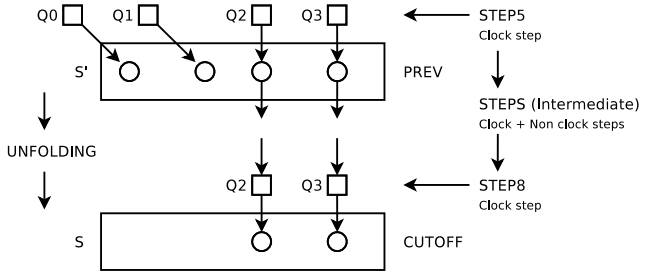


Fig. 13: Local blocking detection.

In Fig. 13 STEP 5 represents an earlier marking which corresponds to the cut-off in STEP 8. The pseudo-code showing the steps for this is shown in Algorithm 2. In the algorithm to check for a queue loading event the corresponding condition in (4) must hold between steps s' and s. When

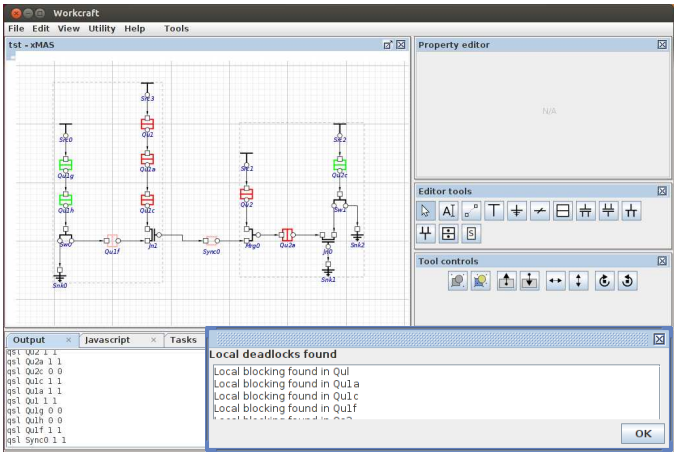


Fig. 14: WORKCRAFT tool - detecting local deadlocks.

this is applied to the example snippet of Fig. 13, $Q2$ and $Q3$ which are activated between STEPS 5 and 8 are assigned to $qstore$ (line 6, Algorithm 2) which means they are recorded as not being blocked between these steps. After processing $q \in (Tqload - qstore)$ (line 11, Algorithm 2) the value $qdlstore$ contains the queues that are locally inactive and which are locally deadlocked.

Algorithm 2 Deadlock detection

```
1:  qstore=φ, qdlstore=φ
2:  step=PREV
3:  while step ≠ cutoff do
4:    if step == clockstep then
5:      for each queue load event e_L occuring in step do
6:        qstore ← qstore ∪ e_L
7:      end for
8:    end if
9:    step++
10: end while
11: for each q ∈ (Tqload - qstore) do
12:   if nobranch occurs from PREV to cut-off then
13:     qdlstore ← qdlstore ∪ q
14:   end if
15: end for
```

If only local deadlocks are found this is reflected back to the user in the form of local blocking messages to highlight the queues that are locally blocked/idle. Fig. 14 shows an example where local blocking has been found in the queues which has been caused by incorrect routing of packets at the switches.

For full deadlock, traces are generated in the WORKCRAFT tool to reflect the sequence of transitions leading to deadlock. Full traces are generated representing all details as well as specific traces which signify the changes happening in the queues which lead to deadlock.

These can be traced directly by simulating traces from the deadlock analysis to find the source of the deadlock. Deadlock relations, described below, are compact representations which offer an alternative representation of the causality.

C. Deadlock relations

Using the CSON model it becomes possible for the verification analysis to be applied in a more structured way by using deadlock relations. The advantage of deadlock relations is they are more compact and they can be used inside the tool to relay critical information to the user in the form of statements about the type and causality of the blocking i.e. which queue is the source of the blocking for another queue in a particular module. Deadlock relations can be specified either locally or globally.

Deadlock relations can be defined in terms of queue blocking or idleness. A queue which is found to be blocked in local module L_A may cause a queue to be blocked in L_B. Correspondingly a queue which is found to be idle in local module L_A may cause a queue to be idle in L_B. But which way this occurs depends on the flow of events in the corresponding occurrence nets which lead to the queues being blocked or idle. The following definitions introduce deadlock relations for local queue blocking and local queue idleness.

