Improving the Efficiency of Formal Verification: The Case of Clock-Domain Crossings

Guillaume Plassan[1,2(✉)], Hans-Jörg Peter[1], Katell Morin-Allory[2], Shaker Sarwary[1], and Dominique Borrione[2]

[1] Synopsys Inc., Mountain View, USA
{guillaume.plassan,HansJorg.Peter,shaker.sarwary}@synopsys.com
[2] Univ. Grenoble Alpes and CNRS, TIMA Laboratory, 38031 Grenoble, France
{katell.morin-allory,dominique.borrione}@univ-grenoble-alpes.fr

Abstract. We propose a novel semi-automatic methodology to formally verify clock-domain synchronization protocols in industrial-scale hardware designs. To establish the functional correctness of all clock-domain crossings (CDCs) in a system-on-chip (SoC), semi-automatic approaches require non-trivial manual deductive reasoning. In contrast, our approach produces a small sequence of easy queries to the user. The key idea is to use counterexample-guided abstraction refinement (CEGAR) as the algorithmic back-end. The user influences the course of the algorithm based on information extracted from intermediate abstract counterexamples. The workload on the user is small, both in terms of number of queries and the degree of design insight he is asked to provide. With this approach, we formally proved the correctness of every CDC in a recent SoC design from STMicroelectronics comprising over 300,000 registers and seven million gates.

Keywords: Formal verification · Clock-domain crossing · Synchronizers · CEGAR · SOC

1 Introduction

Modern large hardware designs typically contain tens of clock domains: different modules use different clocks, adapting consumption and performance to the ongoing tasks, thereby reducing the overall power consumption of the chip.

Moreover, an SoC typically assembles IP blocks coming from various teams, and each block may be optimized for a specific operating frequency. As a result, such architectures create many interconnections between the various clock domains, so-called *clock-domain crossings* (CDCs). To ensure a correct propagation of data through a CDC, hardware designers have to implement specific protocols and modules: *synchronizers*.

With the increasing number of CDCs and synchronization protocols as well as the huge complexity of modern SoCs, proving the functional correctness of

T. Hollstein et al. (Eds.): VLSI-SoC 2016, IFIP AICT 508, pp. 108–129, 2017.
DOI: 10.1007/978-3-319-67104-8_6

all synchronizers became a major challenge. While incomplete functional verification methods, such as testing based on simulation, scale for large designs, they are only able to show the absence of functional errors in a subset of the full design behavior. For exhaustively and automatically proving the correctness of functional properties, *model checking* is the prevalent technique in a modern VLSI design flow.

But, as model checking a property is not scalable on large hardware designs, the question arises whether it is really necessary in practice to have a *completely* automatic verification procedure. That is, can we somehow take the user into the loop and abandon the high degree of automation of model checking to make formal verification scalable?

This paper addresses this question and proposes a new comprehensive methodology for verifying clock-synchronization properties over industrial-scale SoC hardware designs.

Unlike other semi-automatic approaches that require non-trivial manual work in form of deductive reasoning, our approach produces a small sequence of easy queries that only require *local* design knowledge from the user. The key idea is to use the CEGAR principle [8] as the algorithmic back-end, where we let the user influence the course of the algorithm based on information extracted from intermediate abstract counterexamples. The workload on the user is deliberately kept small both in terms of number of queries and the degree of design insight to be provided.

More precisely, this paper makes the following contributions:

- A general semi-automatic algorithm based on CEGAR-based model checking.
- A heuristic to automatically infer design constraints from abstract counterexamples, which are then proposed to the user.
- A comprehensive methodology for verifying CDCs, based on this interaction between the model checking algorithm and the user.
- The application of this new methodology to conclusively prove the correctness of all CDCs on a recent industrial SoC design.

The paper continues as follows. Section 2 recalls the CDC challenges and the various synchronizers. Section 3 presents the state-of-the-art CDC verification flow and its limitations. We provide a novel automated flow in Sect. 4, and the results obtained with it in Sect. 5. We finally compare our approach with the related works before concluding the paper.

2 Clock-Domain Crossing Issues

A CDC typically manifests itself as a digital signal path between two sequential elements receiving clocks from out-of-phase domains (Fig. 1). Even if those clocks have the same frequency, any difference between their phases introduces a latency between their rising edges. This non-predictable behavior is precisely the challenge of designing CDCs.

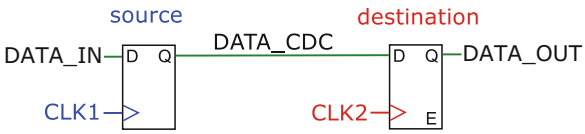


Fig. 1. A simple CDC

Multiple problems arise from CDCs [25], and designers need to implement specific structures to avoid any issue [13]. Consequently, the CDC verification tool must check that all the potential problems have been addressed and corrected.

2.1 Metastability and Multi-flops

The definition of clock domains directly implies that when data changes in the source domain of a CDC, the destination register can capture it at any moment: compliance with the setup and hold time requirements is not guaranteed. Hence, because of a small delay between the rising edges of the two clocks, the data is captured just when it changes, and a metastable value may be propagated (as shown in Fig. 2).

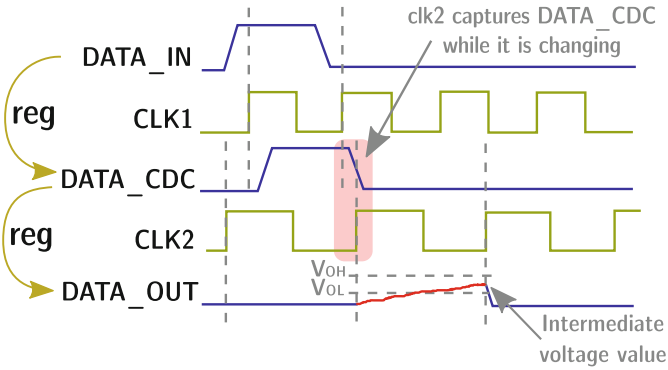


Fig. 2. Metastability behavior

The metastability phenomenon has been identified a few decades ago [7]: if a metastable value is propagated through combinational logic, it can lead to a so-called *dead system*. And it would be very difficult to find the source of this issue after fabrication, as post-production testers do not understand non-binary values.

