# Formal Verification of Mixed Synchronous Asynchronous Systems using Industrial Tools

Ghaith Tarawneh and Andrey Mokhov
School of Engineering, Newcastle University, UK
ghaith.tarawneh@ncl.ac.uk — andrey.mokhov@ncl.ac.uk

***Abstract*—Asynchronous circuits are pervasive in modern synchronous systems, but they are still designed and verified in isolation, using dedicated asynchronous design flows, formalisms and tools. We describe a method to verify gate-level asynchronous circuit implementations using formal verification tools and property languages for synchronous logic. We report observations and findings from applying this method to use case designs using an industrial and an open source formal verification tools for synchronous logic, and compare performance and verification capabilities against two verification tools for asynchronous circuits. Finally, we discuss the advantages and practical considerations of bridging synchronous logic verification tools to the domain of asynchronous circuits. Our main conclusion is that, while there are performance penalties, there is still significant value in enabling users to verify asynchronous circuits using tools that may be more familiar, trusted or more widely adopted.**

## I. Introduction

Software tool support is generally recognized as one of the main reasons hindering the wider adoption of asynchronous design by the industry [1][2]. Even though asynchronous design is based on rigorous theoretical foundations and has clear advantages, its isolated ecosystem of flows, tools and formalisms presents a formidable entry barrier to industrial users. It also introduces complexities in designing modern multi-clock and Globally Asynchronous Locally Synchronous Systems where pipelines, handshake circuits and other forms of asynchronous "glue logic" are used everywhere and expected to operate correctly with their synchronous neighbors.

One of the challenges with integrating synchronous and asynchronous (further referred to as "sync" and "async") components is performing system-level verification. While the independent sync and async verification flows and tools are mature and capable of verifying (sync or async) modules independently, verifying a mixed-timing sync-async system as a whole is non-trivial. Due to the separation of the two flows, compatible behavioral models of the sync/async parts of the system must be created and used to verify each other (Figure 1a). For example, the sync part can be coded as a Signal Transition Graph (STG) to be made compatible with STG-based asynchronous verification tools. This way, the sync/async implementations will be verified against the environmental models of their counter-parts, but the faithfulness of the models themselves will not be verified.

Based on the aforementioned, we argue that there is much value to be gained from being able to verify asynchronous circuits using synchronous design tools. First, this would enable designers to verify a mixed sync-async system as a whole, overcoming the need for environmental models. Second, it will simplify the verification process by unifying two independent flows. Third, it will address the wider issue of EDA tool support for asynchronous circuit by giving designers, particularly in industrial contexts, the option to use more familiar or trusted tools. While we know there are no fundamental reasons preventing the co-simulation (and consequently the verification) of mixed sync-async systems [3], [4], how and to what degree does this work in practice remain open questions.

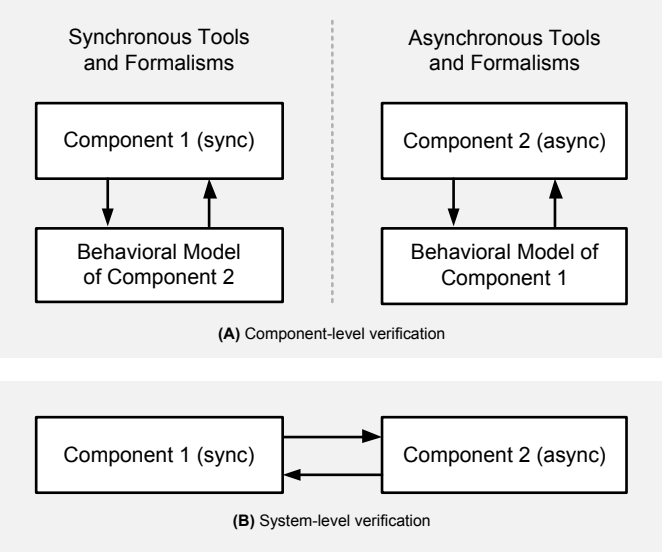
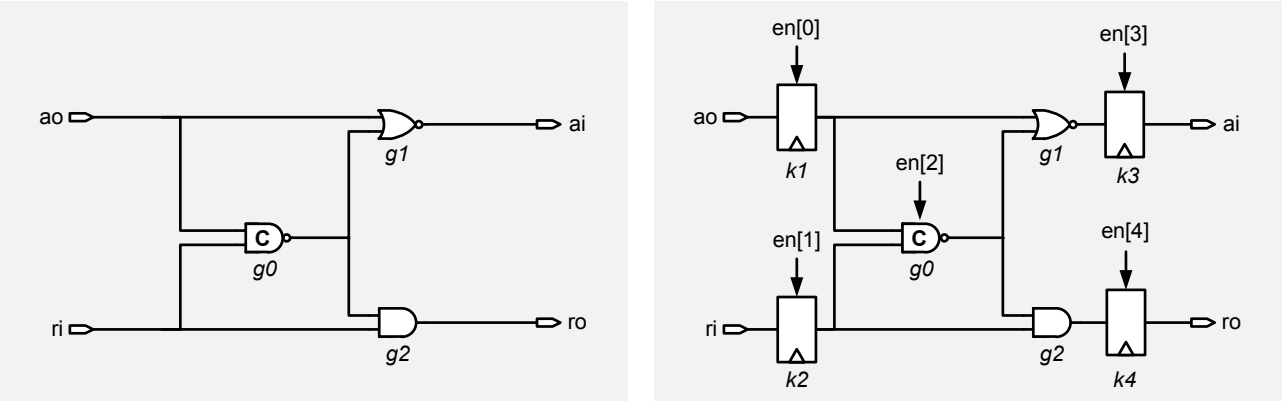



Fig. 1. Two approaches for verifying a mixed-timing system composed of a synchronous and asynchronous components. **(A)** The sync/async components are verified independently using behavioral models of their counterparts. **(B)** Both implementations are verified against each other directly.

