Investigating the side-effects of synchronisers on GALS circuits

Frank Burns

Newcastle University, Newcastle Upon Tyne, UK,
frank.burns@newcastle.ac.uk

Abstract. A choice of synchroniser may be crucial for the correct operation of a GALS circuit. GALS NOCs currently require thousands of synchronisers to communicate information accross clock boundaries. Carefully designed synchronisers are capable of mitigating the effects of metastability errors within a chips lifetime. An arbitrary choice of synchroniser, however, may be affected by metastability or timing issues early on in a chips lifetime, resulting in detrimental side-effects and ultimately failure. This, however, is largely dependent on the circuit design. This paper models GALS communication circuits using xMAS models to investigate the potential side-effects of different types of synchroniser on a variety of xMAS circuits. Different xMAS models are analysed to quantify and classify the level of robustness and the exposure to side-effects based on the synchroniser selection.

Keywords: xMAS models, GALS, Synchronisers, Verification

1 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, SystemC [4][5] or synchronous programming languages such as C or ESTEREL [6]. 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 [7][8][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 and their higher level of abstraction enables them to be easily understood.

Circuit Petri nets [12] 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. In [13] basic techniques for GALS synthesis to Circuit Petri nets 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. Basic techniques for GALS verification were also presented including unfolding to occurrence nets and deadlock analysis.

In [13] an additional xMAS synchroniser primitive was introduced to provide a synchronisation wrapper for synthesising a range of ”glue” solutions e.g. asynchronous, mesochronous, etc. The system is capable of detecting the side-effects of synchronisation problems through unfolding and verification and signalling a potential shutdown. To improve metastability MTBF, designers can take specific measures. For example, they can change the metastability settling time by adding extra register stages to synchronization register chains. The timing slack on each additional register-to-register connection is added to the metastability settling time value. Designers commonly use two registers to synchronize a signal, but some companies recommend using a standard of three registers for better metastability protection. However, adding a register adds an additional latency stage to the synchronization logic, so arises a trade-off between logic and robustness. Also the choice of a specific synchroniser can have a significant impact depending on the particular design.

This paper models GALS communication circuits using xMAS models to investigate the potential side-effects of different types of synchroniser on a variety of xMAS circuits. An arbitrary choice of synchroniser may cause metastability problems and another may not. This, however, is largely dependent on the circuit design. Different xMAS models are analysed to quantify and classify the level of robustness and the exposure to side-effects based on the synchroniser selection.

The main contributions of this work are:

– Analysis of deadlocks in xMAS models due to synchronisers;
– investigaton of the potential side-effects of different types of synchroniser using a variety of xMAS circuits;
– testing the level of robustness of a design based on synchroniser selection.

2 xMAS modelling

2.1 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 [7]. There are eight communication primitives altogether and these are depicted in Fig. 1.

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

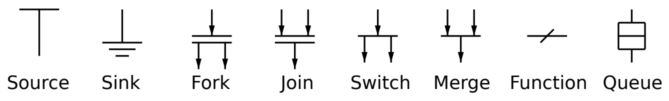

Fig. 1. xMAS primitives.

```
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
```

A merge has multiple input ports and one output port. 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 [7].

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.

2.2 GALS Asynchronous Primitive

We have developed a modelling tool in WORKCRAFT [14] 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.

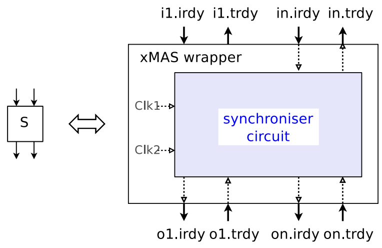

Fig. 2. xMAS synchronisation primitive.

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.

2.3 Synchroniser modelling

For modelling synchronisers [15] 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 [16]

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.

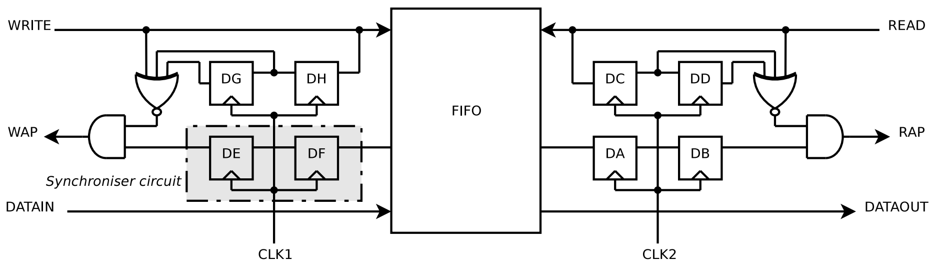

Fig. 3. Asynchronous synchronisation.