Definition 1: A blocking deadlock relation occurs locally between two queues on the same path in module L_A if a queue $q1^{L_A}$ is blocked thereby causing a queue that precedes it $q2^{L_A}$ to be blocked. This relation is expressed as follows $q2^{L_A} \xleftarrow{B} q1^{L_A}$.

Definition 2: An idle deadlock relation occurs locally between two queues on the same path in module L_A if a queue $q1^{L_A}$ is idle thereby causing a queue that follows it $q2^{L_A}$ to be idle. This relation is expressed as follows $q1^{L_A} \xrightarrow{I} q2^{L_A}$.

The deadlock relations are derived directly from the causality in the CSON model. The CSON model can be searched for specific localities where queues are blocked and in which occurrence nets as described in B. The connectivity between queues determined from the net via the $irdy$ and $trdy$ signals together with the blocking order can then be used to establish which queues cause which queues to be blocked. For the blocking order, with respect to some path in which the condition in (4) holds for a queue, the final step $s_f \prec s'$ in which condition (5) holds is used to determine the blocking order. Deadlock relations derived from the causality established from the occurrence net may also be used to link activity across communication channels from one L_N to another. For the GALS models the process can be extended to analyse which queue in a local module causes blocking or idleness in a synchroniser. The following definitions introduce the different types of deadlock relations for synchronisers.

Definition 3: A blocking deadlock relation occurs between a synchroniser S and queue that precedes it in module L_A connecting the queue if the synchroniser is blocked thereby causing the connecting queue $q1^{L_A}$ to be blocked. This is expressed using the relation $q1^{L_A} \xleftarrow{B} S1$. The reverse relation of this can be expressed using $S1 \xleftarrow{B} q1^{L_A}$.

Definition 4: An idle deadlock relation occurs between a synchroniser and a queue in module L_B that follows it connecting the synchroniser if the synchroniser is idle thereby

causing the connecting queue $q1^{L_B}$ to be idle. This is expressed using the relation $S1 \xrightarrow{I} q1^{L_B}$. The reverse relation of this can be expressed using $q1^{L_B} \xrightarrow{I} S1$.

Deadlocks related to the synchroniser can be split into two types: (i) direct i.e. the deadlock is due to a synchroniser handshake failure. This can be caused by an error in the synchroniser or its environment due to handshake problems [33]. (ii) indirect: i.e. timing problems due to the latancy. This is a result of setup time and metastability problems which can result in latency mismatch and subsequent functional errors in the adjoining modules.

The above relations can be chained together. The following equations show examples of chained relations. Equation (6) shows a deadlock relation between a synchroniser $S0$ and its two connecting queues $Q1$ and $Q2$ from local modules L_A and L_B. Equation (7) shows an internal local blocking relation between queues $Q2$ and $Q3$ in module L_B, in conjunction with blocking relations between the synchroniser $S0$ and corresponding local connecting queues.

$$q1^{L_A} \xrightarrow{I} S0 \xrightarrow{I} q2^{L_B} \tag{6}$$

$$q1^{L_A} \xleftarrow{B} S0 \xleftarrow{B} (q2^{L_B} \xleftarrow{B} q3^{L_B}) \tag{7}$$

The following definitions are used to define deadlock relations for queues which are connected on the same path.

Definition 5: A *bde* is a set of queues connected via the same communication path in which contiguous communicating queue pairs exhibit blocking deadlock relations.

Definition 6: An *ide* is a set of queues connected via the same communication path in which contiguous communicating queue pairs exhibit idle deadlock relations.

Equation (6), above, is an example of an ide relation and equation (7) is an example of a bde relation.

Using the deadlock relations a relational map is generated to show complete instances of deadlock activity inside the model. This is achieved by deriving all the deadlock relations using the CSON model to analyse the activity across the channel links and internally inside the local modules. This is expressed in terms of sets of blocking bde equations and idle ide equations. A complete set of bde and ide equations is generated by the analyser.