### A. Main Idea

We propose a simple transformation to convert gate-level asynchronous circuits into clocked (synchronous) correspondents that can be verified using synchronous design tools.[1] Briefly, we insert flip-flops at asynchronous gate outputs and enable them one at a time to simulate transition firing. The created synchronous model is then used by a formal tool to explore the circuit's state space and verify its behavior.

[1]See Section IV-B for related work that exploits the same idea.




(A) Asynchronous Circuit

(B) Transformed Synchronous Circuit

Fig. 2. Proposed transformation to create a synchronous model of an asynchronous circuit. Flip-flops are inserted at inputs and gate outputs to capture net states, and corresponding signals (en) simulate transitions by enabling (up to) one flip-flop at a time.

### B. Contributions

The contributions of this paper are as follows. (i) We propose a transformation to convert asynchronous circuits into synchronous models, and describe methods to encode and check correctness properties using these models with synchronous verification tools. (ii) We report the results of using this methodology with industrial and academic verification tools for synchronous logic, cross-validating the results with MPSAT [5] and a custom tool (ESSET) which we developed for this purpose. (iii) We present a verification flow and use case example of mixed sync-async verification. Finally, (iv) we compare verification performance across three of the tools using a number of benchmark circuits.

## II. Proposed Transformation

### A. Overview

We demonstrate the proposed transformation using the circuit in Figure 2A as a working example. The circuit is a handshake decoupling element (an S-element [6]) and its state is given by nets `ao`, `ri`, `ai`, `ro` and the output of gate `g0`. In a conventional synchronous simulator, the circuit is treated as combinational logic and all of its nets are evaluated on each cycle. This does not allow individual transitions, their possible orderings or their timings with respect to the environment to be simulated. To capture these behavioral mechanics, we insert flip-flops at the outputs of all non-zero delay gates and use a vector of enable signals (`en`) to select which net is updated on each cycle. We prevent multiple transitions from firing during the same cycle by constraining `en` such that no more than a single bit is active at time. However, we allow all `en` bits to be inactive to simulate stall cycles in which no transitions occur (this is necessary for deadlock freeness checking, and is discussed in more details in Subsection III-B).

To use the created synchronous model in a simulation, inputs must be provided in accordance with circuit's specification (or better yet, generated by an actual implementation of the circuit's environment), and a sequence of `en` vectors must be supplied to determine transition ordering. While there could be more than a single valid `en` sequence (and supplying these by hand may be laborious) the model is primarily intended for formal verification and so our focus is to enable it to capture all possible forms of behavior. The task of generating `en` sequences and exploring the resulting circuit behavior will be left to formal tools.

Some gates such as C-elements are capable of retaining their own state and are therefore handled differently by the transformation. They are replaced with clocked equivalents and no flip-flops are inserted at their outputs. The clocked equivalents have an enable input which is connected to the corresponding bit from the enable vector `en`. An example of this special case handling is gate `g0` in Figure 2.

### B. Applying the Transformation

We created a tool that applies the described transformation to asynchronous circuits netlists such as the ones synthesized by Petrify [7]. While there are no reasons preventing the transformation from being performed by hand correctly, the ability to automate it has two important practical implications. First, it avoids manual translation errors that may introduce differences between circuit and model behavior (essentially the same risk with model faithfulness that system-level verification is intended to avoid). Second, it enables the transformation to be integrated seamlessly into existing design flows, remaining transparent to the designer and not affecting other tools in the flow (we present a use case verification flow demonstrating this practically in Section III-E). Another important practical consideration is that the vector `en` can be declared as an internal unbound register instead of being added to the circuit as an input. This way the original circuit and the generated model will have identical interfaces, allowing the model to be used as a drop-in replacement for the circuit.