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. 3. The implementation in Fig. 3 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.

The synchroniser is used to synchronise the asynchronous communication signal with the local clock. The synchroniser circuit, which is a two-flop synchroniser, 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) [15].

The MTBF for a specific signal transfer, or all the transfers in a design, can be calculated using information about the design and the device characteristics. The MTBF of a synchroniser chain is calculated with the following formula and parameters:

$$\frac{e^{t_{MET}}/C_2}{C_1 \cdot f_{CLK} \cdot f_{DATA}} \tag{1}$$

where the C_1 and C_2 constants depend on the device process and operating conditions; f_{CLK} is the clock frequency of the receiving clock domain; f_{DATA} is the toggling frequency of the input data signal and the t_{MET} parameter is the metastability settling time. For a synchroniser chain t_{MET} is the sum of the output timing slacks for each register in the chain.

The overall design MTBF can be determined by the MTBF of each synchroniser chain in the design. The failure rate for a synchroniser is $1/MTBF$, and the failure rate for the entire design is calculated by adding the failure rates for each synchroniser chain, as follows:

$$Failrate_{design} = \frac{1}{MTBF_{design}} = \sum_{n=1}^{nochains} \frac{1}{MTBF_i} \tag{2}$$

For the two-flop synchroniser a failure could result in the addition of a clock cycle to the latency.

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 verification. Potential synchronisation problems due to metastability are exploited in the unfolding by varying or altering the clock cycles. This is used as a margin of error for the two-flop synchroniser to investigate the effect of a change in the latency.

3 Modelling of deadlocks in Synchronisers

The modelling of the GALS circuits and deadlock analysis is conducted using the WORKCRAFT tool. In [13] we described a methodology and approach for analysing deadlocks in GALS communication circuits using an unfolding algorithm. In [14] the analysis method has been augmented using deadlock relations which are derived from Communication Structured Occurrence Nets $CSONs$ [17].

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 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.

3.1 Deadlock relations

Deadlock relations are derived from the nets. 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. 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 $q2^{L_A} \overset{B}{\leftarrow} q1^{L_A}$.*

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 $q1^{L_A} \overset{I}{\rightarrow} q2^{L_A}$.*

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 $q1^{L_A} \overset{B}{\leftarrow} S1$. The reverse relation of this can be expressed using $S1 \overset{B}{\leftarrow} q1^{L_A}$.*

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 $S1 \overset{I}{\rightarrow} q1^{L_B}$. The reverse relation of this can be expressed using $q1^{L_B} \overset{I}{\rightarrow} S1$.*

The above relations can be chained together. The following equations show examples of chained relations. Equation (3) 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 (4) 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.

$$q1^{L_A} \overset{I}{\rightarrow} S0 \overset{I}{\rightarrow} q2^{L_B} \tag{3}$$

$$q1^{L_A} \overset{B}{\leftarrow} S0 \overset{B}{\leftarrow} (q2^{L_B} \overset{B}{\leftarrow} q3^{L_B}) \tag{4}$$

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 (3), above, is an example of an *ide* relation and equation (4) 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 from the unfolding 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 (3), for example, to produce equation (5), reflecting the relation between $q1^{L_A}$ and $q2^{L_B}$:

$$q1^{L_A} \overset{I}{\rightarrow} S0 \cdot S0 \overset{I}{\rightarrow} q2^{L_B} \implies q1^{L_A} \overset{B}{\rightarrow} q2^{L_B} \tag{5}$$

Hence, it becomes possible using the relational model to query directly point-to-point causality between queues in different modules.

3.2 Modelling of deadlocks due to synchroniser problems

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 [18]. (ii) indirect: i.e. timing problems due to the latency. This is a result of setup time and metastability problems which can result in latency mismatch and subsequent functional errors in the adjoining modules.

Direct deadlock The example below 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 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.