A first solution would be to introduce a latency in the destination domain, in order to wait for a stabilization of the value. This timing can be estimated

by considering the clock frequencies and the production technology, as is commonly performed in the *Mean Time Between Failure* computation [12]. A single dedicated register could then, if properly sized, output a stable data. However, such synchronizing registers would result in a significant overhead on the circuit size. Another technique [16,20] involves embedding a monitor in the design which detects and corrects metastable values. However, the overhead would also be significant.

The most common solution is to add latency by implementing cascaded registers [14] (see Fig. 3). While this *multi-flop* structure guarantees within a certain probability that the propagated value is stable, there is no way of telling if it is a '0' or a '1'. Indeed, the data being captured during a change, the multi-flop may output the old or the new value during one cycle; then, at the next destination cycle, the new value is propagated. The drawback of this structure is thus a delay in the data propagation.

2.2 Coherency with Gray-Encoding or Enable Control

When synchronizing buses, there can be *coherency* issues: if some bits of a bus have separate multi-flop synchronizers (see Fig. 3), it cannot be guaranteed that all these synchronizers require a strictly identical latency to output a stable value. When capturing a toggling signal, some multi-flops may settle to the old value and some to the new one. The resulting bus value may then become temporarily incoherent. If multiple synchronized bits converge on a gate, a transient inconsistent value may even be generated.

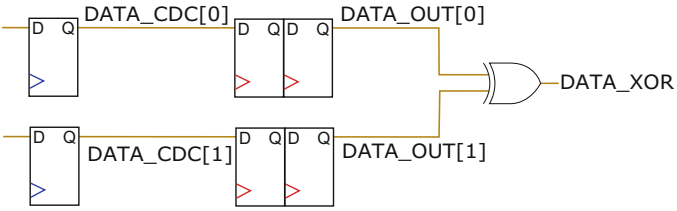


Fig. 3. Bus synchronization

For instance, in Fig. 4, two bits of *DATA_CDC* are toggling at the same cycle. After being synchronized by separate multi-flops, the *DATA_OUT* bus value is not consistent anymore. If both bits are converging on an exclusive OR gate, a glitch can be observed. However, we can add some encoding so that only one bit can change at a time [10] (Gray-encoding in the case of a counter, or mutual exclusion in some other cases). Even if the multi-flops stabilize this toggling signal with different latencies, the bus output value will either be correct regarding the previous or the next cycle. Thus, no false value is propagated. This can create some data loss, but avoids incoherency.

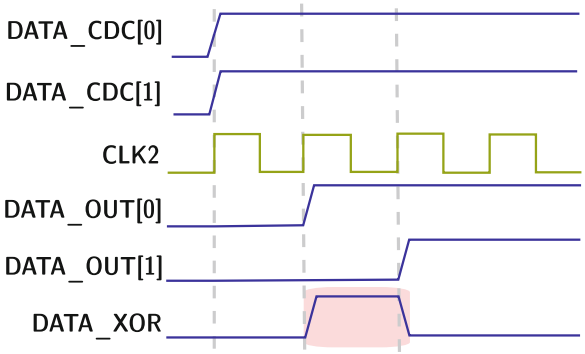


Fig. 4. Bus incoherency behavior

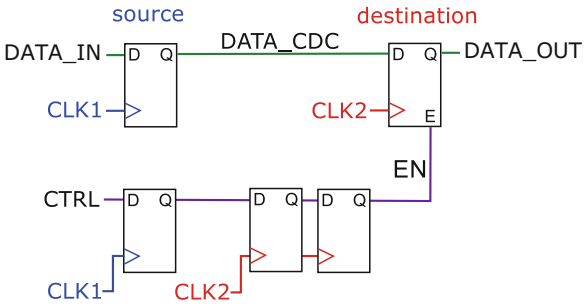


Fig. 5. Enable-based synchronization (Color figure online)

An alternative solution is using a control signal connected to the enable pin of the destination registers (Fig. 5). This *CTRL* signal is set to '1' only after *DATA_IN* stabilizes. Hence, no metastability can be propagated in the destination domain, and there is no need for further resynchronization [10]. Of course, this *'stable'* information comes from the source clock domain, so this control signal must be resynchronized in the destination domain (here with a multi-flop). Note that different synchronization schemes are derived from this structure. The control signal is here connected to the enable pin of the flop (the selection of a recirculation mux), but it could also be connected to a clock-gate enable, or even an AND gate on the data path.

2.3 Data Loss and Handshake

The enable-based synchronizer structure only propagates stable data. However, if the source register keeps sending data, the destination might wait for their stability and lose some packets. The source should then wait for the data to be captured before sending a new one. This can be done with a handshake protocol using request/acknowledge signals, as shown on Fig. 6. (In Figs. 5, 6 and 7, clock

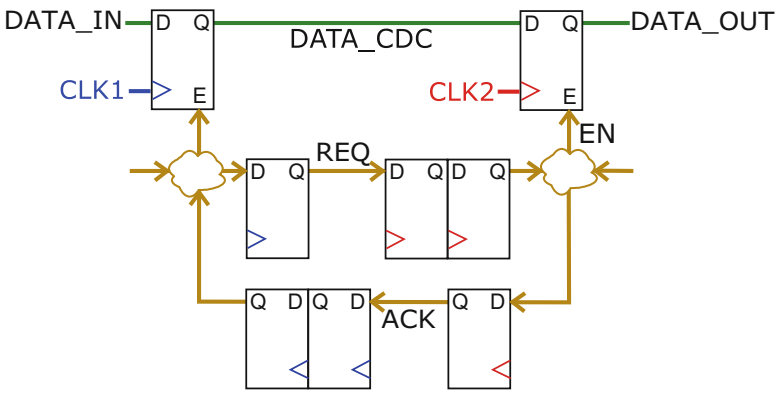


Fig. 6. Handshake synchronization (Color figure online)

domains are shown in blue or red, data is in green and control logic is in yellow or purple.)

