Formal Verification of Synchronizers

Tsachy Kapschitz and Ran Ginosar

VLSI Systems Research Center, Electrical Engineering Department
Technion–Israel Institute of Technology, Haifa 32000, Israel
[ran@ee.Technion.ac.il]

Abstract. Large Systems on Chips (SoC) comprise multiple clock domains, and inter-domain data transfers require synchronization. Synchronizers may fail due to metastability, but when using proper synchronization circuits the probability of such failures can be made negligible. Failures due to unexpected order of events (caused by interfacing multiple unrelated clocks) are more common. Correct synchronization is independent of event order, and can be verified by model checking. Given a synchronizer, a correct protocol is guessed, verification rules are generated out of the protocol specification, and the model checker applies these rules to the given synchronizer. An alternative method verifies correct data transfer and seeks potential data missing or duplication. Both approaches require specific modeling of multiple clocks, allowing for non-determinism in their relative ordering. These methods have been applied successfully to a two-flip-flop synchronizer and a dual clock FIFO.

1   Introduction

Synchronous (clocked) hardware systems are typically partitioned into multiple clock domains. All sequential elements in the same clock domain are clocked at presumably the exact same time, but the clocks at different domains may be mutually unrelated—they may operate at different frequencies, and even when operating at the same frequency they may tick at different times. These relative frequency and phase differences may be unknown a-priori, and may also change over time [1]. Such multi-clock domain systems are also termed GALS (globally asynchronous, locally synchronous) since the different clocks are assumed mutually asynchronous.

Data transfers between different clock domains require synchronization [2]. Data that enters a domain and happens to change exactly when the receiving register is sampling its input may cause that register to become metastable and fail [3]. This problem is mitigated by properly employing synchronizers and by formally verifying their correctness. This paper describes methods for formal verification of synchronizers based on model-checking [4].

Common synchronizers comprise two levels, continuous and discrete. In the realm of continuous time, it is recognized that often an input change coincides with the sampling time and consequently the sampling circuit (typically a flip-flop; this paper does not treat latch-based synchronizers) may take an arbitrarily long time to resolve [3]. The synchronization circuit is allowed a certain time period S for resolution, so that (in case the input indeed changes state at the sampling point) the probability that the circuit fails to resolve S time later is negligible. However, when the

synchronization circuit resolves successfully after such a coincidence within the period S, its output is set non-deterministically. A typical synchronization circuit comprises two consecutive flip-flops (Fig. 1) where the first is clocked at the sampling point and the second one is clocked S time afterwards. Analysis of such synchronization circuits lies in the analog circuit domain [1][5] and their presence may be validated by structural verification algorithms [6]. Both topics are outside the scope of this paper.
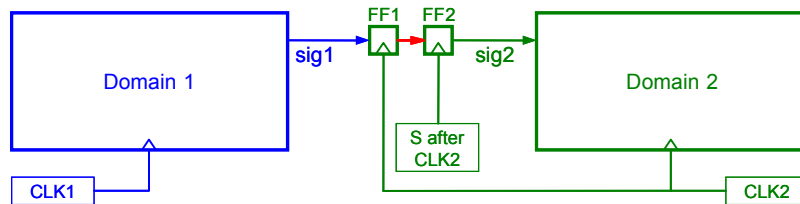

Fig. 1: The two flip flop synchronization circuit with resolution time S

The discrete synchronizer assumes that continuous time issues have already been resolved by the constituent synchronization circuit(s). The synchronizer handles the problem that the output of the synchronization circuit may be non-deterministic. Typical synchronizers achieve that goal by employing a bidirectional handshake protocol. The goal of formal verification is to guarantee correct operation of the synchronizer (rather than the synchronization circuit).

It is relatively easy to simulate a synchronizer using a logic simulator, and even demonstrate incorrect operation in some cases [7]. However, it is infeasible to prove correctness of the synchronizer by mere simulations. The number of different relative orderings of clock edges in a multi-clock environment could be extremely large and simulating all possible cases could be hard to generate and could take excessively long time. This observation leads to formal verification being the preferred method.