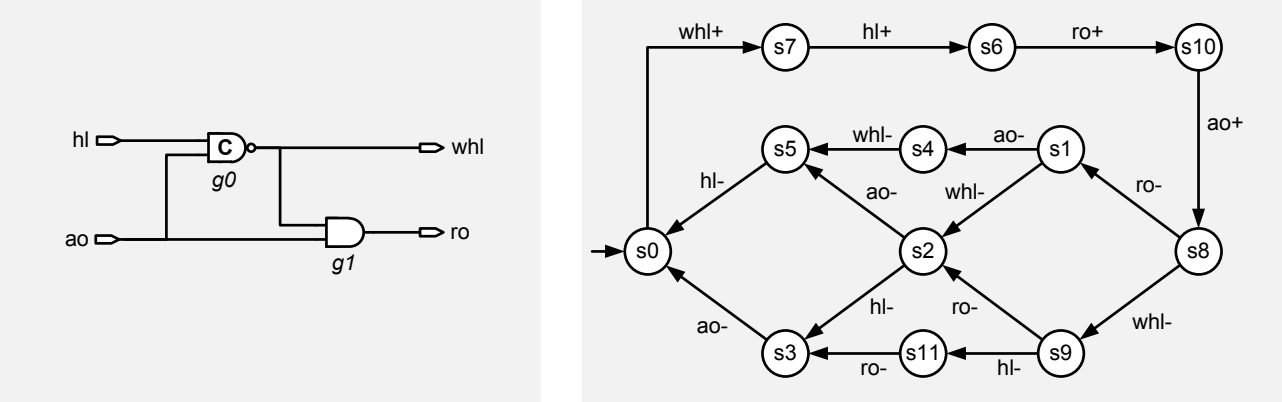



Fig. 3. Example asynchronous circuit (left) and its specification as a state graph (right)

## III. Applications

We discuss different types of verification checks enabled by the proposed transformation. Subsections III-A through III-D describe checks supported by asynchronous tools, which we here perform using synchronous tools, and Section III-E presents a use case verification of a mixed sync-async system.

The results reported below were obtained using two formal verification tools for synchronous logic: an industrial tool and Xprova [8] (an academic tool). Licensing restrictions prevent us from disclosing the identity of the industrial tool and so we refer to it as `IND_TOOL`. We cross-validated the results against MPSAT and an Exhaustive State Space Exploration Tool (ESSET) which we developed for this purpose.

### A. Spec Compliance

In compliance checking [9], circuit behavior is checked against a provided specification and differences are reported as compliance violations. For this check, we assume the input circuit is provided as a gate-level netlist alongside a State Graph (SG) representing its behavior and that of its environment. As an example, we use the High Load Handshake (HLH) circuit and its specification from [10], shown in Figure 3. To check for compliance, we transform the circuit as discussed in Section II and translate its SG into a synchronous Finite State Machine (FSM) in behavioral form, as follows:

```
always @(posedge clk or posedge rst) begin

  if (rst) begin
    state <= 0;
  end else begin
    if ( state == 0  &&  whl_p && en[0] ) state <= 7;
    if ( state == 7  &&   hl_p && en[1] ) state <= 6;
    if ( state == 5  &&  ~hl_p && en[1] ) state <= 0;
    if ( state == 3  &&  ~ao_p && en[2] ) state <= 0;
    // remaining transitions omitted for brevity
  end

end
```

This behavioral spec model is instantiated as a sub-component within the circuit, gaining access to its internal scope of signals.[2] Here, Verilog nets `whl_p`, `hl_p` and `ao_p` are inputs to the flip-flops holding the states of signals `whl`, `hl` and `ao` (respectively), `en` is the transition firing enable vector and `state` is an integer representing spec state.

During verification, the model is simulated in tandem with the circuit, acting as a reference for checking its behavior. To bind and compare circuit behavior to this reference, we generate properties that describe when circuit transitions may fire. For example, the following SVA property asserts that transition `whl-` occurs only when the spec model is in one of the states 1, 4 or 8 (see Figure 3, right panel):

```
wire whl_can_fall = (state == 1) | (state == 4) | (state == 8);

p1: assert property ( @(posedge clk) disable iff (rst)
    $fell(whl) |-> $past(whl_can_fall)
);
```

where the built-in SVA function `$fell` is high iff the input expression was just de-asserted, and `$past` holds the value of the input expression in the previous cycle.

We generate similar properties for all output transitions. In addition, since the circuit is expected to behave correctly only with respect to a certain environment (i.e. when input transitions follow the spec), the formal tool must also be told about input behavior. This is done by generating similar properties to describe input transitions, such as:

```
wire ao_can_fall = (state == 1) | (state == 2) | (state == 3);

p2: assume property ( @(posedge clk) disable iff (rst)
    $fell(ao) |-> $past(ao_can_fall)
);
```

Even though the two properties above are syntactically similar, `p1` is an assertion while `p2` is an assumption. During verification, the formal tool will explore all states in which assumptions are valid (valid circuit inputs) and report any assertion violations (invalid circuit outputs).

[2]This is done using the binding feature supported by many verification tools

This workflow is summarized in Figure 4. Briefly, the asynchronous circuit and its SG are translated into a verification model consisting of (1) a clocked implementation of the circuit, (2) a clocked behavioral FSM representing the SG and (3) properties that describe allowed transitions. The model is a self-contained unit with the same interface as the input asynchronous circuit, and can be passed to a conventional (synchronous) formal verification tool to prove or disprove compliance (as well as other correctness properties which we discuss in following subsections).

In our example (Figure 3), there are in total 4 compliance assertions (corresponding to the rise and fall transitions of outputs `whl` and `ro`). All four assertions received a pass state in both `IND_TOOL` and Xprova, consistent with the results reported by MPSAT and ESSET. We modified the circuit by changing `g1` into a NAND gate and re-checked; all tools now reported compliance violations for signal `ro`.