2.4 Performance with FIFO

The delay introduced by handshake protocols may not be acceptable for a high-rate interface. Putting a FIFO in the CDC allows the source to write and the destination to read at their own frequencies, and increases the data propagation efficiency. In a FIFO, all the previous schemes are implemented (see Fig. 7). The main controls of the CDC are the write and read pointers, which need to be Gray-encoded before being synchronized by a multi-flop. In order to activate

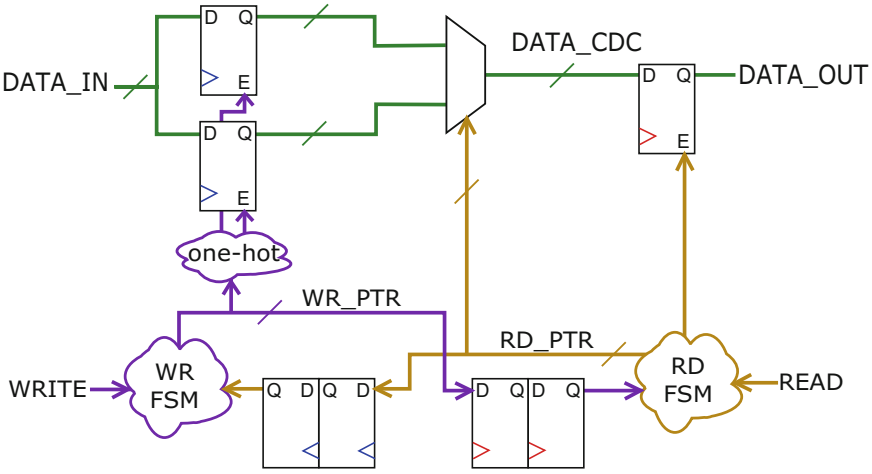


Fig. 7. FIFO synchronization (Color figure online)

the source or destination access, global write and read control signals can be implemented in a handshake protocol.

Using a FIFO implies some data latency (caused by the handshake and the resynchronization of pointers), but allows a higher transfer rate. All the previously mentioned issues are avoided (metastability, coherency, data loss), but its complexity makes the FIFO the most difficult synchronizer to design and verify.

3 Current Verification Approach

While some hardware bugs can sometimes be resolved by the firmware or software layers, incorrect synchronizers typically lead to non-correctable, so-called *chip-killer* bugs. To guarantee the absence of CDC issues in a design, a methodology is needed to check that all the necessary synchronizers are implemented, and that their protocol is followed (Fig. 8).

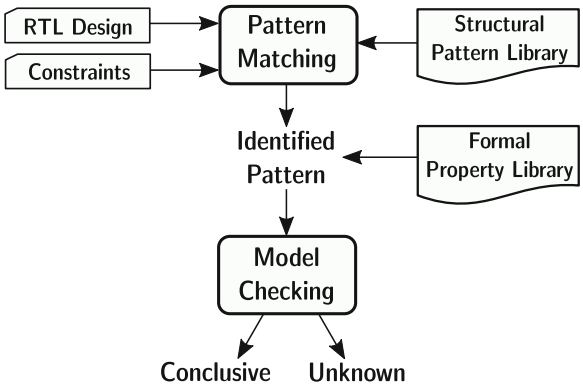


Fig. 8. CDC formal verification methodology

3.1 Structural Checks

After register-transfer-level (RTL) synthesis, on a net-list, it seems easy to structurally detect a CDC between two registers, and even to detect a multi-flop. In contrast, for complex FIFO protocols, identifying the correct control logic is non-trivial. If a single synchronizer structure was used, a proper pattern matching could try to identify it on each CDC. Unfortunately, in industry, many designers create their own synchronizing structures, and the structural library used for pattern matching would never be exhaustive on complex structures such as FIFOs.

In order to provide a robust and automated analysis, state-of-the-art CDC tools provide a more flexible approach which identifies more general patterns (like the one based on enables), without relying on the rigid FIFO or Handshake structures. This is a first quick step to check multi-flops and sort out missing synchronizers. However, a structural approach cannot check protocols and assumptions on the control signal. A functional check must then be run.

3.2 Functional Checks

From the recognition of a synchronizer structure, the extracted information is reused to run functional checks. Formal safety properties to be checked are associated to the generic structures we are using, among which:

1. **Stability**
   The destination register only propagates `DATA_CDC` *when it is stable.*
2. **Coherency** (*Gray-encoding check*)
   At most one bit at a time can change in `DATA`.

As an aid to the user, these formal properties are embedded in the CDC tools and linked to the matched patterns. When a pattern is detected, the properties are automatically synthesized in hardware, mapped to the corresponding RTL signals and formally checked.

3.3 Limitation

After running the structural and functional checks of Fig. 8, the user expects to know which data is correctly resynchronized and which is not. However, experience shows that model checking may not achieve a conclusive result on the properties: some of them reach a timeout even after several days. When this occurs, no information is returned on the cause of the timeout, and the designer is left with no clue on the possible presence of a metastability in the design.

It is well-known that inconclusive results in formal verification are caused by the so-called *state-space explosion problem* which is intrinsic to model checking of hardware designs. In practice, the typical approach to overcome this challenge is, for each property, to extract the CDC logic. The verification is then focused on just a small but relevant part of the design. However, this approach comes with the following issues: First, the verification engineer needs to have a very good understanding of the underlying design, which is not realistic for large RTL models; Then, strong time-to-market constraints do not allow a manual labor-intensive selection of appropriate abstractions for each property; Finally, even with such a high manual effort, a conclusive result cannot be guaranteed.

The approach presented in this paper is also based on focusing on a subpart of the design, but tries to overcome the aforementioned issues by following a CDC-oriented methodology that is based on an interaction between the user and a refinement algorithm.

3.4 Root Causes of Inconclusive Results

In this subsection, we report on common root causes of inconclusive results we observed in the verification of CDCs.

Operation Modes. An SoC can operate in many different modes (initialization, mission mode, test, scan, etc.) controlled by configuration signals, the values of which cannot be automatically inferred by the verification tool. The user must then provide functional design constraints such as clock frequencies, static value of configuration signals, etc. to perform the verification on a realistic mode. This method is user time consuming and error prone, as the user may fail to provide some essential signal constraint.

Clock Gating. In complex low-power designs, some modules can be enabled or disabled via a clock-gate for power saving. If the clock enable signals take inconsistent values, the tool produces unrealistic failures by exercising unreachable states of the design. The user should provide constraints on the value of the clock enable signals.