Indirect relations can also be formed between ide and bde providing relational links between blocking and idle paths. Here the queues on an ide and bde may not be in direct communication with each other but may be influenced by the communication links between. The causality between an ide and a bde is established by analysing the corresponding cross-communication links via the net. Using this information it is possible to analyse a number of unique solutions and trace the set of the original source(s) of the deadlocks.

Applying the relational model it becomes practicable to query the effects between different queues and synchronisers. The querying process uses transitivity to establish links between specific queues. Transitivity may be applied to equation (6), for example, to produce equation (8), reflecting the relation between $q1^{L_A}$ and $q2^{L_B}$:

$$q1^{L_A} \xrightarrow{I} S0 \cdot S0 \xrightarrow{I} q2^{L_B} \implies q1^{L_A} \xrightarrow{B} q2^{L_B} \tag{8}$$

Hence, it becomes possible using the relational model to query directly whether a queue in one local module causes another queue in another local module to be blocked across a particular communication link. This is particularly important when querying point-to-point causality.

V. Experiments

For the experiments a number of different xMAS circuits were tested. These were split into three example types, communication $COMM$, local blocking $GLOC$ and mesh $MESH$ examples. For each of the examples deadlock verification was applied and deadlock relations were derived leading to the establishment of relational information providing point-to-point and querying feedback.

To limit the verification effort the experiments were conducted using a mixed mode consisting of eager and non-deterministic. In this mode the sources were varied between eager and non-deterministic. This mode is significant because it is faster than full non-deterministic which in conjunction with a non-deterministic limit generates a more efficient unfolding leading to faster verification in which the information can be found more efficiently.

For the different types of experiment the following parameters were varied: the queue number qn, the number of sources in, the number of synchronisers sn and the number of xMAS primitives xn. For each experiment the basic time was estimated in seconds it takes to find the presence of deadlock(s).

The results of the verification are shown in tables I-III. Each table is split vertically in two main sections, one for basic information pertaining to deadlocks and the other showing details which includes relational information. The results from the relations section are shown in terms of the number of blocking and idle queues $\langle bd, id \rangle$, the number of blocking and idle equations $\langle bde, ide \rangle$, the number of original sources of deadlock src and the total time required to calculate all basic and relational information. Each table is split horizontally into two halves to show the comparison between results for two different queue sizes $k = 2$ and $k = 3$.

A. Communication examples

The first set of examples are communication examples comprising communicating agents with varying types of communication i.e. asynchronous and mesochronous.

The first example COMM1 is taken from the circuit in Fig. 15. The xMAS model comprises communicating agents split into two parts, comprising local module L_A and local module L_B, across a local asynchronous division. In this example one-way communication is used in which information is passed from one agent to the other. Here a local block

TABLE I: xMAS Verification Results 1

			Basic					Rel					
k	Example	type	qn	in	sn	xn	ql	bd	id	bde	ide	src	time (sec)
2	COMM1	asynch	7	2	1	18	7	4	4	2	1	1	0.289
	COMM2	asynch	16	4	2	30	16	10	8	4	2	2	0.241
	COMM3	asynch	14	3	1	29	14	9	6	5	2	1	0.417
	COMM4	mesoch	19	3	4	42	19	13	10	4	3	2	1.131
	COMM5	mesoch	28	3	4	51	28	18	14	5	3	4	1.928
3	COMM1	asynch	7	2	1	18	7	4	4	2	1	1	0.779
	COMM2	asynch	16	4	2	30	16	10	8	4	2	2	0.463
	COMM3	asynch	14	3	1	29	14	9	6	5	2	1	1.018
	COMM4	mesoch	19	3	4	42	19	13	10	4	3	2	2.546
	COMM5	mesoch	28	3	4	51	28	18	14	5	3	4	3.740

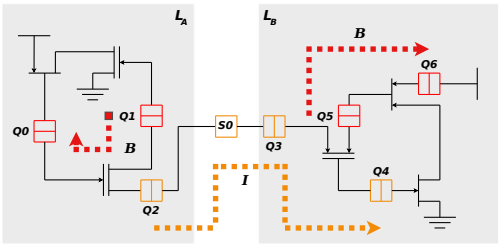


Fig. 15: COMM1 example.