At the same time, it is relatively hard to define general properties that apply to every possible synchronizer and, once proven to hold, guarantee synchronization correctness. There are too many known protocols, numerous flavors of each protocol, and unlimited methods of implementing each flavor of each protocol. Instead, structural analysis is applied to recognize synchronizers and to sort them into several a-priori known types. For each type, a set of properties has been defined, which, when proven to hold, guarantee correctness. In this paper, we consider the verification of a simple two-flip-flop synchronizer and a dual-clock FIFO.

The paper describes how to generate formal verification executions of RuleBase (a model checker [7][8] using PSL [9]) for any multi-clock domain system employing the said types of synchronizers. We start with modeling of mutually asynchronous clocks in Section 2. Next, control verification methods are described in Section 3, and in Section 4 we introduce a more general way of extracting property specifications from Signal Transition Graph definitions of concurrent systems. An alternative method for data-transfer verification is discussed in Section 5.

2   Modeling multiple clocks

The model checker (MC) [8] performs its algorithms in atomic steps called *ticks*. Each synchronous component of the system being verified (the *design system*) is assumed to operate in atomic steps called *clock cycles*. Common model checking assumes a single clock, but synchronizers must be verified while observing multiple clocks. Thus, we need to add special modeling of multiple clocks to our specification.

The MC provides for three methods of clock modeling: A deterministic clock (where the ticks during which the clock changes state are pre-determined), a non-deterministic clock (where the ticks during which the clock changes state are not specified and are selected non-deterministically by the MC), and a constrained-choice clock (where the non-deterministic toggling of the clock is partly constrained by some specification). Using multiple non-deterministic clocks may result in scenarios that are impossible in the real design system, and verification may sometimes fail on such impossible scenarios (*false-negatives*). The set of all possible clock combinations covers the set of possible real scenarios. When the two sets are equal, there will not be any false-negatives.

Consider a design system employing sampling elements that use either positive or negative edge-triggered flip-flops (FFs). The MC offers two modes of modeling such FFs, edge-triggered and level-triggered modes. In *edge-triggered mode*, the FF samples its input on the rising (falling) edge of the clock, just as in the physical world. In *level-triggered mode* (which has no analog in the physical circuit), the clock input is considered an enabling signal for the FFs: Positive (negative) edge-triggered FFs sample on every tick when the clock is high (low) and are disabled otherwise (Fig. 2). When both types of FFs are present in the design system, level-triggered mode cannot model ticks during which no FF samples its input, and hence edge-triggered mode is preferred. In simpler cases, level-triggered is employed, as it is more convenient for clock modeling.
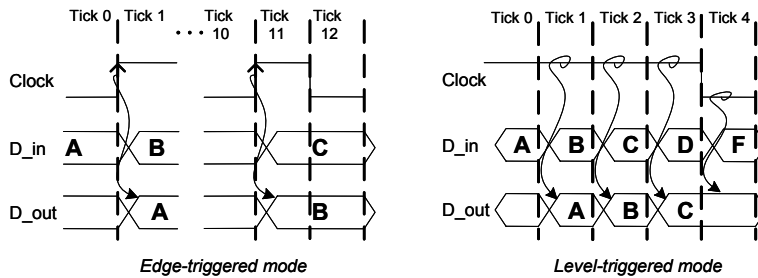

Fig. 2: Flip-flop modeling by edge- and level-triggered modes

Since the only way by which clocks can affect state changes is through their active edges (positive, negative, or both), we consider only active edges, and, in particular, the relative ordering of such edges. Two clocks may be unrelated, namely any arbitrary number of edges of one clock may happen between two successive edges of the other clock. In some cases we are interested in rational clocks, which maintain a rough frequency ratio to each other, as follows.

2.1　　　Mutually–asynchronous clocks

Unrelated, mutually-asynchronous clocks can be modeled as follows in PSL:

```
VAR CLK1, CLK2: 0..1;
fairness CLK1=1;
fairness CLK2=1;
```

The only constraint on non-determinism we introduce compels the clocks to create active edges an infinite number of times, to avoid stagnation of any one of the clock domains. This is our default model for clocks with unknown frequencies. It needs to be extended (namely, its non-determinism is constrained) only when it is desirable to eliminate false-negatives; rational clocks provide such an example.