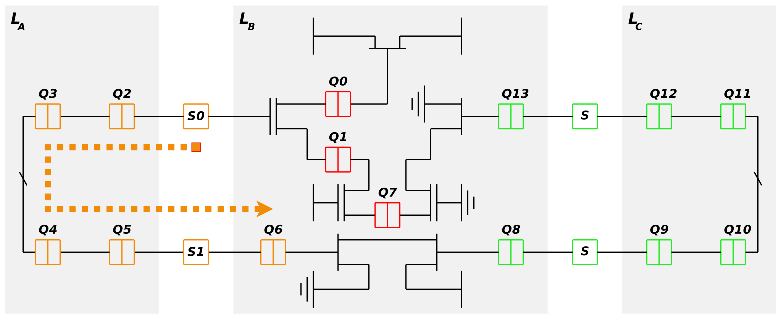

Fig. 4. Direct deadlock example.

The equation showing the synchroniser deadlocks are shown below.

$$S0 \xrightarrow{I} (q2 \xrightarrow{I} q3 \xrightarrow{I} q4 \xrightarrow{I} q5)^{L_A} \tag{6}$$

$$q5^{L_A} \xrightarrow{I} S1 \xrightarrow{I} q6^{L_B} \tag{7}$$

Here an indirect *idle* deadlock occurs in $S1$ due to the chain of *ide* deadlock relations.

Indirect deadlock The example below, in Fig. 5, is based on deadlock due to timing mismatch issues caused by a synchroniser. 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 MTBF time window 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

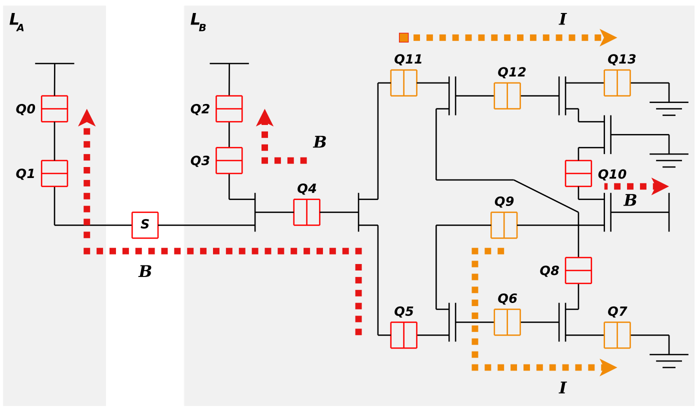

Fig. 5. Synchroniser deadlock example.

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 MTBF resulting in downstream functional errors. The equations related to the deadlock are.

$$q9^{L_B} \xrightarrow{I} q6^{L_B} \xrightarrow{I} q7^{L_B} \tag{8}$$

$$(q0^{L_B} \xleftarrow{B} q1^{L_B}) \xleftarrow{B} S \xleftarrow{B} (q4^{L_B} \xleftarrow{B} q5^{L_B}) \tag{9}$$

An indirect relation between (8) and (9) via the join means that when $q9$ is emptied $q5$ becomes blocked.

For the same circuit a mesochronous synchroniser can be substituted in place of the asynchronous synchroniser and the synchroniser will only deadlock with a level of probability which is significantly lower. The level of robustness of the implementation of the mesochronous case can be approximated based on an estimate of the relative timing gap in relation to metastability.

This estimate is measured in terms of the following equation:

$$\sum_{n=1}^{nochains} \frac{1}{k \cdot e^{-(t' \cdot n)/t_i}} \tag{10}$$

where $t' \cdot n$ provides a measure of the relative timing gap.

4 Analysis and experiments

A set of experiments was conducted using a variety of xMAS circuits and different synchronisers. Verification proceeds by searching for direct deadlocks or those

Table 1. xMAS Verification Results [k=2]

Example	i	s	n	asynch	dlk	lrb	time(s)
PC1	4	4	34	asynch	2	0	0.291
PC2	3	1	30	asynch	1	0	0.430
PC3	3	1	30	mesoch	0	0.055	0.341
Agent1	6	2	50	asynch	2	0	0.402
Agent2	6	2	50	mesoch	0	0.028	0.360
Agent3	6	2	52	mesoch	0	1.515	0.368
Mesh1	8	4	104	asynch	4	0	1.550
Mesh2	8	8	104	asynch	0	0.007	1.628
Mesh3	16	12	228	asynch	0	0.004	5.770

due to setup problems which are reported immediately if they are present. In the presence of deadlocks the level of robustness is set to 0. In the absence of deadlocks the level of robustness is measured in terms of a metric based on the metastability gap. The level of robustness of the designs was measured based on its relative level to the circuit and the choice of synchroniser. A trade-off is possible in certain cases based on the choice of synchroniser versus the level of robustness.