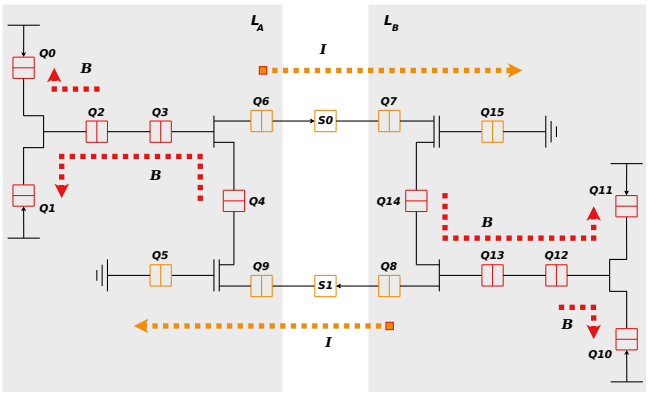


Fig. 16: COMM2 example.

occurs in L_A causing L_A to shutdown. As a consequence no information can be transmitted through the synchroniser. This in turn causes a shutdown in L_B.

The equations depicting the deadlock relations are shown below.

$$q0^{L_A} \xleftarrow{B} q1^{L_A} \tag{9}$$

$$q2^{L_A} \xrightarrow{I} S0 \xrightarrow{I} q3^{L_B} \xrightarrow{I} q4^{L_B} \tag{10}$$

$$q6^{L_B} \xleftarrow{B} q5^{L_B} \tag{11}$$

In COMM1 the source of the deadlock problem is $q1^{L_A}$ which causes L_A to deadlock which subsequently causes the synchroniser and L_B to deadlock. The results of the experiment are shown in table I. The relational information is shown on the right.

The second example COMM2 is taken from the xMAS model in Fig. 16. This is a model of two interacting agents, using two-way communication, which pass information between each other. The two agents are linked together using two asynchronous synchronisation units.

Here a local block occurs in L_A causing L_A to partially shutdown. As a consequence no information can be transmitted through the synchroniser $S0$ causing L_B to partially shutdown. Simultaneously a local block occurs in L_B this causes a block via synchroniser $S1$. This in turn results in a complete shutdown in both L_A and L_B.

The equations for the deadlock relations are shown below.

$$q1^{L_A} \xleftarrow{B} q2^{L_A} \xleftarrow{B} q3^{L_A} \xleftarrow{B} q4^{L_A} \tag{12}$$

$$q0^{L_A} \xleftarrow{B} q2^{L_A} \tag{13}$$

$$q6^{L_A} \xrightarrow{I} S0 \xrightarrow{I} (q7 \xrightarrow{I} q15)^{L_B} \tag{14}$$

$$q11^{L_B} \xleftarrow{B} q12^{L_B} \xleftarrow{B} q13^{L_B} \xleftarrow{B} q14^{L_B} \tag{15}$$

$$q10^{L_B} \xleftarrow{B} q12^{L_B} \tag{16}$$

$$q8^{L_B} \xrightarrow{I} S1 \xrightarrow{I} (q9 \xrightarrow{I} q5)^{L_A} \tag{17}$$

From a query the two sources of deadlock are $q6^{L_A}$ and $q8^{L_B}$.

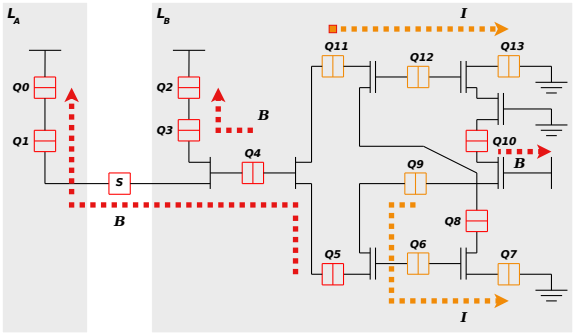


Fig. 17: COMM3 example.

The third example COMM3, in Fig. 17, is based on deadlock due to timing mismatch issues caused by a synchroniser.