2.2　　　Rational clocks

At times, the design system assumes a certain fixed frequency ratio of two clocks [10][11]. Consider the example of 3:2 frequencies of CLK1 and CLK2. In that case, there are one or two active edges of CLK1 between any two active edges of CLK2. This is modeled using a non-deterministic auxiliary cyclic counter:

```
var counter: 0..2;
var CLK1, CLK2: 0..1;
assign next(counter) := if (counter = 1) then {0, 2}
                        else (counter + 1) mod 3 endif;
assign CLK1 := counter != 0;
assign CLK2 := counter  = 0;
```

One possible trace of the resulting clocks is shown in Fig. 3(a), where edges are shown as up-arrows. A refined model also enables simultaneous edges of the two clocks, as demonstrated in Fig. 3(b):

```
assign CLK1 := if (counter != 0) then 1 else {0, 1} endif;
```

Note that this model may produce unrealistic relative orderings, in addition to the desired ones.
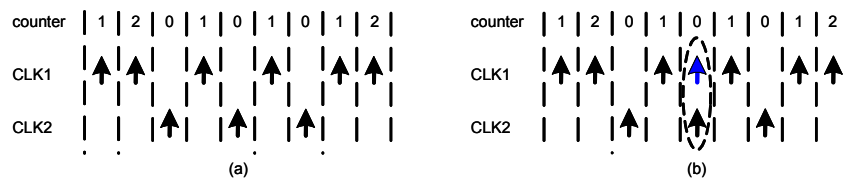

Fig. 3: A 3:2 clocking trace

In general, given two clocks CLK1 and CLK2 with frequency ratio of *m:n* (WLOG *m>n*), between any two active edges of CLK2 there should be *N* active edges of CLK1, where $\lfloor m/n \rfloor \leq N \leq \lceil m/n \rceil$. If *m* is divisible by *n* then: $(m/n\text{-}1) \leq N \leq (m/n+1)$. It is possible to cover in a single execution of MC a wide range of possible ratios *m/n* (including both *m>n* and *m<n* cases), if *m:n* is specified as a non-deterministic variable.

3 Control verification

As stated above, data transferred between two mutually asynchronous clock domains are wrapped by a handshake protocol, implemented with control signals between the domains. In this section we consider verification of the protocol by examining the control signals. We first verify the entire synchronizer, using multiple clocks. Next, we describe an alternative method in which the synchronizer is decomposed into the two clock domains and each part is verified separately.

3.1 The two flip-flop synchronizer

The most commonly used synchronizer is based on the well-known two-flip-flop (2FF) synchronization circuit (Fig. 1) [2][3]. This synchronizer appears in many flavors, one of which is shown in Fig. 4 (note that we distinguish between "synchronization circuits" and "synchronizer"—the latter includes two instances of the former). The sender issues a request R1, which gets synchronized using a 2FF synchronization circuit, yielding R2. (In high frequency designs, more than two flip-flops may be required [3][10].) The receiver then latches the data and sends an acknowledgement (A2) back to the sender, through another 2FF synchronization circuit. The data must be held stable in S_BUF as long as the request is true and the acknowledgement has not yet been asserted. A "push" synchronizer (where data is sent in the same direction as the "request" signal) is shown, but the same principles apply also to pull, push-pull, and control-only synchronizers.
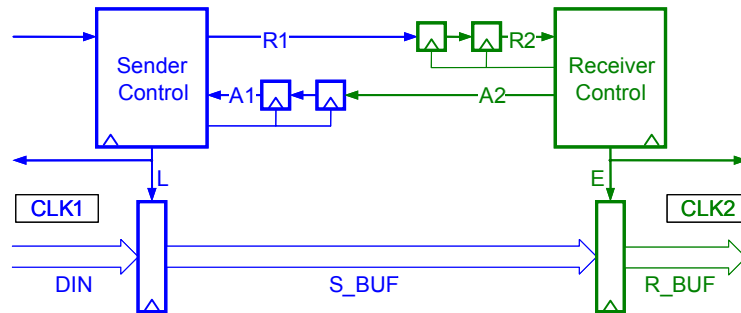
**Fig. 4: The two flip-flop synchronizer
(comprising two 2FF synchronization circuits)**

The desired synchronizer handshake protocols may be described by means of STGs (Signal Transition Graphs) that define order of events (logic level transitions) in the synchronizer [12]. One possible protocol is shown in Fig. 5.