Protocols. In addition to design setup, functional assumptions should be given on the primary inputs of the design, e.g., for handshake protocols.

Considering all the above, in all practical cases we encountered, inconclusiveness was primarily caused by missing constraints. But even with all this information – that is not always trivial to write – a model checker may still not reach a conclusive result, due to the design complexity. We need a new approach both to tackle this complexity issue and to identify missing constraints.

4 User-Aided Abstraction Refinement

The objective of our approach is to avoid the state-space explosion problem in model checking hardware designs. To that end, our key idea is to let the user *aid* the model checking process by replying yes/no to a series of questions whose answers only require local design knowledge.

Technically, our underlying framework is a *counter-example-guided abstraction refinement* (CEGAR) [8] algorithm: we maintain a sequence of abstractions with increasing precision until a definite result can be established. In contrast to fully automatic CEGAR approaches, the user here influences the refinement process. We therefore call our approach *user-aided abstraction refinement* (UsAAR). Figure 9 gives an overview on the semi-automatic algorithm in the context of the overall methodology.

4.1 Localization Abstractions

Our abstractions are obtained via *localization reduction* [18]: we replace some nets in the original design with primary inputs, called *cut points*. An abstraction A is more precise than an abstraction B if the cone-of-influence (with cut-points) of the property in A is an extension of the one in B.

The rationale for this notion of abstraction is that, in practice, all the relevant control logic for a given CDC is implemented *locally*. Thus, properties requiring

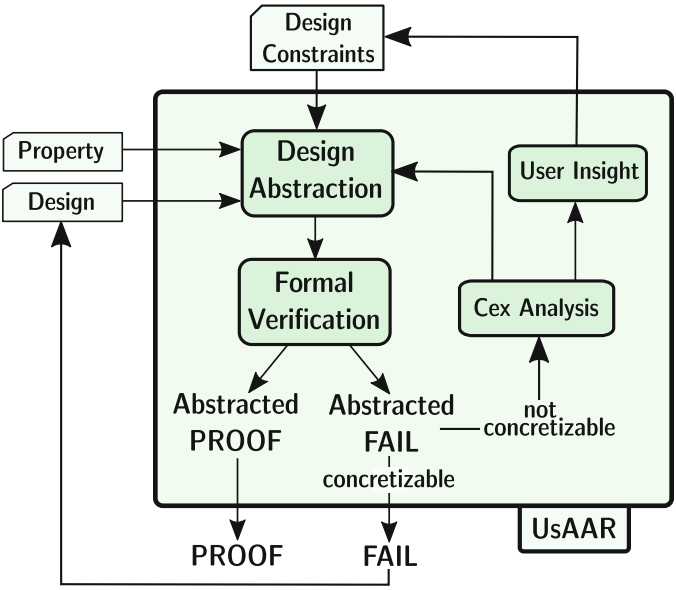


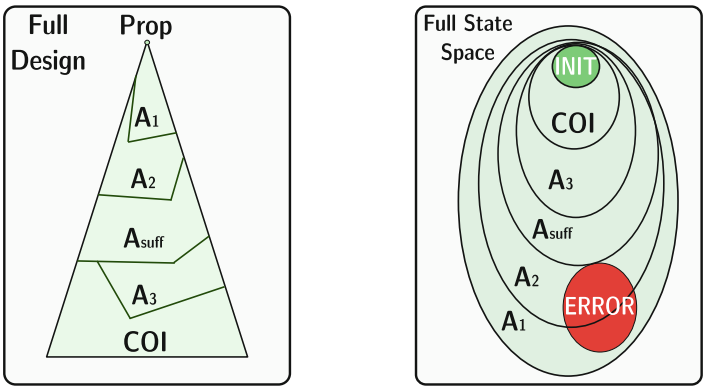
Fig. 9. The UsAAR algorithm of our CDC verification methodology

the correctness of synchronizing protocols should have small abstractions that suffice to either prove the property or to reveal bugs.

Figure 10 illustrates the abstraction process for a correct, hard to prove, property. By removing parts of the circuit from the cone-of-influence of the property (keeping only A_1 from COI), and leaving the unconnected nets free, the set of reachable states is enlarged (i.e., it represents an over-approximation). As a result, states in which the property fails (the error states), initially unreachable, may become reachable (A_1). In this context, refining the abstraction consists in iteratively adding back some of the removed circuit, and as a consequence reducing the reachable state space (from A_1 to A_2), until a sufficiently precise abstraction is obtained ($\mathsf{A}_{\mathsf{suff}}$), for which no error state is reachable. The challenge here is to find that part of the design that can be pruned away without spuriously making any error state reachable.

4.2 The Core Algorithm

The algorithmic core of our methodology (Algorithm 1, in pseudo code) is a semi-automatic algorithm which is based on the automatic *counterexample-guided abstraction refinement* (CEGAR) [8] principle. A localization abstraction of the design is incrementally made more precise in a sequence of *refinement rounds*. In each round, the safety property is checked on the abstraction: if the property is satisfied, the algorithm terminates with Result "proof"; if a counterexample is found, a refinement heuristic decides whether and how the abstraction should be refined, or it concludes that the counterexample is *concretizable*, i.e., the



(a) Localization abstractions with different degrees of precision

(b) Corresponding sets of reachable states

Fig. 10. Various abstractions for a given design and property

counterexample is also valid for the full design, and the algorithm can terminate with Result "fail".

Throughout the algorithm, we maintain a set of constraints C_{global} and a set of nets F. We call F the *focus*: it induces a localization abstraction $D^{\#}$ of design D (Line 5). The constraints C_{global} are used when property P is checked on $D^{\#}$ (Line 6). Starting with no constraints (Line 2) and F just holding the nets in the combinational fan-in of P (Line 3), we incrementally add elements to both sets, thereby making the over-approximation more precise. Based on the abstract result $R^{\#}$ obtained in Line 6, we either immediately terminate (in case $R^{\#} = \textbf{proof}$), or continue analyzing the abstract counterexample $cex^{\#}$ (in case $R^{\#} = \textbf{fail}$). The next subsection details this analysis.

