Conclusive formal verification of clock domain crossing properties

Guillaume Plassan

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITÉ GRENOBLE ALPES

Spécialité : Nano-Electronique et Nano-Technologies (NENT)

Arrêté ministériel : 25 mai 2016

Présentée par

Guillaume PLASSAN

Thèse dirigée par **Dominique BORRIONE, Professeur,**
et codirigée par **Katell MORIN-ALLORY, Maître de conférences,**

préparée au sein du **Laboratoire « Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés (TIMA) »**
dans **l'École Doctorale « Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS) ».**

Conclusive Formal Verification of Clock Domain Crossing Properties

Thèse soutenue publiquement le 28 mars 2018,
devant le jury composé de :

M. Nicolas HALBWACHS
Directeur de recherche, CNRS Alpes, Président
M. Wolfgang KUNZ
Professeur, Université de Kaiserslautern, Rapporteur
M. Jean-Pierre TALPIN
Directeur de recherche, INRIA Rennes, Rapporteur
Mme Dominique BORRIONE
Professeur émérite, Université Grenoble Alpes, Directrice de thèse
Mme Katell MORIN-ALLORY
Maître de conférences, Grenoble INP, Co-directrice de thèse
M. Hans-Jörg PETER
Ingénieur-Docteur, Harman International, Co-encadrant de thèse
M. Florian LETOMBE
Ingénieur-Docteur, Synopsys, Membre

Acknowledgments

This thesis has been a long journey, which could not have been possible without the help and support of many great people.

First and foremost, I owe a great deal to my industrial advisor Hans-Joerg Peter, who consistently proved to be a great colleague, teacher and researcher. I wish to thank very much my directors Dominique Borrione and Katell Morin-Allory, whose time and precious advices made me learn a lot. The University of Grenoble Alpes and Grenoble INP are also acknowledged for giving me the opportunity of this thesis, and for providing the quality education that led to it.

Thank you very much to Jean-Pierre Talpin and Wolfgang Kunz, who accepted to review my manuscript. For accepting to be part of the jury, Nicolas Halbwachs is greatly thanked. I also thank Florian Letombe for doing more than just representing Synopsys in the jury.

For providing such a pleasant and proficient working environment, I thank my colleagues Julien, Laurent, Paul, Emmanuel, Paras, Shaker, Fahim, and particularly Nikos Andrikos and Dmitry Burlyaev. Mejid Kebaili and Jean-Christophe Brignone also deserve credit for initiating me to the CDC topic and participating in a fruitful collaboration.

I thank my dear family for their support throughout my studies and for always cheering me up with good meals.

Last but not least, a thought for my friends, who persistently provided a hearty comfort zone, whether climbing, improvising or sharing good moments. Thank you to Adé (for enduring our engineering talks), Kévin (for resigning from his job to open a bar), Léa (for her cakes), Marc (for his impressive climbing skills) and Pierre (for his persistent DIY spirit). Of course, there is a special thank to my Quand Mêmes: Baptiste (because we are INProx), Bé (for her joy), Candice (for being herself), Catherine (for becoming a QM), Cathy (for her love of order), Diane (for her music taste), Guiouane (because we will lead the world), Ioul (for his quantic spirit), Johan (for being a climbing circus), Julien (for his dog), Ludo (for his inspiration), Micham (for making decisions), Nico (for managing us), Pablo (a.k.a. Mr. Pleasure), Pat (for forcing me to take breaks), Polo (for keeping his coolness), Robin (even though I'm purple now), Steph (for his absurdity).

Contents

CHAPTER 1

Introduction

Our society increasingly relies on digital systems to assist us in our daily life. Some systems improve our quality of life: smartphones provide an easy access to worldwide information, thermostats adjust the room temperature without any human interaction,... Other systems compute complex tasks better than the human mind: planes navigate steadily, modern cars break when facing an immediate danger, factory robotic arms perform repetitive task at a high rate,...

Any engineer would agree that the creation of a system necessarily comes with functional issues, also called bugs. Having a bug in the final product may have tremendous consequences. If multiple smartphones explode in the pockets of random customers, this will lead to a vast product recall, losses in brand recognition, and an unsellable product. In addition to the financial impact, bugs in security-critical systems may lead to leaking private data or even endangering lives. Even if a bug is identified before getting to the customer, it may take weeks to be fixed hence delaying the whole production and leading to potential market losses. Thus, systems require proper techniques to be validated on multiple functionality, safety and security criteria, so that bugs are identified and fixed as early as possible in the design flow.

In particular, most critical systems involve embedding software on hardware. Along its development, software can be tested and patched to fix the bugs, even remotely. Conversely, integrated hardware cannot be modified after its fabrication. For these microelectronic designs, the validation is then required to be performed on a virtual model, before producing the real circuit.