In this protocol, two event sequences flow in parallel: data sampling (E+$\rightarrow$E-) and partial acknowledgement (A+$\rightarrow$R-). The fork in the STG enables several alternative scenarios, and verification rules should allow all of them, as derived in Section 4 below. Given a synchronizer (e.g. in terms of HDL), the verification task comprises three steps: (1) guessing which protocol is used by the synchronizer, (2) using the STG that specifies that protocol in order to generate verification rules, and (3) verifying the rules with the Model Checker.
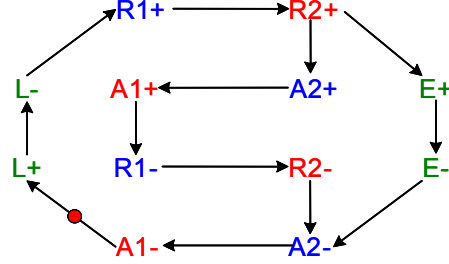
Fig. 5: STG of the 2FF synchronizer

3.2 Separate verification of two domains

Verifying a design with multiple clocks may incur a heavy computational load. We consider separate verification of these domains as a simpler alternative. The 2FF synchronizer can be easily partitioned into its constituent domains, as in Fig. 6. The cross-domain relations R1+→R2+, A2+→A1+, R1-→R2- and A2-→A1- are implied by the structure (2FF synchronization circuits) and are assumed verified by means of structural analysis. Thus, they do not need to be verified by assertions.
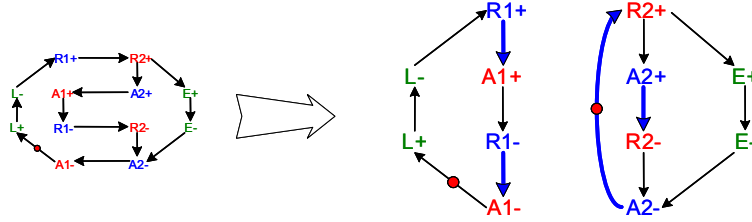

Fig. 6: STG of the 2FF synchronizer
decomposed into the Sender and Receiver clock domains

In the following, we present some verification rules that are derived informally from the two STGs. A formal approach is described in Section 4. We seek to verify that the behaviors made possible by the STGs (relative to the signals that are defined in the STGs) are followed by the circuit being verified, and that sequences that are prohibited by the STGs cannot happen in the circuit.

The rules are designed to verify that the STG events occur in a correct order and that each event happens only in the state(s) in which it is enabled. Consider the event L– in Fig. 6. L is the signal that loads data into the sender's register (Fig. 4). The event L– must take place exactly between L+ and R1+. The following rule verifies that:

```
AG (!A1 & !R1 & L -> fell(L) before! !(!A1 & !R1) )
```

Namely, once L is high, it should fall strictly before either R1 or A1 rise. The rule is violated if data load happens concurrently with the setting of R. This is indeed a real problem that can manifest itself in a system where the receiver is much faster than the sender. If L and R1 are asserted simultaneously (as some practical synchronizers tend to do…), L+ will load the data into the sender's register only at the end of CLK1

cycle, but meanwhile R1 may reach the receiver (recall $T_{CLK1} \gg T_{CLK2}$) and cause it to sample data even before it was actually sent [10].

It is insufficient to verify that data is loaded into the sender's register before handshake starts. We should also verify that there is no overrun by the next cycle:

```
AG ( fell(L) -> !R1 & !A1 )
```

In words, once L goes low, it should remain low until A1 is reset.

4   Converting synchronizer STG into verification rules

In this section we discuss how to convert the synchronizer STG directly into PSL assertions. We employ STG to specify synchronizer behavior, where each STG node represents a single switching event of some signal, and where arcs imply precedence. For our methods, the STG nodes have to be uniquely identifiable. For instance, two nodes that indicate the transition `x+` are tagged `x+/1, x+/2`. They can be distinguished by the state of all other variables in each case (some of which may be "don't care").

Note that similar properties are required for synthesis of STG into asynchronous circuits [14]. To verify that the given synchronizer complies with the STG, we prove that:

- Signal transition events (in the synchronizer) occur in the order specified by the STG.
- A signal transition event (in the synchronizer) occurs only in the states where it is allowed.