### B. Deadlock Freeness

Deadlocks are states with no enabled transitions. We express deadlock freeness as the following SVA property:

```
wire ao_may_fall = (state == 1) | (state == 2) | (state == 3);
wire hl_may_fall = (state == 2) | (state == 5) | (state == 9);

wire ao_may_rise = (state == 10);
wire hl_may_rise = (state == 7);

wire ao_may_trans = ao_may_rise | ao_may_fall;
wire hl_may_trans = hl_may_rise | hl_may_fall;
wire ro_may_trans = ro_p ^ ro;
wire whl_may_trans = whl_p ^ whl;

wire exist_enabled_transition = ao_may_trans | hl_may_trans
    | ro_may_trans | whl_may_trans;

deadlock_free: assert property (
   @(posedge clk) disable iff (rst) exist_enabled_transition
);
```

In the above, we declare and define four Verilog wires (in the form `x_may_trans`) to indicate whether the corresponding signals can transition during the current state. We assert that at least one transition is enabled on each simulation cycle.

Note that we define `x_may_trans` differently for input and non-input (internal + output) signals. In this example, we verify the circuit against its spec and so enabled input transitions are defined with respect to the spec's `state` variable. For internal and output signals (in this case only `ao` and `hl`), transitions are enabled if their corresponding flip-flops have different input and output values (i.e. pending transitions that await firing). Another important detail is that the model must be allowed to stall (to not fire any transition) by allowing all `en` bits to be low. If stall cycles are not allowed then the formal tool will not reach or detect any deadlock states, by definition.

We ran the deadlock check described above on the HLH circuit (Figure 3) and the assertion received a pass state in all four tools. Afterwards, we changed `g1` into a NAND gate and observed that all tools reported a deadlock violation in the modified circuit.

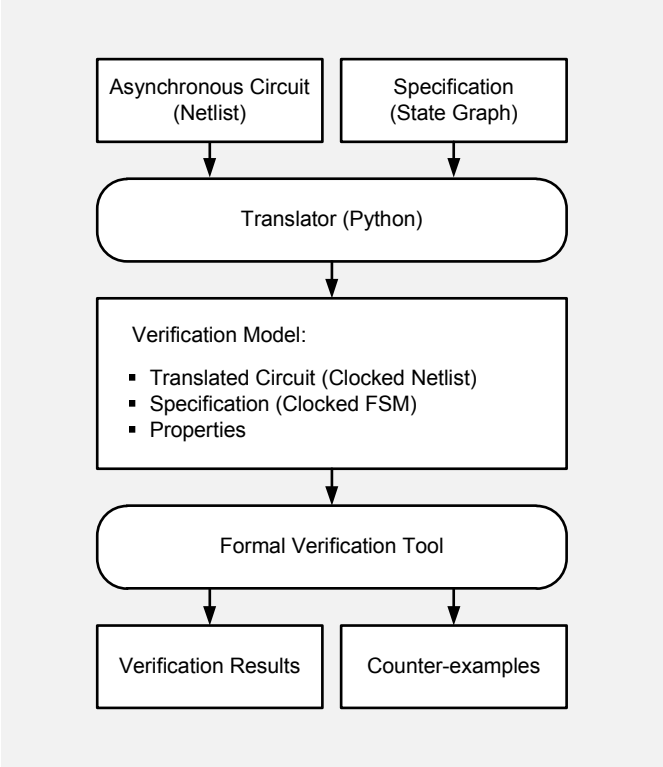



Fig. 4. Verification flow showing how the proposed tranformation enables asynchronous circuits to be verified using formal tools for synchronous logic

### C. Output Persistency

An internal or output signal is persistent iff, once its transition is enabled, the transition either fires or continues to be enabled [9]. This property is encoded as follows:

```
wire ro_may_trans = ro_p ^ ro;
wire whl_may_trans = whl_p ^ whl;

persistency_ro: assert property (
    @(posedge clk) disable iff (rst)
       $fell(ro_may_trans) |-> $changed(ro)
);

persistency_whl: assert property (
    @(posedge clk) disable iff (rst)
       $fell(whl_may_trans) |-> $changed(whl)
);
```

In the above, we re-use the definitions `ro_may_trans` and `whl_may_trans` (used in the deadlock freeness property), alongside SVA's internal functions `$fell` and `$changed`, to assert that, on each cycle where a transition of `x` has just been disabled (`$fell(x_may_trans)` is high), the transition fired on the same cycle (`$changed(x)` is high). In other words, firing is the only allowed mechanism to disable transitions.

The HLH circuit (Figure 3) passed output persistency checks in all tools, and replacing `g0` to with a non-inverted variant caused all tools to report persistency violations for output `whl`.