As the society asks for more and more complex tasks to be automated, the complexity of these designs significantly increases. The more complex the system, the more bugs are prone to be introduced, and the harder it is to identify them. To manage the increasing complexity of hardware systems, a specific design and verification flow is processed.

1.1 Hardware Design flow

Modern *systems on chip* (SoC) comprise billions of transistors, which require to be designed with a *very large scale integration* (VLSI) process (see Figure 1.1). Different types of verifications are exercised at each step of the flow, so that bugs are caught and fixed at the earliest stage possible.

The first step in any design flow is to specify the requirements, which describe how the system should behave in its environment. To simplify the problem into sub-problems, the design is divided into submodules. This modularity allows to reuse modules from one

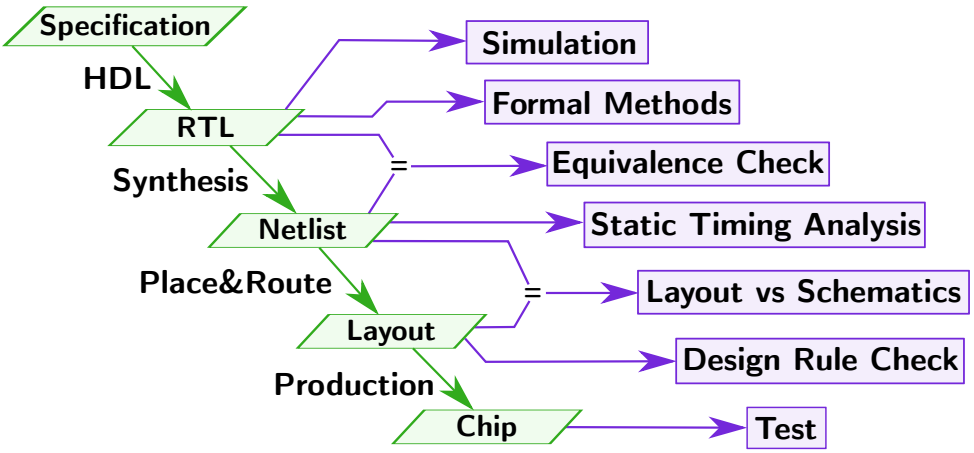


Figure 1.1: Hardware design flow

project to the next, which accelerates the overall development and verification. For instance, a common scheme to design *central processing units* (CPU) is to assemble generic modules from previous projects along with state-of-the-art cores from other companies. Whereas some individual modules have already been verified on their own, their interconnections remain at risk. Indeed, this modular approach requires a lot of interconnections, and since the modules have a different origin, their specifications need to be very precise, which is often not the case.

To formally represent the functionality described by the specification, digital hardware is usually designed using a *Hardware Description Language* (HDL) such as Verilog or VHDL. In HDL, logic gates are designed (AND, OR, NOT) along with memories (D-flip-flop, latch) and arithmetic operations (adder, comparator, ...). This Register Transfer Level (RTL) allows to design large hardware with modularity and flexibility in the architecture. To verify its functionality, simulation stands as the most accepted technique (see Figure 1.13). Simulation virtually exercises the system by feeding patterns of values from a test bench to the inputs of the design, while internal values are checked against a reference. However, this method cannot exhaustively exercise all potential runs of a design, which leads to a hard effort in maximizing the coverage. Thus, formal methods are used to model the design into mathematical equations which fully describe its behavior. Whereas this method guarantees the absence of bugs, it has been observed not to be able to deal with recent complex systems. It is then mostly used on small critical modules whose correctness is required to be guaranteed.

While digital hardware models abstract the physical values as zeros and ones, the final product will be analog. For instance, when an RTL signal toggles, the corresponding physical wire will actually experience a voltage ramp which stabilizes at the intended value (0 being the ground, and 1 being a fixed voltage). Thus, the RTL is *synthesized* into a netlist which only instantiates standard analog components: logic gates and memories. Each component is associated to a technology library with timing constraints related to technology, voltage, and environment factors. The netlist is then formally checked for equivalence against the RTL, which guarantees that the functionality has not been

modified by synthesis. Also, the timing between components is checked using *static timing analysis* (STA). Indeed, a digital design runs on a specific frequency, managed by an oscillating *clock* signal. On the rising edge of this clock, D-flip-flops (also called registers) will memorize the value of their input signals. As they can only capture a 0 or a 1, this input needs to be stable, and not in the middle of a voltage ramp. The propagation delay of a data is evaluated across all combinational gates to guarantee that their signals will be stable on the each rising edge of the clock.

Each component of the technology library is mapped to an analog model, described using layers of materials (mostly silicon-based semiconductors). All these components are then *placed* on a given area, and *routed* (interconnected) with metal layers. As the fabrication of each layer in a clean room is roughly estimated to cost a few millions of dollars, the placement and routing intends to minimize the number of layers. Since this optimization step may introduce bugs, the overall layout is checked against the netlist for functionality issues, with a so-called *layout-versus-schematics* (LVS) check. Also, a *design rule check* (DRC) ensures that these layers of materials are manufacturable with the chosen technology.