TABLE II: xMAS Verification Results 2

			Basic					Rel					
k	Example	type	qn	in	sn	xn	ql	bd	id	bde	ide	src	time (sec)
2	GLOC1	mesoch	4	3	1	15	3	4	0	2	0	1	0.081
	GLOC2	mesoch	18	2	2	32	8	3	6	1	2	1	0.407
	GLOC3	asynch	14	4	4	34	8	3	7	3	2	4	0.312
	GLOC4	mesoch	23	9	7	67	12	8	8	4	4	8	0.409
3	GLOC1	mesoch	4	3	1	15	3	4	0	2	0	1	0.143
	GLOC2	mesoch	18	2	2	32	8	3	6	1	2	1	0.817
	GLOC3	asynch	14	4	4	34	8	3	7	3	2	4	0.473
	GLOC4	mesoch	23	9	7	67	12	8	8	4	4	8	1.083

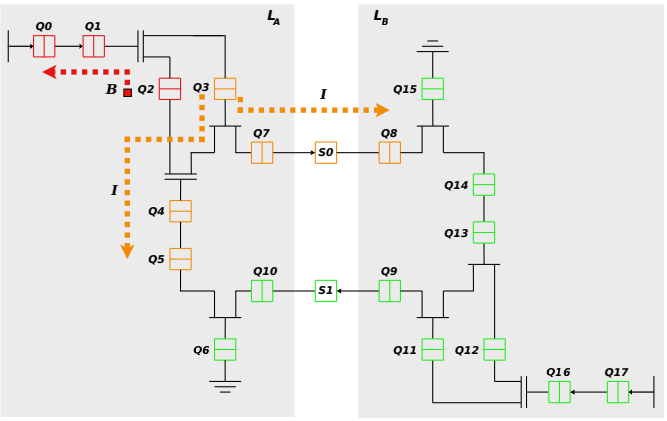


Fig. 18: GLOC2 example.

Module L_A communicates with L_B accross an asynchronous channel. L_B merges its own internal source with the incoming stream from L_A and a switch is used to filter all external packets upwards and all native packets downwards. All sources in the example are eager. The circuit on the right requires a specific relative timing between the information flows to operate properly. Specifically the feedback from $q8^{L_B}$ and $q9^{L_B}$ are used to limit the upward and downward packet flow so that the upward and downward transfers become balanced. Queue $q4^{L_B}$ represents a common channel. Due to the setup and resolution time (associated with the MTBF) for the synchroniser being larger than the restricted flow limit will allow, the common channel as a consequence will sequence too many native packets. Thus, when $q9^{L_B}$ is emptied this channel becomes blocked. If the synchroniser is removed or replaced by an ordinary queue the balance requirements of the circuit are met so it will operate according to the flow requirements. The deadlock is indirectly caused by the synchroniser due to latency problems with setup and resolution time resulting in downstream functional errors. The circuit works as described for the fair arbitration policy that has been adopted for the merge primitive when it is functioning in eager mode. However, the circuit is time dependent and will, therefore, react differently based on the selection of the synchroniser; whether a deadlock occurs is directly dependent on the type and design of synchroniser i.e. substituting a mesochronous synchroniser in the above circuit will not result in a deadlock for an MTBF margin of error of 1 cycle. This is because of the difference in mechanisms due to the setup and resolution times of the different synchronisers. This means a specific choice or design of a synchroniser may be used, to limit a particular type of deadlock such as this, so long as it adheres to the correct timing requirements of the circuit. The actual choice of synchroniser is dependant on the design and flow requirements of the particular circuit.

The remaining examples in table I, COMM4-COMM5, show the results for a number of larger communication examples. These examples show a corresponding increase in time as the number and sources of deadlock increases.

B. *Local deadlocking examples*

The second set of experiments are examples in which the GALS modules are structurally designed so that different kinds of structural local deadlocks are generated internally inside the GALS modules. The deadlocks are not generated by loops but occur locally inside each module due to local structural blocking which results in partial shutdown. The results of the experiments are shown in table II.

The GLOC2 example shown in Fig. 18 shows a communication example of two interacting agents communicating, using two-way communication, which pass information between each other. The two agents are linked together using two mesochronous synchronisation units.

Here a local block occurs in L_A causing L_A to partially deadlock. As a consequence no information can be transmitted through the synchroniser $S0$. This deadlock only causes a partial shutdown in L_B and L_B remains largely operative. As a consequence synchroniser $S1$ remains operational and information still flows from L_B to L_A.