4.1   Identifying enabling states

Correct ordering of all events may be assured by the exhaustive proof of correct execution of all events. Each event has its own condition that enables its execution. In STG, the condition is fulfilled by a marking (a mapping of tokens to arcs) where all arcs incoming into the event carry tokens, enabling firing of the event. The condition is converted into a rule that verifies that the enabled transition actually takes place before the enabling state is changed:

```
AG ( EnablingState -> Transition(E) before !EnablingState )
```

To verify that events take place only in the proper states, we verify that

```
AG (Transition(E) -> SetOfEnablingStates)
```

For example, in Fig. 6:

```
AG ( !R2 & !A2 & !E -> rose(R2) before !( !R2 & !A2 & !E))
AG ( rose(R2) -> !A2 & !E )
```

Thus, an event (a STG node) is enabled to make its signal transition during state of some other signals. More generally, multiple states may be allowed at the time of this transition. Consider, for instance, the A2+ event in the Receiver domain of Fig. 6. To determine the set of states in which A2+ is enabled, one should identify the signals

that are guaranteed invariant when this transition can happen. While the leftmost token is placed on the incoming arc of A2+ (allowing A2+ to fire), the rightmost token may be located on any one of three different arcs: R2+$\rightarrow$E+, E+$\rightarrow$E-, or E-$\rightarrow$A2-. Thus, E may be either high or low when A2 rises. Hence, only the state of R2 is well defined during the event A2+ (namely, R2=1). Thus, the A2+ event is enabled by R2 & !A2.

Let's introduce some notations. For each event E (an STG node):

- *sig(E)* - the transitioning signal of E. E.g., *sig*(A2+)=A2.
- *st(E)* – the state of *sig(E)* when E is enabled. E.g., *st(*A2+*)*=!A2.
- *S(E)* - the disjunction of all states in which E is enabled, not including *st(E)*. E.g., *S*(A2+)=(R2&!E)|(R2&E)=R2.
- *T(E)* - the signal transition of E. E.g., *T*(A2+)=*rose*(A2).
- *P(E)* – (the set of paths parallel to paths that contain E): all paths that start at a predecessor of E, terminate at a successor of E, but do not include E. E.g. *P(*A2+*)*={…$\rightarrow$E+$\rightarrow$E-$\rightarrow$…}
- *C(E)* - all signals whose transitions are included in *P(E)*. These signals may change concurrently with E. E.g., *C*(A2+)=E.
- *W(E)*=*US* \ (*C(E)*∪{*sig(E)*}) where *US* is the universal set of signals: the set of the signals that cannot change concurrently with E. E.g., *W*(A2+)={R2}.
- *St_E(w)* - the state of signal w when E is enabled. This is meaningful only for $w \in W(E)$. E.g., St_{A2}(R2)=1.

The disjunction of states *S(E)* may be expressed as

$$S(\pmb{E}) = \bigwedge_{w \in W(\pmb{E})} (w = St_{\pmb{E}}(w)) \cdot$$

The following procedure generates $St_E(w)$ for each E and $w \in W(E)$. The `Enabled()` function returns all enabled events at a given marking of the STG. The `Fire(`E`)` function fires an event E in the STG (removing tokens from all arcs incoming into E and placing tokens on all arcs outgoing from E). The `LastTr[*]` array is used by the algorithm for holding the last STG-node covered for each signal. `Pred[*][*]` is a two-dimensional array that for each STG-node E and each signal u holds the STG-node that corresponds to the last transition of u covered before the firing of E. That array is built by exhaustive firing of the STG:

```
Pred[*][*] ← null
LastTr[*] ← null
While (∃E ∧ ∃u: Pred[E][u] == null)
{
  Foreach F ∈ Enabled() {
    Foreach signal v { Pred[F][v] ← LastTr[v] }
    Fire(F)
    LastTr[sig(F)] ← T(F)
  }
}
```

In a deterministic STG , there is only one well-defined state for each signal in $W(E)$. Thus, it is sufficient to determine these states by static analysis of the STG, as is done above. The following algorithm finds $W(E)$:

```
W(*) ← {}
Foreach event E
    Foreach signal u
    {
        Eᵤ = Pred[E][u]
        If foreach E'ᵤ such that sig(E'ᵤ)==u ∧ T(E'ᵤ)≠T(Eᵤ):
            ∃p=path(Eᵤ,E'ᵤ) ∧ E∈p then
            {
                W(E) ← W(E) ∪ {u}
                If T(Eᵤ)=rose(u) then Stₑ(u) ← 1
                                  else Stₑ(u) ← 0
            }
    }
```

If $E_u \in P(E)$ then $\exists E'_u$ such that $\forall p$=path(E_u, E'_u): $E \notin p$. Otherwise E_u is a predecessor of E and E must occur before any other opposite transition of $u \Rightarrow u$ belongs to E-invariant-set, which is a contradiction.