4.3 Analysis of Abstract Counterexamples

In our variant of CEGAR, the refinement heuristic first determines a set of cut points that are logically relevant for the abstract counterexample. This is done by computing (an over-approximation of) the *justifiable set* of cut points J (Line 11). The validity of the counterexample is independent of any cut point that is not part of J. Intuitively, any net that is not contained in a minimal justifiable set can be set to a random value without invalidating the reachability of the error state. But since computing a minimal justifiable set is a hard problem on its own, heuristics are used to compute a small but not necessarily minimal set. A common technique is to use ternary simulation to identify inputs that do not impact the overall validity of the counterexample.

Once J is obtained, the heuristic Analyze classifies each element in J into specific categories (Line 12): clock, reset, data, control, etc., using a backward

Algorithm 1. UsAAR for a design D and a property P

```
1: R ← unknown
2: C_global ← ∅
3: F ← CombFanin(P)
4: while R = unknown do
5:    D# ← Abstract(D, F)
6:    (R#, cex#) ← Check(P, D#, C_global)
7:    if R# = proof then
8:       // Terminate and report proof
9:       R ← proof
10:   else if R# = fail then
11:      J ← Justify(cex#, D#) \ F
12:      (C_prop, ref) ← Analyze(J, D)
13:      (C_acc, ref') ← Review(C_prop)    // User interaction
14:      ref ← ref ∪ ref'
15:      if C_acc = ref = ∅ then
16:         // Terminate and report fail
17:         R ← fail
18:      else
19:         // Refine the abstraction and continue
20:         C_global ← C_global ∪ C_acc
21:         F ← F ∪ ref ∪ Nets(C_acc)
22:      end if
23:   end if
24: end while
25: return R
```

traversal of the RTL which starts at the synchronizer pattern. Then, realistic constraints corresponding to each category are inferred. For instance, when encountering a potential setup issue such as a missing clock-gating constraint, Analyze proposes to assume that the control input of the clock-gate is always set to an enabling value. Or if a net is found to be logically irrelevant for the user, Analyze infers a stopper constraint to ensure that the net (and its fan-in) will not be part of any future abstraction. In the asynchronous FIFO of Fig. 7, this stopper constraint would be applied on the net *DATA_IN*. Indeed, in this case, the property is independent of the *DATA_IN* value. Only the following control logic is relevant.

All constraints C_{prop} inferred by Analyze are then reported for review (Line 13). In case the user rejects a constraint, the corresponding net is marked for automatic refinement. After the manual classification process, the accepted constraints C_{acc} are added to the set of global constraints C_{global} (Line 20). For all nets that are marked for automatic refinement *ref*, we extend the focus so that the subsequent abstraction are more precise by additionally comprising those nets (Line 21).

4.4 Soundness, Completeness, Validity

The algorithm terminates if either the model checker reports a *proof* or if no new constraints or nets for automatic refinement can be inferred, in which case a *fail* is reported (Line 15). The soundness of reported *proofs* follows straight forward from the fact that our localization abstraction represents an over-approximation. The soundness of reported *fails* follows from the definition of the justifiable set: the abstract counterexample $cex^{\#}$ only depends on nets within the focus, i.e., on nets that were not abstracted out or for which the user provided constraints. Hence, $cex^{\#}$ remains a valid counterexample for any greater set $F' \supset F$, and in particular, on the full design D.

In every non-terminating round, we either monotonically make the abstraction more precise or constrain the design behavior. Hence, since the underlying design is finite, the algorithm terminates. Completeness follows from the fact that the algorithm either terminates with a sufficient abstraction, or it ultimately reaches the full design, i.e., $D^{\#} = D$.

When manually adding constraints one runs the risk of over-constraining the design's behavior, which can lead to vacuous proofs. However, our UsAAR methodology is designed to minimize the risk of over-constraining. The setup constraints inferred by our heuristics are combinational and structurally close to the CDC control logic, which makes them easy for the user to review. Then, they do not over-constrain but ensure that the design does not exhibit spurious behavior. On the other hand, stopper constraints (i.e., static cut points) are conservative: they lead to an over-approximation that preserves all safety properties.

5 Case Study

We applied our new methodology on two hardware designs: a small parametric FIFO and a complex SoC from STMicroelectronics. The first one reveals the benefits of the different steps of the flow. The second one proves the validity of the methodology on an SoC from industry.

5.1 Asynchronous FIFO

Design Presentation. This hardware design includes a FIFO similar to the one presented in Fig. 7. To mimic a state-space explosion on the *DATA_IN* and *WRITE* paths of the source domain, an FSM was implemented with a self-looping counter on 128 bits, along with some non-deterministic control logic. Also, the source and destination clocks are enabled by sequential clock-gates, controlled by two independent primary inputs.

This design is parameterized by the width of the data being propagated, and by the depth of the FIFO. By varying these two size parameters, we increase the design complexity and analyze the corresponding performance of the methodology.

Results. Using an industrial tool to structurally analyze the design, three formal properties were extracted.

- A data stability property is created on signal DATA_CDC.
- Two coherency properties are extracted on the address buses after synchronization, one on RD_PTR and one on WR_PTR. Indeed, the write and read pointers are synchronized with multi-flops, and should then follow Gray-encoding (see Sect. 2.2).

To verify them, the open source model checker ABC [5] is used with the engines *PDR* [11] and *BMC3* [4] in parallel. For each property, the runtime limit for timeout is set to 15 min (denoted *T/O* in Table 1). We run the experiments on a workstation with 24 Intel Xeon 2.6 GHz CPUs and 220 GB of memory. Four different schemes are applied to generate the results in Table 1:

1. **Standard:** Model-checking each property on the full (non-abstracted) design.
2. **CEGAR:** A UsAAR variant where we reject all constraints. It can be seen as a reduction of UsAAR to standard CEGAR.
3. **UsAAR:** The full semi-automatic algorithm presented in Sect. 4 including automatic refinement and constraint inference together with manual constraint classification.
4. **Standard w/ constraints:** Repeated run of the standard scheme with all the accepted constraints from the UsAAR scheme.

A first observation is that the coherency properties are proved in less than a second in all four schemes and variations of the design. This is not surprising considering that the Gray-encoding implemented in this design does not depend on any non-deterministic control logic. Henceforth, we will then focus on the data stability property.

