GALS synthesis and verification for xMAS models

Frank Burns
School of Computing Science
Newcastle University, UK
Email: frank.burns@ncl.ac.uk

Danil Sokolov
School of EEE
Newcastle University, UK
Email: danil.sokolov@ncl.ac.uk

Alex Yakovlev
School of EEE
Newcastle University, UK
Email: alex.yakovlev@ncl.ac.uk

***Abstract*—In this paper a novel Globally Asynchronous Locally Synchronous (GALS) synthesis and verification environment is introduced for xMAS models. xMAS models are a new communication paradigm which can be used to model circuits and networks for the purpose of synthesis, testing and verification. Previous attempts at synthesis and verification of xMAS models have been proposed for synchronous implementations only. This paper provides an extension of xMAS and translation into Circuit Petri net models for GALS synthesis and verification. Synthesis techniques based on Circuit Petri net translation are presented and a new xMAS component is introduced which acts as a wrapper for different GALS styles. Novel verification techniques using unfolding to occurrence nets are then proposed. Our results show that the work presented here provides a suitable platform for integrating xMAS into a GALS environment.**

I. Introduction

Whilst there has been a lot of interest in researching new architectures for GALS [1][2][3], there have been few attempts at providing synthesis solutions for GALS communication. Thus, generation of GALS from specifications has been limited to hardware description languages such as Verilog, VHDL [4] or synchronous programming languages such as C or ESTEREL [5]. Some research has been done in synthesising GALS directly from nets. In [6] they provide a method for generating GALS implementations from dataflow graphs. In [7] a method is presented for synthesising GALS controllers from multi-burst graph specifications. Although these methods are useful they are not general in their approach and are limited to specialist cases.

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 communication primitives together. This kind of modelling tends to be unwieldy and non-intuitive. xMAS [8] represents a significant improvement in the representation and modelling of communication systems. It provides a set of graphical communication primitives which are more natural and their higher level of abstraction enables them to be easily understood. Whilst xMAS modelling has been applied to model synchronous implementations and it is extendable to asynchronous modelling, currently, there are no tools which provide a GALS environment for it. This paper aims towards a comprehensive, transparent tool for GALS by providing a platform which integrates xMAS into a GALS synthesis and verification environment.

Although xMAS model checking has been covered extensively at the Boolean level for purposes such as deadlock checking using invariant models [8][9], no research has been done using net checking models such as Petri nets. Circuit Petri nets [10] provide a natural means for translation of the xMAS equations and they are also well suited to the visualisation of distributed models of local machines in terms of concurrency. For verification they capture a complete knowledge in the unfolding hence providing a represention of the full causality. They are also flexible allowing different execution policies to be defined. In this paper we introduce a method of direct translation to Circuit Petri nets together with an execution policy so that the nets are executed according to the model semantics. For verification a novel unfolding algorithm is presented from Circuit Petri nets to occurrence nets which allows for detailed visual deadlock checking and modelling. The verification algorithm is flexible, it is not reliant on the use of invariants and is adaptable to the timing of alternate GALS implementations.

For global communication an additional xMAS primitive is introduced which provides a wrapper for synchronisation. The primitive is in keeping with the xMAS tradition of hiding the details of the ”glue logic” from the user. A synthesis module is provided for the synchronisation primitive for synthesising a range of ”glue” solutions e.g. asynchronous, mesochronous, etc. This part of the synthesis involves translation of synchronisation components to Signal Transition Graphs (STGs) that enables their verification using established Petri net unfolding techniques [11].

The main contributions of this work are:

- integration of xMAS into a GALS environment including the addition of a new xMAS primitive;
- GALS synthesis for xMAS models using Circuit Petri Net representations and STGs;
- verification including a novel unfolding algorithm.

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 [8]. The benefit of the equations is there is a clear distinction between the transformation applied to data and the logic coordinating its movement. There are eight communication primitives altogether and these are depicted in Fig. 1.

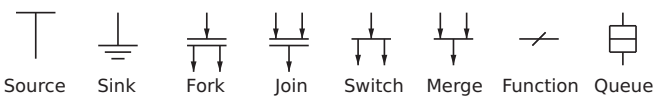

Fig. 1: xMAS primitives.