4.2   Generating verification rules

Having introduced the formalism in 4.1, we can write a rule of the following form for each STG event:

$$\text{AG}(\ S(E)\ \&\ st(E)\ \rightarrow\ T(E)\ before!\ !S(E)\ ) \tag{1}$$

The weak *before* (without !) operator is employed when the event E refers to an input signal (The circuit cannot compel an input). In order to verify that signal transitions take place only in their enabling states, we verify the following rules for each signal c:

$$\text{AG}(\ rose(c)\ \rightarrow\ \bigvee_{\forall E:T(E)=\text{rose}(c)} S(E))$$
$$\text{AG}(\ fell(c)\ \rightarrow\ \bigvee_{\forall E:T(E)=\text{fell}(c)} S(E)) \tag{2}$$

Eq. (1) and (2) constitute the set of verification rules for logic that may be specified by the STG. For instance, applying the equations to the C-element STG in Fig. 7 yields the following rules:

```
AG (!A & !C -> rose(A) before C )
AG (!B & !C -> rose(B) before C )
```

```
AG (A & B & !C -> rose(C) before! !(A & B) )
AG (A & C -> fell(A) before !C )
AG (B & C -> fell(B) before !C )
AG (!A & !B & C -> fell(C) before! !( !A & !B ) )
AG ( rose(A) -> !C)
AG ( fell(A) -> C)
AG ( rose(B) -> !C)
AG ( fell(B) -> C)
AG ( rose(C) -> A & B)
AG ( fell(C) -> !A & !B)
```
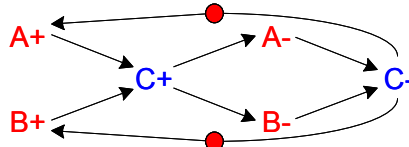

Fig. 7: C-element STG

5 Verifying data transfers

As we have seen, a synchronizer usually wraps the cross-domain data transfer by a control handshake protocol. The advantage of control verification of domain-decomposed synchronizers (Section 3.2) is that no multiple clock modeling is required. However, control verification suffers of the following disadvantages:

- Control verification is protocol specific. The rules are useful for verifying a certain synchronizer and cannot in general be applied to other synchronizers.
- The STG that specifies the control protocol may need to be changed in order to satisfy some properties, such as Complete State Coding [13], in order to enable rule derivation. The "well-behaved" STG is often more complex than the original one.
- Third, quite complex rules may be needed for verification, incurring a heavy MC computational load.

In the previous section we discussed control verification of handshake protocols. In this section we show how to prove that the actual data transfer is correct, without considering the control signals. If the controller has an error, it will be discovered through data verification. The goal of data transfer verification is to prove that any data item sent by the sender is eventually sampled exactly once by the receiver.

5.1 2FF data transfers

The data transfer part of a 2FF synchronizer is shown in Fig. 8. The verifier interprets the loading of data DIN into the leftmost register as an attempt by the sender to send it. A sampling into the rightmost register is interpreted as an attempt by the receiver to receive data. The verifier must prove that no data item is either missed or sampled more than once by the receiver.
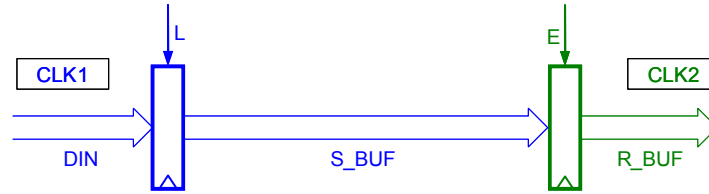
Fig. 8: Cross-domain data transfer structure

The first verification rule checks data integrity:

```
AG ( CLK1 & L & DIN(0)=1 ->
     next_event( CLK2 & E )( S_BUF(0)=1 ))
```