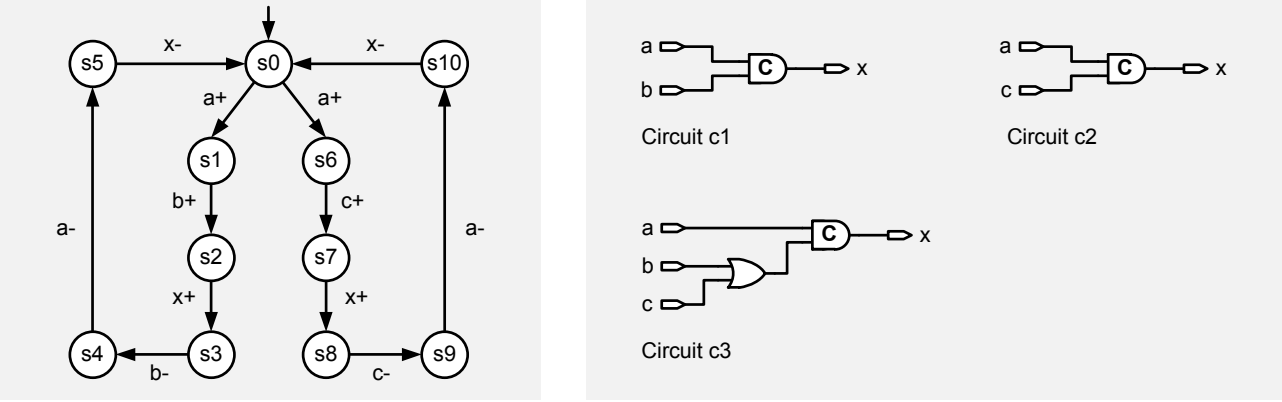



Fig. 5. Specification with non-determinstic choice (two a+ transitions at state s0) (left) and three possible implementations (right). The transitions undergone by all three circuits are within the spec, but only c3 captures the spec fully.

### D. Non-deterministic Choice

Asynchronous circuit specifications may include non-deterministic choice; cases in which identical input transitions diverge from the same source to different destination states. These cases represent an additional complexity to formal verification tools. Now it is insufficient to just check that all circuit transitions are captured by the spec; we must also check that all spec transitions are captured by the circuit.

To illustrate this, consider the spec and possible implementations c1 - c3 shown in Figure 5. The transitions of circuits c1 and c2 are within spec but neither captures the full spec (circuit c1 captures states s1 through s5 but not s6 through s10, and the converse is true for c2). Circuit c3 is the only correct implementation since it captures the entire spec. The core of the problem here is that the environment does a non-deterministic choice in state s0 with two outgoing transitions labeled by the same event a+. Both of these transitions must be explored during verification.

We can force formal tools to investigate all possible spec branches by adding unbound variables to the verification model and using them to decide between identical spec transitions. Since these variables are unbound, the formal tool will explore all their possible values, amounting to checking the circuit against all possible spec regions emanating from non-deterministic choice forks. For example, the spec in Figure 5 is translated to the following FSM behavioral code:

```
reg nond0; // unbound variable

always @(posedge clk or posedge rst) begin

    if (rst) begin
        state <= 0;
    end else begin
       if ( state == 0 && a_p && en[0] &&  nond0 ) state <= 1;
       if ( state == 0 && a_p && en[0] && ~nond0 ) state <= 6;
         // remaining transitions omitted for brevity
    end

end
```

Here we add and use an unbound variable nond to decide, at state s0, between transitioning to s1 or s6. The tool will then check the circuit for compliance, deadlock freeness and persistency across all spec branches. We have used this approach to check the three circuits c1, c2 and c3 against the spec in Figure 5 and confirmed that only c3 is correct with respect to the spec. This result was consistent across all tools. Without the support for non-deterministic choice, the tool could have erroneously accepted one of the incorrect implementations c1 or c2 (depending on which branch happened to be explored).

### E. Mixed Sync-Async Verification

In addition to verifying asynchronous circuits independently, the proposed transformation also enables us to verify systems composed of both sync and async components. We do this by translating asynchronous modules and combining them with the remaining (synchronous) parts of the system. The generated system netlist can then be passed to a synchronous formal verification tool, alongside a system-level specification provided by the designer (Figure 6).

As a use case example, we consider the system shown in Figure 7, where three CPUs use a shared bus to communicate with a memory module. The CPUs are optimized for either high performance or low power each, and are enabled in different combinations, no more than two at a time, depending on workload and battery level. A power management unit is used to control which CPUs are active at any time.

In this example, the CPUs run on independent clocks so an asynchronous 3-way arbiter is used to mediate bus access. A CPU can access the bus only when its request has been granted by the arbiter. We wanted to use system-level formal verification to prove the following two properties:

1) no more than a single CPU can be granted bus access at any time, and
2) the system is deadlock free.

Following the verification flow in Figure 6, we first translated the arbiter into a clocked netlist and combined it with the remaining modules to create a synchronous system netlist.