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 the analysis can be performed 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 $k = 2$, the number of sources i, the number of synchronisers s, the number of xMAS primitives n, the type of GALS implementation, the number of synchroniser deadlocks dlk, the level of robustness lrb, and the time in seconds it takes to calculate the results.

The first set of experiments are producer consumer examples. These are basic communication examples using point-to-point communication only. The first example calculates a direct deadlock in 0.291s. The second example $PC2$ uses an asynchronous synchroniser has 1 deadlock due to setup problems and its level of robustness is 0. The third example $PC3$ which uses a mesochronous synchroniser for the same design, is deadlock free, and, therefore, its level of robustness is is estimated. This appears as 0.055 in Table 1.

The next set of experiments, shown in Table 1, are agent examples in which the GALS modules are structurally designed so that varying numbers of communicating agents communicate with each other. The first example uses an asynchronous synchroniser has 2 deadlocks due to setup problems. The second ex-

Table 2. xMAS Verification Results [k=3]

Example	i	s	n	asynch	dlk	lrb	time(s)
PC1	4	4	34	asynch	2	0	0.452
PC2	3	1	30	asynch	1	0	1.135
PC3	3	1	30	mesoch	0	0.055	1.062
Agent1	6	2	50	asynch	2	0	1.165
Agent2	6	2	50	mesoch	0	0.028	1.129
Agent3	6	2	52	mesoch	0	1.515	1.142
Mesh1	8	4	104	asynch	4	0	4.520
Mesh2	8	8	104	asynch	0	0.007	4.692
Mesh3	16	12	228	asynch	0	0.004	18.532

ample *Agent*2 uses a mesochronous synchroniser has 0 deadlocks and its level of robustness is 0.028. The third example *Agent*3 which uses two mesochronous synchronisers is deadlock free but the level of robustness is much higher 1.515 due to the estimate for the metastability gap being larger.

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 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 level of robustness is much lower for an increase in the number of synchronisation units used. For the larger examples it takes significantly longer to test the level of robustness to synchronisation problems.

Table 2 shows results for the same set of experiments using a different queue size [k=3] which shows a comparison of times.

5 Conclusions

We have provided a GALS synthesis and verification environment for xMAS. This has been used for analysing problems caused by synchronisation. It is based on unfolding and deadlock analysis which allows for both checking and visualisation of different types of synchroniser deadlocks. A unique deadlock analysis approach using relations has been described for verifying the examples.

The verification approach is flexible and adaptable to the timing of alternate GALS implementations. Different GALS synchronisers can be selected based on the chosen implementation style. The approach taken enables the investigaton of the potential side-effects of different types of synchroniser using a variety of xMAS circuits. The approach allows for testing the level of robustness of a design based on synchroniser selection.

Acknowledgments

Acknowledgments to Alex Yakovlev et. al.

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. 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.
6. 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.
7. 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.
8. Chatterjee, S., Kishinevsky, M.: Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics. *Proc. of Intl. Conf. on Computer Aided Verificationi, CAV'2010*, 2010.
9. Gotmanov, A., Chatterjee, S., Kishinevsky, M.: Verifying deadlock-freedom of communication fabrics. *Proc. VMCIA*, 214–231, 2012.
10. 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.
11. 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.
12. Yakovlev, A., Gomes, L., Lavagno, L.: Hardware Design and Petri Nets. *Springer*, (2000)
13. F. Burns, D. Sokolov and A. Yakovlev, "GALS Synthesis and Verification for xMAS models," *Proceedings of DATE'2015*, pp. 1419–1424, 2015.
14. WORKCRAFT homepage. *http://workcraft.org/*
15. D. Kinniment, "Synchronization and Arbitration in Digital Systems," *Wiley Publishing*, 2008.
16. 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.
17. Koutny, M., Randell, B.: Structured Occurrence Nets: A Formalism for Aiding System Failure Prevention and Analysis Techniques. *Proc. ACM Fundamenta Informaticae*, 41–91, 2009.
18. 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.