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

978-3-9815370-4-8/DATE15/©2015 EDAA

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. 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. The signals $irdy$ and $trdy$ stand for initiator ready to send and target ready to receive. 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 input 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 [8].

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
  mem_j = if (i.irdy and i.trdy and j=pre(tl))
          then i.data else pre(mem_j)
```

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

We have developed an xMAS tool for graphical entry of xMAS diagrams. It incorporates an xMAS module for constructing the xMAS models. In addition to the symbols for all the basic primitives a new asynchronous synchronisation primitive has been added to the basic set of primitives shown in Fig. 2. 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.

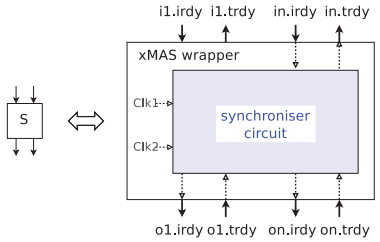

Fig. 2: xMAS synchronisation primitive.

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 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.

III. GALS Synthesis of xMAS models

A. Circuit Petri net synthesis

1) Primitive Translation: Circuit Petri Nets (CPNs) are a specific type of STG used for modelling the behaviour of logic gates (i.e. level-based components) [10]. Translation of xMAS models to Circuit Petri nets requires translation from the basic primitives; for each primitive in xMAS there is a corresponding Circuit Petri Net. 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 added by the system. The following diagrams provide the translation for the key primitives.

Fig. 3 shows the translation of the basic source primitive. In Fig. 3 the source is comprised of two internal signals, an $oracle$ and a ready signal $irdy$. The $oracle$ and $irdy$ signals are connected by internal read arcs (bi-directional) which allow the $irdy$ signal to be enabled and disabled by the $oracle$. The resetting of the $irdy$ signal depends on the external connection $trdy$ which is connected by a read arc to the incoming signal from the receiving primitive that the source is sending to. Additional signals are added to the oracle $s = 0$ and $s = 1$. These are added by the system and are used for setting the mode for the oracle: dead $(s = 0)$, eager $(s = 1)$ or non-deterministic (mixed). The sink has a similar structure to the source in which the $irdy$ and $trdy$ signals are reversed.

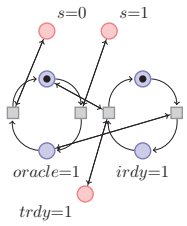

Fig. 3: CPN transl. - Source.

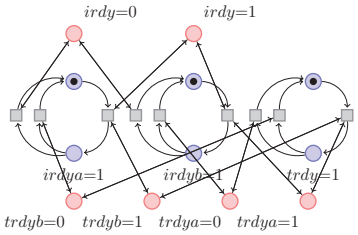

Fig. 4: CPN transl. - Fork.

The fork, (Fig. 4), is comprised of three signals, $irdya$, $irdyb$ and $trdy$ as depicted by the equations for the fork. External connections are shown to the ready $irdy$ signal and the two incoming receive $trdy$ signals. The external receive signals are crossed as to align with the semantics of xMAS. The cross-over is required for synchronisation purposes. The join net (not shown), which is for a restricted join, has a similar structure to the fork in which the roles of the $irdy$ and $trdy$ signals are reversed.

Fig. 5 shows the translation of the basic switch primitive which is shown optimized. The switch primitive is comprised of four signals, sw, $irdya$, $irdyb$ and $trdy$. The sw signal is connected to the internal $irdy$ signals by read arcs which are used to determine the selection of the $irdya$ or $irdyb$ signals. The sw signal is data dependent and is sensitive to changes that occur in external data signals $s(i.data)$.

The merge primitive shown in Fig. 6, is comprised of four signals, $irdy$, mg, $trdya$ and $trdyb$. The mg signal which is set to operate a fairness policy by default is connected to the