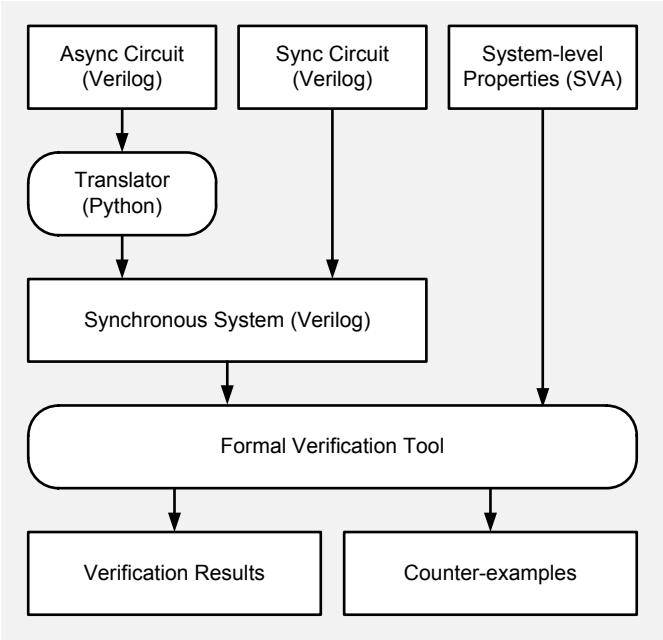



Fig. 6. Verification flow for a mixed sync-async system

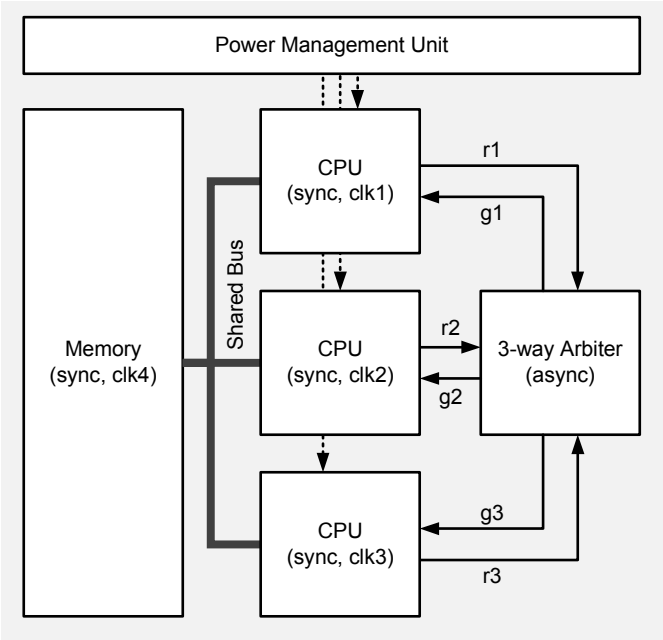



Fig. 7. Use case example for mixed sync-async verification: a multi-clock domain system with an asynchronous arbiter.

Next, we expressed the two properties above in SVA. Deadlock freeness was encoded as described in Section III-B while the absence of bus access conflicts was encoded as follows:

```
wire b1 = r1 & g1; // cpu 1 using bus
wire b2 = r2 & g2; // cpu 2 using bus
wire b3 = r3 & g3; // cpu 3 using bus

no_bus_access_conflict: assert property (
    @(posedge clk1 or
      posedge clk2 or
      posedge clk3 or
      posedge clk4) disable iff (reset)
        $onehot0({b1, b2, b3})
);
```

where the function `$onehot0` returns true if at most one bit of the input expression is high.[3]

The system uses a flat arbiter implementation from [11] (Figure 8). This implementation has the advantage of being simpler than its alternatives but suffers from a hidden caveat. It may enter a deadlock state if all three requests arrive at the same time (one transition sequence leading to this is `r1+`, `r2+`, `r3+`, `M1a+`, `M2b+`, `M3a+`). However, since the system's power management unit is designed to prevent all CPUs from being active simultaneously, we expect this to never happen in this environment. We wanted to prove this formally.

Our results from running this example were as follows. Both `IND_TOOL` and Xprova proved that (1) no more than a single CPU can access the bus simultaneously and (2) the system does not enter a deadlock state, consistent with

[3]In this system, CPUs relinquish bus access before de-asserting their request signals, allowing the arbiter to grant access to other pending requests without delay (early release protocol [11]). We therefore define bus access as the conjunction of a CPU's request and grant signals.

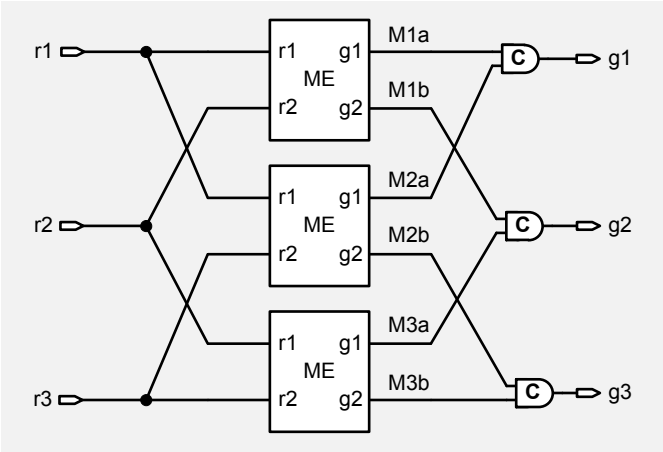



Fig. 8. Implementation of the 3-way arbiter in Figure 7

our expectation. We modified the power management unit to introduce a fault and allow all three CPUs to be active at the same time. This was discovered by both tools – both generated counter-example waveforms showing deadlock occurrences.

This example demonstrates two points. First, we are able to perform system-level verification of a mixed sync-async system using formal tools for synchronous logic. Second, system-level verification can be used to prove or disprove properties that cannot be verified at the component-level without making unverified environmental assumptions. In our case, we proved that the arbiter does not enter a deadlock state by verifying it against other modules directly. If we verified the sync and async parts separately instead (as shown in Figure 1A), we would need to create an accurate formal model of the power management unit – a very non-trivial task.