The standard scheme is not able to prove the property in all 35 variations of the design (Column "Standard"). Using the simple CEGAR approach, the property is proved in all variations within 4 to 15 min (Column "CEGAR"). Interestingly, the proof runtime is stable when the FIFO depth is fixed and the data width increases. By looking at the last abstraction exercised, we notice that *DATA_IN* is always abstracted out. Its value does not depend on the source logic. Hence, heuristics from the proof engine inferred that the proof does not depend on the data value, which make the analysis as simple for 8 bits as it is for 128 bits. Actually, even if the source logic of the data was greatly more complex, the CEGAR result would be the same.

Along the UsAAR run, two static constraints are automatically inferred on the enables of the clock-gates. Because having a non-deterministically enabled clock is not a realistic design behavior, we decide to accept them. As a result, the stability property is solved in all 35 variations of the design within 10 s each (Column "UsAAR"). Same as with simple CEGAR and contrary to the standard scheme, the complexity of the data source logic is irrelevant for the proof.

Interestingly, even when applying the inferred enabling constraints on the standard scheme, not all properties can be solved (Column "Standard w/ constraints"). Also in this case, by comparing with column "UsAAR", we notice

Table 1. CDC properties proof CPU runtime (in sec) on the asynchronous FIFO

FIFO depth	Data width	Standard	CEGAR	UsAAR	Standard w/ constraints
3	8	T/O	389	7	22
	16	T/O	390	7	35
	32	T/O	392	7	66
	64	T/O	390	7	870
	128	T/O	391	7	T/O
4	8	T/O	592	6	15
	16	T/O	591	6	28
	32	T/O	594	6	57
	64	T/O	593	6	145
	128	T/O	594	6	243
5	8	T/O	641	7	14
	16	T/O	651	7	53
	32	T/O	641	7	69
	64	T/O	640	7	180
	128	T/O	693	7	374
6	8	T/O	558	7	13
	16	T/O	558	7	55
	32	T/O	563	7	62
	64	T/O	563	7	203
	128	T/O	562	7	414
7	8	T/O	574	7	10
	16	T/O	574	7	49
	32	T/O	575	7	68
	64	T/O	574	7	150
	128	T/O	575	6	841
8	8	T/O	589	7	11
	16	T/O	590	7	36
	32	T/O	579	7	60
	64	T/O	580	7	150
	128	T/O	580	7	463
9	8	T/O	868	9	14
	16	T/O	863	9	43
	32	T/O	868	9	74
	64	T/O	864	9	210
	128	T/O	865	9	475
TOTAL PROVED		0	35	35	34

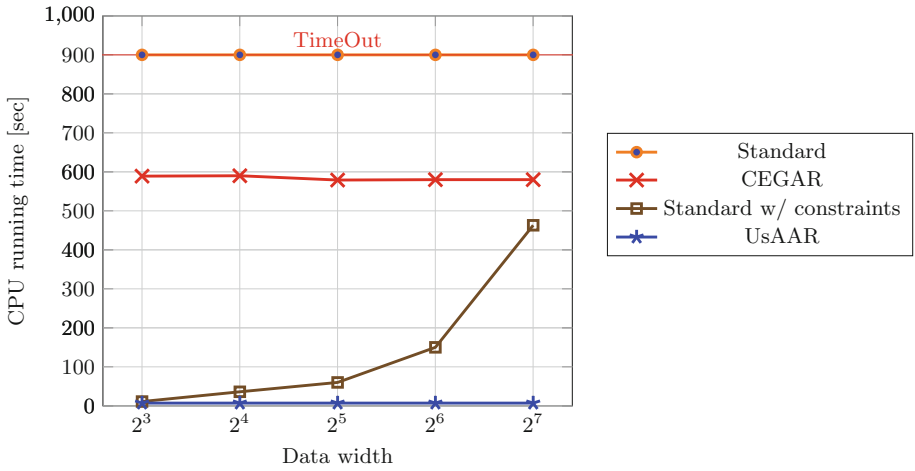


Fig. 11. Performance comparison for FIFO depth 8

that the runtime is always higher than when using both the inferred constraints and CEGAR. This observation along with Fig. 11 points out the importance of using both CEGAR and constraints in order to reach a conclusive result.

5.2 CPU Subsystem

Design Presentation. The second case study is a complex SoC hardware design from STMicroelectronics, intended for a gaming system. It is a low-power architecture, with a state-of-the-art quad-core CPU and many different interfaces. In total, it holds over 300,000 registers and 7 million gates. The CDC setup is mainly done in a clock and reset control module, which selects configurations for the whole system among its 38 clock domains and 17 primary resets. However, many configuration signals (such as clock-enable signals) are not controlled by this module. Since the design has a Globally Asynchronous Locally Synchronous (GALS) intent, CDC signals are always synchronized in the destination module.

Figure 12 gives an overview of some synchronizations around the CPU. Data communication with the CPU environment (the rest of the SoC) is synchronized by a customized FIFO with a 4-phase protocol based on the one described in Sect. 2.4, with additional low power and performance optimizations. Only one communication is shown in Fig. 12, among the ten in each direction. The figure also shows the communications with the clock and reset controller, and the handshake with the low power management block. Note that the CPU is one central module which, due to its complexity, is likely to cause a timeout in the model checking algorithm when considered in its entirety.

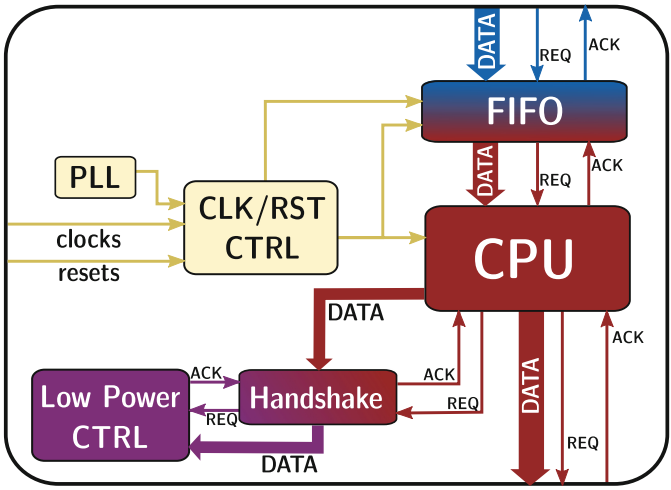


Fig. 12. Overview on the synchronizers at the interface of the CPU

Many synchronizers (mostly FIFOs) are split between modules of this subsystem. Hence, we cannot proceed in a module-by-module CDC analysis. Working at this hierarchy level is particularly relevant for us.