A similar rule can be written for the value 0. Integrity is checked only for a single data bit because all the other bits will behave in the same way, as guaranteed by structural verification. In addition to data integrity, we should verify that:

• Data is not duplicated—the receiver does not sample the data if the sender did not send any:

```
AG ( CLK2 & E -> AX ( (CLK1 & L) before (CLK2 & E) ))
```

• Data is not missed—the receiver eventually receives data that was sent by the sender:

```
AG ( CLK1 & L -> AX ( (CLK2 & E) before! (CLK1 & L) ))
```

In words, between any two sendings there must be one reception, and vice versa. The second assertion uses the strong *before!* operator (with !) to verify that the event (CLK2 & E) eventually takes place even if the subsequent event (CLK1 & L) does not happen at all.

5.2   Dual clock FIFO data transfers

The 2FF synchronizer incurs a long latency, and successive data transfers every clock cycle are impossible. Dual-clock FIFO synchronizers (Fig. 9) are typically employed when high throughput transfers are needed [2][7]. Data items are written by the sender into a dual port memory, and are subsequently read by the receiver. Write and read pointers are used to manage memory access. *Full* and *empty* indications should prevent any over- or under-runs; they are computed based on comparing the write and read pointers. Since these pointers belong to different clock domains, they must first be synchronized. Each pointer consists of several bits, depending on the FIFO size. Thus, it becomes necessary to synchronize multi-bit items (without resorting to the slow 2FF synchronizer).
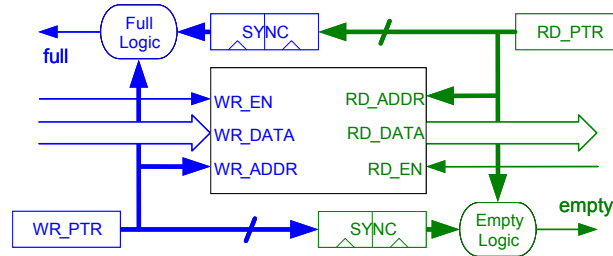
Fig. 9: A dual-clock FIFO synchronizer

A fast parallel multi-bit synchronization circuit is shown in Fig. 10. It faces the danger that more than one bit may become metastable at the same time. Going out of metastability, some of these bits may resolve to one and others may resolve to zero. Consequently, the combined multi-bit value is invalid, and in general such a synchronization circuit should be avoided [10]. However, if it is guaranteed that at most one bit changes state at any given cycle, this circuit is safe, in the sense that the receiver never receives a value that was not sent by the sender.
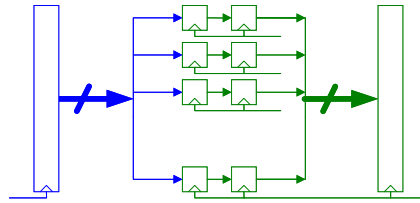

Fig. 10: A parallel synchronizer: Safe if only one bit changes at a time

The dual-clock FIFO employs Gray-code for its write and read pointers, assuring that at most one bit changes at a time. To verify the FIFO that contains such parallel synchronization circuits, we must allow for non-deterministic delay. We employ the following non-deterministic delay model for the FFs that constitute this parallel synchronization circuit. If the input changes at the same tick as the clock, the output is non-deterministically either the old or the new value (an earlier version has been proposed in [15]):

```
Module FF_nondet (/*INPUT*/ clk) (/*OUTPUT*/ q)
ASSIGN
          next(q) :=
             if (clk) then
               if (fell(d) | rose(d)) then {d , q}
               else  d endif
             else q endif;
```

The two FIFO clocks are unrelated, and therefore modeled as described in Section 2.1. The following rule verifies that any data written to the FIFO is eventually read out:

```
forall x: boolean:
  formula
  { AG ( wclk & wen & !full & wr_data(0)=x ->
    AF ( rclk & ren & !empty & rd_data(0)=x))}
```

The main difference between this formula and the one used above to validate the simple 2FF synchronizer is the awareness that since the FIFO may hold multiple data items, a written value may not be the one which is read on the next read access of the receiver. This is reflected by the use of the AF operator, meaning that any written value should be read out some time in the future. However, this check is insufficient, because it does not verify that the data items are read in the same order as they were written. To verify ordering, we introduce indexing, as follows:

```
define DEPTH := 4;
var widx : 0..(2^DEPTH -1);
ASSIGN
    init(widx):=  0;
    next(widx):=  if    (wclk & wen & !full)
                  then  (widx+1) mod DEPTH
                  else  widx endif;
```