TABLE I
BENCHMARK RESULTS

| Circuit | N | States | Verification Time (sec) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | IND_TOOL | | | MPSAT | | | ESSET | | |
| | | | Deadlock | Compliance | Persistency | Deadlock | Compliance | Persistency | Deadlock | Compliance | Persistency |
| Async Counter | 8 | 3570 | 11.4 | 25 | 141 | 8.7 | 8.8 | 10.1 | 0.68 | 0.69 | 0.68 |
| | 9 | 7154 | 12 | 27 | 272 | 30 | 30 | 37 | 1.9 | 1.9 | 1.9 |
| | 10 | 14322 | 27 | 56 | 672 | 133 | 133 | 159 | 5.6 | 5.6 | 5.6 |
| C-element | 8 | 512 | 19 | 63 | 28 | 0 | 0 | 0 | 0.2 | 0.2 | 0.2 |
| | 9 | 1024 | 46 | 73 | 72.8 | 0 | 0 | 0 | 0.26 | 0.26 | 0.26 |
| | 10 | 2048 | 204 | 245 | 239 | 0 | 0 | 0 | 0.48 | 0.48 | 0.48 |
| Ring Oscillator | 21 | 42 | 5.7 | 9.3 | 4.66 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 31 | 62 | 6 | 25.8 | 11.2 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 41 | 82 | 6.75 | 55.7 | 17.3 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 51 | 102 | 7 | 108 | 25.6 | 0 | 0 | 0 | 0 | 0 | 0 |

## IV. DISCUSSION

### A. Performance

The proposed approach provides a workaround solution to verify asynchronous circuits using tools that were not built for this purpose. It is therefore expected that verification performance will be lower compared to dedicated asynchronous verification tools. In this section we attempt to answer two questions: (1) what is the magnitude of performance loss? and (2) what can we still verify in feasible time?

One difficulty with answering these questions is that the four verification tools available to us (that were used in this work) use different verification approaches that perform better with different types of circuits. For example, MPSAT uses the unfolding technique [12] which makes it faster when processing highly-concurrent circuits, at the expense of an overhead for sequential circuits. These differences confound attempts to establish the overhead of our approach by comparing performance across tools. Since performance is likely to depend on circuit type, we have selected three types of benchmark circuits with different scaling characteristics for our comparison:

1) $N$-bit counters ($O(N)$ signals, $O(2^N)$ states, fully sequential),
2) $N$-way C-elements ($O(N)$ signals, $O(2^N)$ states, fully concurrent), and
3) $N$-stage ring oscillators ($O(N)$ signals, $O(N)$ states, fully sequential).

Table I compares verification times for three of the tools we used (we excluded Xprova from benchmarking since it cannot verify designs with more than 64 state bits). Based on these results, our conclusions are as follows. First, for circuits with low degrees of concurrency (e.g. async counters), IND_TOOL (with the proposed conversion) has comparable/faster performance compared to MPSAT in deadlock and compliance checks, likely because MPSAT is constructing unfoldings for these purely sequential circuits. Verification time is still larger for output persistency checks, and larger in general when comparing with ESSET (which relies on explicit state space enumeration). Second, for circuits with high degrees of concurrency, MPSAT performs much better than IND_TOOL (< 0.1 second vs. up to 239 seconds). ESSET performs much better too (< 1 second), although still noticeably lower than MPSAT. Third, IND_TOOL performs poorly when the number of circuit components/signals is large, even though when there are few states. In ring oscillator benchmarks, verification time was 5.7+ seconds for circuits with as few as 42 states. This is likely because IND_TOOL explores all possible values of a considerably large en vector, as opposed to MPSAT and ESSET which maintain internal lists of enabled transitions and do not have to re-compute them on each cycle.

In general, benchmark results indicate that there is a considerable performance overhead when using synchronous tools to verify highly concurrent asynchronous circuits, compared to asynchronous tools. Even though asynchronous circuits are often highly concurrent, many mixed systems consisting of relatively small-sized asynchronous circuits are still well within the scope of what can be feasibly verified using synchronous tools and the proposed approach. For example, the mixed system discussed in Section III-E was verified by IND_TOOL in less than a minute, despite containing several sync modules (in addition to the async arbiter). We observed similar results from verifying similar systems [10] where the number and size of asynchronous handshake circuits and pipeline controllers was relatively small. For such systems, we argue that the increase in verification time is offset by the advantage of being able to verify asynchronous circuits *in-situ*.

### B. Related Work

Our approach is inspired by prior research by the asynchronous circuits community, in particular:

- Roncken et al. [13] use go signals to control progress in asynchonrous circuits in a fine-grained manner for the purpose of silicon test and debug. The idea is further developed in [14], where go signals are used to model non-determinism in asynchonrous circuits in the context of formal verification of *link-joint* models using the theorem proving system ACL2.
- Dobkin et al. [15] present an algorithm for converting STGs into sets of assertions written in the Property Specification Language (PSL), which can be used by standard assertion-based verification tools for synchronous designs. This approach allows the designer to verify the correct behaviour of synchronous circuits against a model of the asynchonrous part of the system.