Results. We used an industrial tool to structurally analyze the STMicroelectronics design. Only some straightforward constraints regarding reset and clock setup were applied; we did not use any other design insight. All clock multiplexers were constrained to select the mission-mode clock, and static primary inputs were constrained to the value given in the design specification (subsystem configuration). The structural analysis identified several thousand synchronizers, most of them multi-flops which do not need a functional check. It also extracted 78 stability and 47 coherency properties. We verify each extracted property in the same four schemes that were presented previously.

Table 2 shows the results for model checking the stability and Gray-encoding properties. Without any automatic refinement, the standard scheme can only prove 40 out of 125 properties (Column "Standard"). After increasing the timeout limit to several hours, the same results are obtained. Using automatic refinement (the CEGAR scheme), 33 more properties can be proved (Column "CEGAR"). Also, it should be noted that all proofs from the standard scheme get confirmed by the CEGAR scheme. CEGAR proves to be particularly efficient for proving Gray-encoding properties, as the encoding logic is generally local to the synchronizer.

The most striking observation, however, is that during the first UsAAR run, 40 setup constraints are automatically inferred and are all easily accepted. These include global interface enables (scan or test enables, internal configuration signals, ...), and also internal soft resets and clock-gate enables which were missing

Table 2. Results on the CDC stability and Gray-encoding properties

		Standard	CEGAR	UsAAR first run	**UsAAR second run**	Standard w/ constraints
Stability	# Proof	29	31	45	**78**	43
	# Fail	0	0	33	**0**	0
	# Inconclusive	49	47	0	**0**	35
	CPU time [min]	771	734	583	**31**	557
Gray-enc.	# Proof	11	42	42	**47**	11
	# Fail	0	0	5	**0**	0
	# Inconclusive	36	5	0	**0**	36
	CPU time [min]	540	86	27	**15**	540

in the design specification. It leads to 87 proved properties and provides counterexamples for the remaining 38. Note that in those abstract counterexamples, many irrelevant signals are automatically hidden using the justifiable subset, which makes debugging easier.

By reviewing them, we observe spurious behaviors in the handshakes, which are fixed by adding 22 missing Boolean assumptions enabling the protocols. Indeed, in some cases the *WRITE* or *READ* of the FIFO represents an information coming from the CPU, and would depend on a software execution. When these signals are abstracted out, they take random values which do not follow the handshake protocol, hence creating a failure. After consulting STMicroelectronics, we decide to constrain them to a realistic behavior. Here, the worst case would be to set them to '1' which would mean the CPU always transfers data. We stress the fact that no deep design knowledge is needed during this process, and the constraints represent a realistic design behavior.

With these new constraints, the second UsAAR run is able to conclude all 125 properties correctly. Compared to the fully automatic approaches, the final UsAAR proof runtime is accelerated by more than 20×. In fact, the most difficult property concludes in only 7 min.

Finally, the last column shows that having the proper constraints is not sufficient to get proofs; the efficiency of UsAAR indeed relies on the *combination* of automatic CEGAR and manual constraint classification.

Regarding the size of the abstractions: on the full design, some properties have a cone-of-influence of more than 250,000 registers. Interestingly, our variant of CEGAR is able to find sufficient abstractions containing only up to 200 registers. This ratio confirms our assumption that only the local control logic has a real influence on the correctness of a CDC property.

Overall, a relevant metric to score the different flows would be the total time spent by the verification engineer starting with the design setup and ending with achieving conclusive results for all properties. It would allow us to conclude on the complexity and usability of different methodologies, as for instance the manual extraction and constraining explained in Sect. 3.3. However, this time depends very much on the design complexity, reuse, and user insight. Such an

experiment would assume the availability of two concurrent verification teams on the same design, an investment that could not be made by our industrial partners.

6 Related Work

The implementation of CDC synchronizers recalled in Sect. 2 is well known in the hardware design community. Tools for verifying such synchronizers are provided by leading EDA vendors (Synopsys SpyGlass CDC [27], Mentor Questa CDC [23], Real Intent Meridian CDC [24], ...). Most of these tools provide a verification flow including structural checks up to the generation of related formal properties.

Academia is also active in this research area. Some approaches focus on functionally verifying CDC synchronizers; e.g., Burns et al. proposed a new verification flow using xMAS models [6]. However, the user needs to define the boundaries of the synchronizers, which is not scalable.

Kwok et al. presented a verification flow [19] purely based on a structural analysis that matches parts of the design with a property library to generate assertions. These assertions can be model checked for functional verification. Litterick proposed a similar flow [22], replacing model checking by simulation on SVA assertions. Kapschitz and Ginosar published an overview [15] on the general CDC verification flow, showing the need for multiple clock modeling and formal verification. However, they did not detail how synchronizers can be identified, nor their flow automation, nor how to deal with a high design complexity.

Li and Kwok described a CDC verification flow [21] similar to ours, including the extraction of a formal property from an automatic structural identification. They performed abstraction refinement along with synthesis to prove some properties, but the underlying techniques were not explained in detail. In their flow, inconclusive properties after the abstraction refinement are *promoted* to the top-level and the user needs another methodology to proceed with the formal verification.

Recently, Kebaili proposed to improve the structural checks in order to detect the main control signals of the synchronizer [17]. The properties to be verified would only rely on these control signals (with a handshake-based protocol), hence avoiding the state-space explosion in the data path.

In other hardware verification domains, some methodologies combine manual with automatic reasoning. For instance, for verifying the FlexRay physical layer protocol, Schmaltz presented a semi-automatic correctness proof [26] in which the proof obligations are discharged using Isabelle/HOL and the NuSMV model checker. This proof was also applied to larger verified system architectures [1].