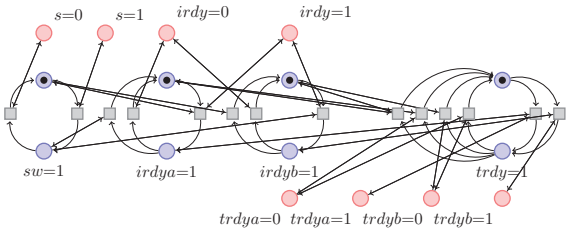

Fig. 5: CPN translation - Switch.

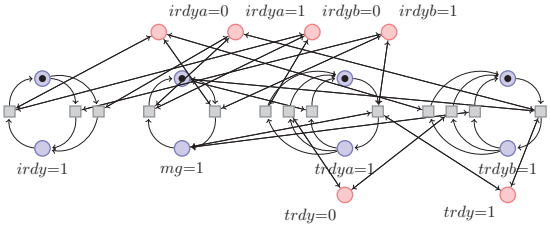

Fig. 6: CPN translation - Merge.

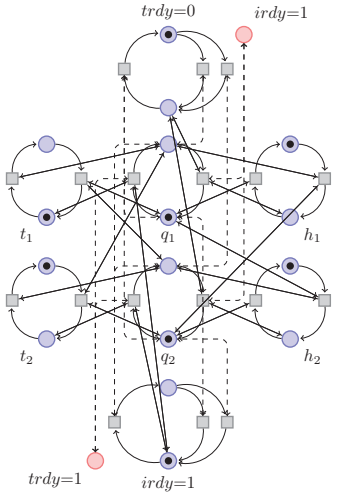

Fig. 7: CPN translation - Queue.

internal $trdy$ signals and is used to determine the selection of the corresponding output path, $trdya$ or $trdyb$.

Fig. 7 shows a translation for a queue primitive of size 2. This uses a one-hot representation. This is used for the head and tail pointers which are shown to the left and the right hand side of the queue body. The main body of the queue comprises a number of slots, one for each queue entry. Each input slot is connected to corresponding head and tail pointers. Each slot is also connected to the ready and receive signals. 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 send and receive 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, q and hd and can be scaled in size by adding additional sections.

The Circuit Petri net translator, inside the xMAS tool, accepts a Json (data-interchange format) representation of the xMAS model and translates it into a Petri net representation. For translation the primitives are generated as depicted in the diagrams. A data-line is automatically added to include data signals. The net primitives are then connected by a process which links together all the external connections.

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. 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 add priorities to transitions, whereby a transition cannot fire, if a higher-priority transition is enabled.

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.

In addition changes in source and sink oracles must occur as a multiple of queue transfers. For this two relations are required. Firstly:

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

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. 3); 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.

Secondly a specific mapping $\Pi = \Pi_{QO}$ is required (3) which relates queue to source and sink oracles:

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

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. 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.

B. Synchroniser synthesis

For synchroniser synthesis synchronisation primitives specified by the user are passed to a synchronisation synthesis module inside the xMAS tool. Circuits are automatically generated by the synthesizer for the primitives which are subsequently converted into an STG representation. Implementation options are provided which enable the user to make a decision with regard the internal details based on the GALS style that is required. The style is chosen from a selection of available GALS implementations: asynchronous, mesochronous, pausible clocking, etc. For each implementation 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 are used later in the verification.

An example implementation generated by the synchronisation module for the asynchronous style is shown in Fig. 8. This 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.

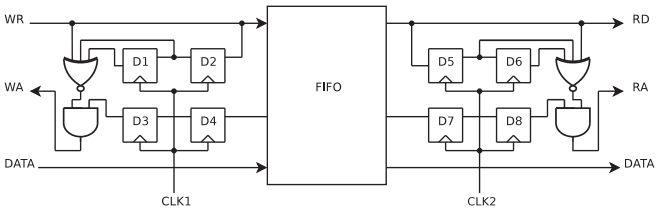

Fig. 8: Asynchronous synchronisation.

IV. Verification

A. Circuit Petri Net verification

The xMAS models are verified by a process of unfolding to occurrence nets and deadlock checking [12]. The unfolding proceeds in a parallel manner based on the execution semantics of the xMAS model. The unfolding algorithm is shown below. 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. In the algorithm the SEED refers to the initial conditions. For synchronisation of clocking domains the

Algorithm 1 Unfolding algorithm