The equations depicting the deadlock relations are shown below.

$$q0^{L_A} \overset{B}{\leftarrow} q1^{L_A} \overset{B}{\leftarrow} q2^{L_A} \tag{18}$$

$$q3^{L_A} \overset{I}{\rightarrow} q4^{L_A} \overset{I}{\rightarrow} q5^{L_A} \tag{19}$$

$$(q3 \overset{I}{\rightarrow} q7)^{L_A} \overset{I}{\rightarrow} S0 \overset{I}{\rightarrow} q8^{L_B} \tag{20}$$

From a query, using the querying option available inside the Workcraft tool, the origin of the deadlock is $q2^{L_A}$ which

is blocked which in turn causes $q3^{L_A}$ to be idle due to the cross-communication link via the fork. The parts of the system which remain active are highlighted in green. In the verification tool it is possible to query any of the relational effects between the queues.

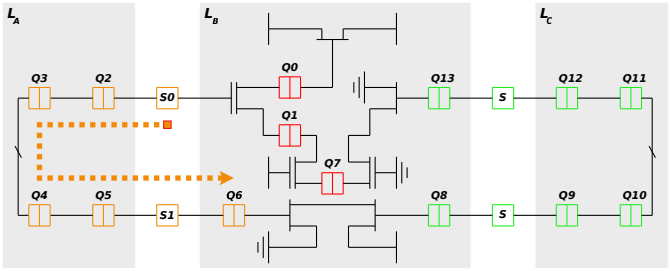


Fig. 19: GLOC3 example.

The next example GLOC3 is based on a direct deadlock error caused by a synchroniser $S0$. In this example module L_B communicates with two modules L_A and L_C via asynchronous channels. L_B transmits packets to L_A. As a result of a synchronisation handshake error [33] in synchroniser $S0$, $S0$ fails to communicate with L_B causing it to become idle resulting in a shutdown in communication between L_A and L_B. However, due to its design L_B only partially shuts down and still manages to communicate packets with L_C.

The examples in the table show how the time increases with an increase in xMAS nodes and a corresponding increase in number of local deadlocks ql which are proportional to xn.

C. Mesh examples

The examples Mesh1 to Mesh5 are mesh structures using in excess of 100 nodes. These used more complex structures consisting of many intra-modular and inter-modular loops. The number of mesochronous and asynchronous synchronisation units was varied for each experiment. These experiments were used to test the scalability of the verification. The results are shown in Table III. The results show how the time scales with increasing number of deadlocks. For the larger examples it takes significantly longer to search for deadlock details. The amount of relational information also increases significantly. The results show that the verification works well for examples using many hundreds of nodes.

Analysis of the GALS problem is challenging for formal methods because the large number of queues induces a very large state space leading to exponential increase; the limitations here are multi-dimensional due to the increase in xMAS nodes as well as the queue size (see Table III). Synchroniser issues are also limiting but these are not as problematic as for asynchronous circuits. Because of the overheads on net size vs xMAS model (approx 50 percent difference) and the exponential increase in unfolding the approach is currently limited to relatively small examples. The scalability indicates that the additions of several thousand xMAS nodes will require runtimes at least an order of magnitude higher. This can be mitigated due to the fact that the level of non-determinism may be limited in the analysis for certain circuits but this is example dependent and requires careful selection for a particular example to ensure that deadlocks are not missed. For CPNs improvements in efficiency such as the exploitation of parallelisation and merges may be used to mitigate the state explosion problem. Invariant techniques are much more efficient [11] [12] - but for some xMAS circuits may not always be so easy to derive - and therefore these techniques may fail when they can't be found.

VI. Conclusions

We have introduced a structured visual GALS modelling and verification environment for communication circuits. An integrated GALS platform has been provided using the Workcraft tool which allows a comprehensive approach by integrating multiple tasks into a unified environment. The visualisation capabilities provide enhanced feedback to the user during verification making it much easier for the user to investigate the causality of problems.