Localization abstractions and related refinement techniques were pioneered by Kurshan in the 1980s and eventually published in the mid 1990s [18]. The fully automatic variant of the CEGAR principle was introduced by Clarke et al. in the context of over-approximating abstractions defined through state-space partitionings [8,9]. The works by Andraus et al. propose a CEGAR approach for

data-paths in hardware designs [2,3]. Orthogonal to our approach, their abstractions are obtained by replacing data-path components by uninterpreted functions which, in turn, also requires a more powerful model checker based on SMT. Our methodology can be seen as an extension of these works mentioned above, as it enables the integration of user insight into the refinement process.

7 Conclusion and Outlook

This paper presents a complete formal verification flow for conclusively proving or disproving CDC synchronizations on industrial-scale SoC hardware designs. Our core contribution is a semi-automatic model checking algorithm, where the user aids the (otherwise fully automatic) verification process by classifying a sequence of automatically inferred constraints.

We demonstrated the efficiency of our approach on an STMicroelectronics SoC design which was persistently difficult to verify: prior approaches required to manually extract the cone-of-influence of the synchronizers, which resulted in a tedious (and costly) work for verification engineers.

In contrast, our new methodology allowed the full verification without requiring any deep design knowledge. This very encouraging practical experience suggests that we identified an interesting sweet-spot between automatic and deductive verification of hardware designs. On the one hand, it is a rather easy manual task to classify simple design constraints that refer to single nets where, on the other hand, this information can be crucial to guide an otherwise automatic abstraction refinement process.

Another positive side-effect of our methodology is that it gradually results in a functional design setup. Note that all accepted constraints (except the stopper constraints) do not depend on a certain property, but reflect general design properties and are therefore globally valid. This does not only speed-up the overall CDC verification time, when constraints are reused while verifying multiple properties, it also helps further functional verification steps in the VLSI flow. For instance, the same design constraints can be reused for functionally verifying false and multi-cycle paths.

As a next step, we plan to improve the constraint inference in order to generate sequential SVA assumptions. This feature would guide the user into creating more complex constraints representing realistic design behaviors without decreasing the proof coverage. Also, we investigate into other functional properties. The long-term goal is to extend our methodology to many critical functional verification steps in the VLSI flow.

Acknowledgement. We wish to thank Mejid Kebaïli and Jean-Christophe Brignone from STMicroelectronics for reviewing and confirming the validity of our methodology.

References

1. Alkassar, E., Böhm, P., Knapp, S.: Formal correctness of an automotive bus controller implementation at gate-level. In: Kleinjohann, B., Wolf, W., Kleinjohann, L. (eds.) DIPES 2008. ITIFIP, vol. 271, pp. 57–67. Springer, Boston, MA (2008). doi:10.1007/978-0-387-09661-2_6
2. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Refinement strategies for verification methods based on datapath abstraction. In: ASP-DAC, pp. 19–24 (2006)
3. Andraus, Z.S., Sakallah, K.A.: Automatic abstraction and verification of Verilog models. In: Design Automation Conference (DAC), pp. 218–223 (2004)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). doi:10.1007/3-540-49059-0_14
5. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14295-6_5
6. Burns, F., Sokolov, D., Yakovlev, A.: GALS synthesis and verification for xMAS models. In: DATE (2015)
7. Chaney, T., Molnar, C.: Anomalous behavior of synchronizer and arbiter circuits. IEEE Trans. Comput. **C–22**(4), 421–422 (1973)
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:10.1007/10722167_15
9. Clarke, E., Grumberg, O., Long, D.E.: Model checking and abstraction. In: ACM (1991)
10. Cummings, C.E.: Clock domain crossing design & verification techniques using systemverilog. In: SNUG, Boston (2008)
11. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134 (2011)
12. Gabara, T.J., Cyr, G.J., Stroud, C.E.: Metastability of CMOS master/slave flip-flops. In: IEEE Custom Integrated Circuits Conference. pp. 29.4/1–29.4/6, May 1991
13. Ginosar, R.: Fourteen ways to fool your synchronizer. In: Asynchronous Circuits and Systems, pp. 89–96 (2003)
14. Ginosar, R.: Metastability and synchronizers: a tutorial. IEEE Des. Test Comput. **28**(5), 23–35 (2011)
15. Kapschitz, T., Ginosar, R.: Formal verification of synchronizers. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 359–362. Springer, Heidelberg (2005). doi:10.1007/11560548_31
16. Karimi, N., Chakrabarty, K.: Detection, diagnosis, and recovery from clock-domain crossing failures in multiclock SoCs. Comput. Aided Des. Integr. Circuits Syst. **32**(9), 1395–1408 (2013)
17. Kebaili, M., Brignone, J.C., Morin-Allory, K.: Clock domain crossing formal verification: a meta-model. In: IEEE International High Level Design Validation and Test Workshop (HLDVT), pp. 136–141, October 2016
18. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
19. Kwok, C., Gupta, V., Ly, T.: Using assertion-based verification to verify clock domain crossing signals. In: Design and Verification Conference, pp. 654–659 (2003)

20. Leong, C., Machado, P., et al.: Built-in clock domain crossing (CDC) test and diagnosis in GALS systems. In: Proceedings of the DDECS 2010, pp. 72–77, April 2010
21. Li, B., Kwok, C.K.: Automatic formal verification of clock domain crossing signals. In: ASP-DAC, pp. 654–659, January 2009
22. Litterick, M.: Pragmatic simulation-based verification of clock domain crossing signals and jitter using SystemVerilog Assertions. In: DVCON (2006)
23. Mentor Graphics: Questa CDC. https://www.mentor.com/products/fv/questa-cdc/. Accessed Jan 2017
24. Real Intent: Meridian CDC. http://www.realintent.com/real-intent-products/meridian-cdc/. Accessed Jan 2017
25. Sarwary, S., Verma, S.: Critical clock-domain-crossing bugs. Electron. Des. Strateg. News **53**, 55–64 (2008)
26. Schmaltz, J.: A formal model of clock domain crossing and automated verification of time-triggered hardware. In: FMCAD. pp. 223–230, November 2007
27. Synopsys: Spyglass CDC. https://www.synopsys.com/verification/static-and-formal-verification.html. Accessed Jan 2017