```
Add the conditions in the SEED to the prefix
Initialise the priority queue q with the events possible in the SEED
Initialise the cut-off set to φ
while q ≠ φ do
    sort the queue in order of priority (highest first)
    for each transition t in the highest priority set do
        if cut-off is not detected then
            Add corresponding event e and postset to prefix
            Check valid xMAS queue addition with reference to clock domain
            Insert new possible events into the queue
        else
            cut-off ← cut-off ∪ e
        end if
    end for
end while
add the postsets of all cut-off events to the prefix
```

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 partition is set at different rates according to the rate of queue firing. This can be set in the unfolding algorithm by controlling the addition of the xMAS queues at different rates according to the relative frequencies of the local partitions in which they reside.

From the unfolding deadlock checking is made from the occurrence nets. This uses traditional global verification deadlock analysis techniques. Local xMAS deadlocks [13] are also analysed at this stage. In [13] 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. If only local blocking is found this is reflected back to the user in the form of local blocking messages to highlight the queues that are locally blocked. For full deadlock, traces are generated in the xMAS tool to reflect the sequence of transitions leading to deadlock; queue traces may also be highlighted and visualised separately.

To limit the verification effort experiments were conducted using a mixed-mode consisting of eager and non-deterministic. In this mode the sources are 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 deadlocks can be found more efficiently. The experiments were conducted using an Intel Core i7 3.4GHz processor.

For the experiments a number of different xMAS circuits were tested. The results of the verification are shown in Table 1. The results are shown in terms of the queue size k, the number of sources i, the number of synchronisers s, the number of xMAS primitives n, the type of GALS implementation and the time in seconds it takes to find the presence of deadlock(s). The first example COMM1 is taken from the circuit in Fig. 9.

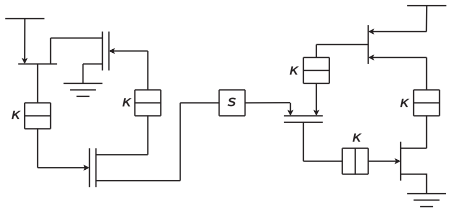

Fig. 9: COMM1 example.

The xMAS model comprises communicating agents split into two parts across a local mesochronous division. In this example one-way communication is used in which information is passed from one agent to the other. This experiment uses the least number of xMAS nodes and takes a trivial amount of time to complete.

The second example COMM2 is taken from the xMAS model in Fig. 10. 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. Due to the size of the model the time taken is significantly longer than the first example.

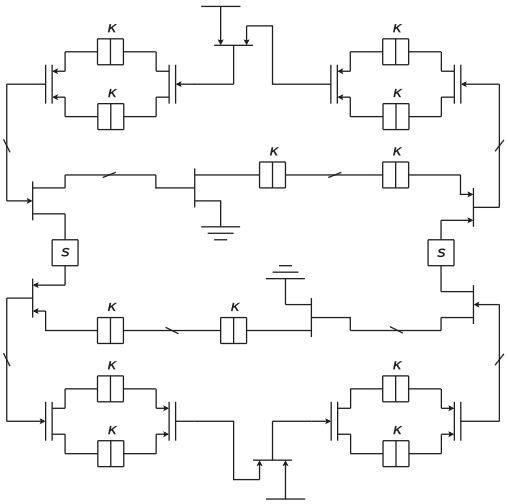

Fig. 10: COMM2 example.

The next set of experiments, shown in Table 1, GLOC, are examples in which the GALS modules are structurally designed so that significant numbers of structural local deadlocks are generated internally inside the GALS modules. These deadlocks are not generated by loops but occur locally inside each module due to local structural blocking [13]. In these experiments all the local modules were linked together using mesochronous synchronisation units. The number of inputs and synchronization units were kept the same for each experiment. The number of local deadlocks for the 3 different examples were 17, 25 and 33 respectively. For the experiments the table shows the time taken for all the deadlocks to be detected.

TABLE I: xMAS Verification Results

	k = 2					k = 3				
Example	i	s	n	asynch	time(s)	i	s	n	asynch	time(s)
COMM1	2	1	16	mesoch	0.058	2	1	16	mesoch	0.306
COMM2	2	2	42	asynch	0.624	2	2	42	asynch	1.438
GLOC1	5	5	79	mesoch	0.283	5	5	79	mesoch	0.430
GLOC2	5	5	109	mesoch	0.692	5	5	109	mesoch	1.152
GLOC3	5	5	139	mesoch	1.437	5	5	139	mesoch	2.357
Mesh1	8	4	104	asynch	0.723	8	4	104	asynch	2.380
Mesh2	8	8	104	asynch	1.116	8	8	104	asynch	2.951
Mesh3	16	12	228	asynch	4.714	16	12	228	asynch	15.788
Mesh4	16	24	228	asynch	6.606	16	24	228	asynch	27.791