An approach to verification has been provided based on unfolding and deadlock analysis using Structured Occurrence nets which is well suited for GALS analysis. A novel representation has been presented using deadlock relations which enables the point-to-point causality between deadlocks to be viewed. This includes analysis of local and global deadlocks and enables visualization of total or partial system shutdown. Results show that deadlocks can be visualised easily and resolved efficiently. For future work we intend to adapt the system to check for livelock and test a wider range of synchronisers including MUTEX based.

Acknowledgements

This research was supported by EPSRC grants EP/K001698/1 (UNCOVER) and EP/I038551/1 (GAELS).

References

[1] S. Suhaib, D. Mathaikutty and S. Shukla, "Dataflow Architectures for GALS," *ACM Journal. Electronic Notes in Theoretical Computer Science (ENTCS)*, Vol 200, No. 1, pp. 33–50, 2008.

[2] T. Jungeblut, J. Ax, M. Porrmann and U. Ruckert, "A TCM-based architecture for GALS NoCs," *Proceedings of ISCAS'2012*, pp. 2721–2724, 2012.

[3] A. Yakovlev, P. Vivet and M. Renaudin, "Advances in asynchronous logic: from principles to GALS and NOC, recent industry applications, and commercial CAD tools," *Proceedings of DATE'2013*, 2013.

[4] C. Koch-Hofer, Y. Renaudin, Y. Thonnart and P. Vivet, "ASC, a System C extension for modelling asynchronous systems, and its application to an asynchronous NOC," *Proc. on Networks-on-Chip NOCS'2007*, pp. 295–306, 2007.

[5] Fatma Jebali, Frdric Lang and Radu Mateescu, "A Specification Language for Globally Asynchronous Locally Synchronous Systems," *Proc. ICFEM'2014*, pp. 219–234, 2014.

[6] Damien Thivolle and Hubert Garavel, "Verification of GALS Systems by Combining Synchronous Languages and Process Calculi," *Proc. SPIN'2009*, pp. 241–260, 2009.

[7] S. Ramesh, Sampada Sonalkar, Vijay D'Silva, Naveen Chandra and B. Vijayalakshmi, "A Toolset for Modelling and Verification of GALS System," *Proc. CAV'2004*, pp. 506–509, 2004.

[8] L. Janin and D. Edwards, "AsipIDE Tutorial - Bringing together GALS design and open-source tools in a hardware-software-FPGA co-simulation flow," *Tutorial at Conference ASYNC-NOCS*, 2010.

[9] S. Chatterjee, M. Kishinevsky and U. Ogras, "xMAS: quick formal modelling of communication fabrics to enable verification," *IEEE Design and Test of Computers*, Vol 29, No. 3, pp. 80–88, 2012.

TABLE III: xMAS Verification Results 3

			Basic					Rel					
k	Example	type	qn	in	sn	xn	ql	bd	id	bde	ide	src	time (sec)
2	VMesh1	mesoch	60	4	8	108	60	40	28	9	7	8	0.262
	VMesh2	asynch	60	8	8	116	60	59	9	10	4	8	1.845
	VMesh3	mesoch	126	9	18	231	126	126	18	24	8	13	4.282
	VMesh4	asynch	150	9	17	246	150	111	57	14	11	17	7.577
	VMesh5	mesoch	192	16	48	400	192	181	44	35	18	37	15.016
3	VMesh1	mesoch	60	4	8	108	60	38	30	8	8	8	0.751
	VMesh2	asynch	60	8	8	116	60	60	8	9	4	8	3.075
	VMesh3	mesoch	126	9	18	231	126	125	19	19	8	13	8.701
	VMesh4	asynch	150	9	17	246	150	109	59	14	11	17	18.580
	VMesh5	mesoch	192	16	48	400	192	181	44	41	29	37	33.542

[10] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," *Proc. of Intl. Conf. on Computer Aided Verification*, – 2010.

[11] A. Gotmanov, S. Chatterjee and M. Kishinevsky, "Verifying deadlock-freedom of communication fabrics," *Proc. VMCIA*, pp. 214–231, 2012.

[12] F. Verbeek and J. Schmaltz, "Hunting deadlocks efficiently in microarchitectural models of communication fabrics," *Proc. Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'11)*, pp. 223–231, 2011.