`ridx` is defined similarly for the reader. The write and read indices are increased each time a value is written or read, respectively. The following formula uses those indices:

```
forall idx: 0.. (2^DEPTH -1):
forall x: boolean:
 formula
 { AG (( wclk & wen & !full & widx=idx & wr_data(0)=x)    ->
   next_event(rclk & ren & !empty & ridx=idx) (rd_data(0)=x))}
```

The dual clock FIFO synchronizer serves as an example for cross-domain data transfers through memory and for verifying parallel synchronization circuits.

6   Conclusions

We have demonstrated how to verify synchronizers using model checking. Rather than attempt to prove any synchronizer, we assume a given scheme and compare the synchronizer with its assumed specification. Synchronizer behavior is specified using signal transition graphs (STG). The STG is converted into verification rules, and the model checker applies these rules to the given synchronizer. If model checking fails, another STG specification may need to be considered.

An alternative method has been described, by which we verify correct data transfer through a synchronizer and seek any potential for missing or duplicating the data. Both approaches require specific modeling of multiple clocks, allowing for non-determinism in their relative ordering.

These methods have been applied successfully to a number of synchronizers, such as the two-flip-flop synchronizer and dual clock FIFOs, and including the more complex adaptive predictive synchronizer of [11].

Synchronizers may be subject to metastability effects. These effects may be contained by specific synchronization circuits, so that their failure probability is negligible. However, even when metastability is handled properly, synchronizers may still fail, if they encounter an unexpected order of events due to interfacing multiple

unrelated clock domains. Correct synchronization is independent of such order of events; we have shown how to employ model checking to verify that.

7 References

[1] A. Iyer and D. Marculescu, "Power Efficiency of Voltage Scaling in Multiple Clock, Multiple Voltage Cores", *IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*, pp. 379-386, Nov. 2002.

[2] W. J. Dally and J. W. Poulton, "Digital System Engineering", Cambridge University Press, 1998.

[3] L. Kleeman, A. Cantoni, "Metastable behavior in digital systems", *IEEE Design and Test of Computers*, pp. 4-19, Dec. 1987.

[4] E.M. Clarke, O. Grumberg and D.A. Peled, "Model Checking", The MIT Press, 2000.

[5] C. Dike, E. Burton, "Miller and noise effects in a synchronizing flip-flop", *IEEE J. Solid-State Circuits,* 34(6) , pp. 849-855, June 1999.

[6] T. Kapschitz, R. Ginosar, R. Newton, "Verifying Synchronization in Multi-Clock Domain SoC", *DVCon* 2004.

[7] Y. Semiat, R. Ginosar, "Timing measurements of synchronization circuits", *9th Int. Symp. Asynchronous Circuits and Systems (ASYNC)*, pp. 68-77, May 2003.

[8] I. Beer, S. Ben-David, C. Eisner, A. Landver, "RuleBase: an industry-oriented formal verification tool", *Design Automation Conference*, pp. 665-660 June 1996.

[9] M. Gordon, J. Hurd and K. Slind, "Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving," *CHARME*, LNCS 2860, pp. 200–215, 2003.

[10] R. Ginosar, "Fourteen Ways to Fool Your Synchronizer", *9th Int. Symp. Asynchronous Circuits and Systems (ASYNC),* pp. 89-96, 2003.

[11] U. Frank and R. Ginosar, "A Predictive Synchronizer for Periodic Clock Domains," *PATMOS*, LNCS 3254, pp. 402–412, 2004.

[12] J. Sparso, S. Furber, "Principles of Asynchronous Circuit Design", Kluwer Academic Publishers, Dec. 2001.

[13] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Complete state encoding based on the theory of regions", *2nd Int. Symp. Asynchronous Circuits and Systems*, pp. 36-47, March 1996.

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, E80-D(3), pp. 315– 325, 1997.

[15] K. Yorav, S. Katz and R. Kiper, "Reproducing Synchronization Bugs with Model Checking", *CHARME* 2001.