Finally, the examples Mesh1 to Mesh4 are mesh structures comprising more than 100 nodes. These were split into two sizes using more complex structures consisting of many intra-modular and inter-modular loops. The number of asynchronous synchronisation units was varied for each experiment. These experiments were used to test the scalability of the verification. The results for the experiments show the variation in time with the number of synchronisation units used. For the larger examples it takes significantly longer to search for a deadlock.

B. STG verification

For synchronisation, after the synchronisers have been implemented, verification is achieved by STG validation. In the xMAS tool a menu is provided for all synchronisation primitives which provides a selection of options for each style of implementation. In the synchronisation module, an additional menu is provided which enables the conversion of each synchronisation primitive from a circuit implementation into an STG representation.

Once generated the STGs can then be edited inside the xMAS tool and validated using a range of available verification tools used for STG analysis. These include standard STG synthesis and verification tools such as Petrify [14] and MPSat [15]. These tools are used for ensuring that the resulting circuits exhibit hazard free behaviour by providing protection against critical races hence providing deadlock protection at the circuit level.

V. Conclusions

We have introduced a GALS synthesis and verification environment for xMAS. A method of formal translation of xMAS models to Circuit Petri nets has been presented. Execution policies enable the nets to work according to the required execution semantics. A synchronisation primitive has been introduced to xMAS. Different GALS synchronisers can be synthesised based on the chosen implementation style.

A verification environment has been provided together with an algorithm for verifying the Circuit Petri nets. This is based on unfolding and deadlock checking using occurrence nets which allows for both checking and detailed visualisation of local and global deadlocks. The verification algorithm is flexible and adaptable to the timing of alternate GALS implementations. The approach taken hides the complexity of the GALS implementation from the user.

Acknowledgements

We would like to acknowledge the EPSRC who supported this research through grants UNCOVER (EP/K001698/1) and GAELS (EP/I038551/1).

References

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

[2] Fan, X., Krstic, M., Grass, E., Sanders, B., Heer, C.: Exploring pausible clocking based GALS design for 40-nm system integration. *Proceedings of DATE'2012*, 118–121 (2012)

[3] Jungeblut, T., Ax, J., Porrmann, M., Ruckert, U.: A TCM-based architecture for GALS NoCs. *Proceedings of ISCAS'2012*, 2721–2724 (2012)

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

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

[6] Praxxibhbu, H., Thomas, S., Rodrigues, J., Olsson, T., Carlsson, A.: A GALS ASIC implementation from a CAL dataflow description. *Proceedings of NORCHIP'2011*, 1–4 (2011)

[7] Oliviera, D., Lussari, E.: Synthesis of robust controllers for GALS FPGA from multi-burst graph specification. *Proc. South. Conf. on Programmable Logic (SPL)'2011*, 123–129 (2011)

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

[9] Gotmanov, A., Chatterjee, S., Kishinevsky, M.: Verifying deadlock-freedom of communication fabrics. *Proc. VMCIA*, 214–231 (2012)

[10] Yakovlev, A., Gomes, L., Lavagno, L.: Hardware design and Petri nets. *Springer*, (2000)

[11] Esparza, J., Schroter, C.: Unfolding based algorithms for the reachability problem. *Journal Fundamenta Informaticae - Concurrency Specification and Programming (CSP'2000)*, Vol 47, no. 3-4, 231–245 (2001)

[12] Esparza, J., Romer, S., Volger, W.: An improvement of McMillans's unfolding algorithm. *Proc. Formal Methods in System Design*, Vol 20, no. 3, 285–310 (2002)

[13] Verbeek, F.: Formal Verification of On-Chip Communication Fabrics. *PhD thesis*, March (2013)

[14] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, Vol. E80-D, no. 3, 315–325 (1997)

[15] Khomenko, V., Koutny, M., Yakovlev, A.: Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT. *Fundamenta Informaticae*, Vol. 70, no. 1-2, 49–73 (2006)