[13] S. Ray and R. Brayton, "Scalable progress verification in credit-based flow-control systems," *Proc. Design Automation and Test in Europe Conference and Exhibition (DATE'2012)*, pp. 905–910, 2012.

[14] D. E. Holcomb and S. A. Seshia, "Compositional Performance Verification of Network-on-Chip Designs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 33(9), pp. 1370–1383, 2014.

[15] S. Joosten and J. Schmaltz, "Generation of inductive invariants from register transfer level designs of communication fabrics," *Proc. Formal Methods and Models for Codesign (MEMOCODE) 2013*, pp. 57–64, 2013.

[16] S.J. Joosten and J. Schmaltz, "Automatic Extraction of Micro-Architectural Models of Communication Fabrics from Register Transfer Level Designs," *Design and Test Europe (DATE'15)*, Grenoble, France, March, pp. 9–13, 2015.

[17] F. Burns, D. Sokolov and A. Yakovlev, "GALS Synthesis and Verification for xMAS models," *Proceedings of DATE'2015*, pp. 1419–1424, 2015.

[18] A. Yakovlev, L. Gomes and L. Lavagno: "Hardware design and Petri nets," *Springer*, 2000.

[19] WORKCRAFT homepage. *http://workcraft.org/*

[20] I. Poliakov, V. Khomenko and A. Yakovlev, "Workcraft – a framework for interpreted graph models," *Proc. Int. Conf. on Applications and Theory of Petri Nets (ATPN'09)*, pp. 333–342, 2009.

[21] M. Koutny and B. Randell, "Structured occurrence nets: a formalism for aiding system failure prevention and analysis techniques," *Proc. ACM Fundamenta Informaticae*, pp. 41–91, 2009.

[22] A. Chakraborty and M. Greenstreet, "Efficient Self-Timed Interfaces for Crossing Clock Domains," *Proc. 9th IEEE Int'l Symp. Asynchrouns Circits and Systems (ASYNC) 2003*, IEEE CS Press, pp. 78–88, 2003.

[23] I. Poliakov, D. Sokolov and A. Mokhov, "Workcraft: a static data flow structure editing, visualisation and analysis tool" *Proc. Int. Conf. on Applications and Theory of Petri Nets and other models of concurrency (ICATPN'07)*, pp. 505–514, 2007.

[24] D. Kinniment, "Synchronization and Arbitration in Digital Systems," *Wiley Publishing*, 2008.

[25] M. Krstic, M. Grass, E. Gurkaynak, F. and P. Vivet, "Globally Asynchronous, Locally Synchronous Circuits, Overview and Outlook" *Design and Test of Computers, IEEE*, Vol. 24, No. 5, pp. 430–441, 2007.

[26] K. McMillan, "A technique of state space search based on unfolding," *Formal Methods in System Design*, Vol. 6(1), pp. 45-65, 1995.

[27] J. Esparza, S. Romer and W. Volger, "An improvement of McMillans's unfolding algorithm," *Proc. Formal Methods in System Design*, Vol. 20, No. 3, pp. 285–310, 2002.

[28] B. Bonet, P. Haslum, V. Khomenko, S. Thibaux and W. Vogler, "Recent advances in unfolding technique," *Theoretical Computer Science*, Vol. 551, pp. 84-101, 2014.

[29] J. Fernandes, M. Koutny, L. Mikulski, M. Pietkiewicz-Koutny, D. Sokolov and A. Yakovlev, "Persistent and nonviolent steps and the design of GALS systems," *Fundamenta Informaticae*, Vol. 137(1), pp. 143-170, 2015.

[30] J. Kleijn and M. Koutny, "Causality in Structured occurrence nets," *In: Dependable and Historic Computing. vol. 6875 of LCNS*, Springer pp. 283–297, 2011.

[31] F. Verbeek, "Formal Verification of On-Chip Communication Fabrics," *PhD thesis*, March, 2013.

[32] S. Wouda, S.J.C Joosten and J. Schmaltz, "Process algebra semantics and reachability analysis for micro-architectural models of communication fabrics," *IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'15)*, Austin, USA, September, 2015.

[33] F. Verbeek, S. Joosten and J. Schmaltz, "Formal Deadlock Verification for Click Circuits," *19th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'13)*, Santa Monica, May, pp. 19–22, 2013.