In general, the idea of clocking an asynchonrous circuit is not new. [16] presents a synchronous back-end for the Tangram compiler with two aims: (i) providing a fast approach for prototyping asynchronous circuits using synchronous FPGAs, and (ii) reducing the risk of adoption of asynchonrous design methodology in industry by supporting the synchronous mode of execution as a fall-back scenario. [17] introduced a systematic approach for testing and debugging of asynchonrous circuits by incorporating conventional synchronous scan-chains. Elastic circuits [18] provide a way to achieve many of the benefits of asynchonrous circuits through conventional synchronous design flow. This paper follows this direction of research but in the context of formal verification.

There are existing industrial design flows relying on conventional EDA tools for mixed sync-async systems. For example, [19] describes the design flow developed by Tiempo, but without providing implementation details of the underlying formal verification methodology. Proteus [20] is another example of an industrial design flow developed by TimeLess Design Automation, which uses CSP as the specification language [21]. As described in [20], Proteus has no support for formal verification of asynchonrous circuits, but uses cosimulation and proprietary coverage tools to ensure that an implementation-environment pair matches the CSP specification.

## V. Conclusion

We describe a set of transformations and property encodings to verify asynchronous circuits using formal tools for synchronous logic. The transformation gives designers the option to use conventional synchronous tools, an important consideration for industrial users, and enables them to verify mixed sync-async systems using a unified verification flow, set of tools and formalisms. We demonstrate the method practically by verifying a use-case multi-clock system consisting of three synchronous CPUs, a shared bus and an asynchronous arbiter. Even though dedicated asynchronous tools are much faster at verifying asynchronous circuits that have high degrees of concurrency, the method remains a feasible and attractive solution for many realistic mixed sync-async systems in which the asynchronous parts are relatively small.

## Acknowledgments

This work was supported by EPSRC grant EP/N031768/1 (project POETS).

## References

[1] A. Kondratyev and K. Lwin, "Design of asynchronous circuits using synchronous CAD tools," *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 107–117, 2002.

[2] S. M. Nowick and M. Singh, "Asynchronous design – Part 2: Systems and methodologies," *IEEE Design & Test*, vol. 32, no. 3, pp. 19–28, 2015.

[3] R. Milner, *On relating synchrony and asynchrony*. University of Edinburgh. Department of Computer Science, 1980.

[4] ——, "Calculi for synchrony and asynchrony," *Theoretical computer science*, vol. 25, no. 3, pp. 267–310, 1983.

[5] V. Khomenko, M. Koutny, and A. Yakovlev, "Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT," *Fundamenta Informaticae*, vol. 70, no. 1, 2, pp. 49–73, 2006.

[6] A. Bardsley, *Implementing Balsa handshake circuits*. University of Manchester, 2000.

[7] 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*, vol. 80, no. 3, pp. 315–325, 1997.

[8] G. Tarawneh and A. Mokhov, "Xprova: Formal Verification Tool with Built-in Metastability Modeling," in *2017 17th International Conference on Application of Concurrency to System Design (ACSD)*, June 2017, pp. 74–79.

[9] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, and A. Yakovlev, "Automated verification of asynchronous circuits using circuit Petri nets," in *Asynchronous Circuits and Systems, 2008. ASYNC'08. 14th IEEE International Symposium on*. IEEE, 2008, pp. 161–170.

[10] D. Sokolov, V. Khomenko, A. Mokhov, A. Yakovlev, and D. Lloyd, "Design and verification of speed-independent multiphase buck controller," in *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*. IEEE, 2015, pp. 29–36.

[11] A. Mokhov, V. Khomenko, and A. Yakovlev, "Flat arbiters," *Fundamenta Informaticae*, vol. 108, no. 1-2, pp. 63–90, 2011.

[12] V. Khomenko, "Model checking based on prefixes of petri net unfoldings," 2003.

[13] M. Roncken, S. M. Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland, "Naturalized communication and testing," in *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*. IEEE, 2015, pp. 77–84.

[14] C. Chau, W. A. Hunt, M. Roncken, and I. Sutherland, "A Framework for Asynchronous Circuit Modeling and Verification in ACL2," in *Haifa Verification Conference*. Springer, 2017, pp. 3–18.

[15] R. Dobkin, T. Kapshitz, S. Flur, and R. Ginosar, "Assertion based verification of multiple-clock gals systems," in *Proc. IFIP/IEEE Int. Conference on Very Large Scale Integration (VLSI-SoC)*, 2008.

[16] A. Peeters and K. Van Berkel, "Synchronous handshake circuits," in *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*. IEEE, 2001, pp. 86–95.

[17] K. van Berkel, A. Peeters, and F. te Beest, "Adding synchronous and LSSD modes to asynchronous circuits," *Microprocessors and Microsystems*, vol. 27, no. 9, pp. 461–471, 2003.

[18] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1437–1455, 2009.

[19] A. Yakovlev, P. Vivet, and M. Renaudin, "Advances in asynchronous logic: From principles to GALS & NoC, recent industry applications, and commercial cad tools," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 1715–1724.

[20] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC flow for GHz asynchronous designs," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[21] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010.