The layout is then sent to a *founder* which produces *wafers* of *dice* in clean rooms. Each hardware is tested using input patterns, and rejected if the output values do not fit the expected values. Finally, the hardware is packaged into chips for the consumer to use.

1.2 Clock-Domain Crossing

The complexity of hardware designs has been significantly rising in the last years, as a consequence of more and more ambitious PPA objectives (*Performance, Power, Area*). Indeed, systems are required to maximize performance while decreasing power consumption, using as little silicon area as possible (to lower manufacturing costs).

A simple solution to improve performance is to increase the clock frequency, hence increasing the rate of operations. However, in modern processors, a third of the power is consumed by the clock. Designers then need to find a tradeoff between performance and power saving.

One way is to adapt the frequency of the clock to the ongoing tasks: a fast clock for performance-demanding tasks, and a slow clock for non-critical tasks. Many different clocks are then used in different modules, each one with an optimized frequency. Large hardware designs typically contain tens of these clock domains. As a result, such architectures create many interconnections between the various clock domains, so-called *clock-domain crossings* (CDCs).

A clock-domain crossing (CDC) typically manifests itself as a digital signal path between two sequential elements receiving clocks from out-of-phase domains (Figure 1.2). 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.

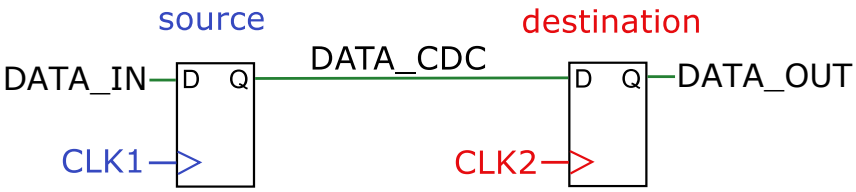


Figure 1.2: A simple CDC

Multiple problems arise from CDCs [25, 75] (metastability, bus incoherency, glitch, data loss, ...), and designers need to implement specific structures at RTL-level to avoid any issue [25, 31] (multi-flop, handshake, Gray-encoding, ...). Consequently, verification engineers must check that all the potential problems have been addressed and corrected. In the following sections, we will provide an overview of the major issues around CDCs, and solutions to overcome them.

1.2.1 Metastability

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 Figure 1.3).

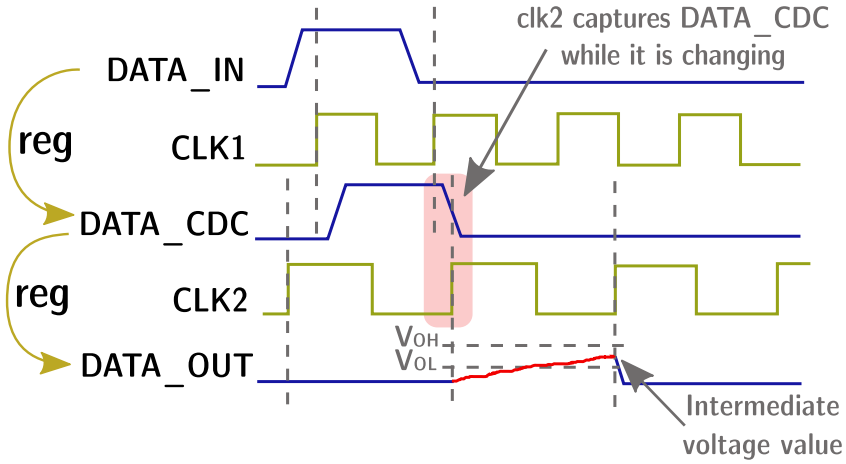


Figure 1.3: Metastability waveform

The metastability phenomenon has been identified a few decades ago [17]: 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 is to introduce a latency in the destination domain, in order to wait for the 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 [29]. 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 [42, 51] 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 [32] (see Figure 1.4). 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.

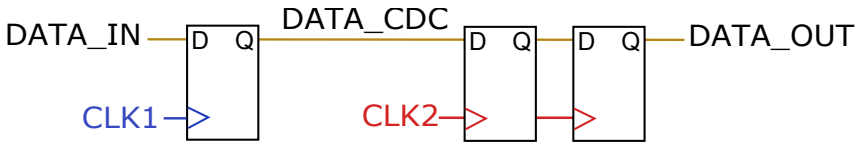


Figure 1.4: Multi-flop synchronizer

1.2.2 Glitch

Whenever two signals converge on a combinational gate, a transient inconsistent value (glitch) may appear. While on a synchronous path, static timing analysis ensures that this glitch is resolved within a clock period, in the context of a CDC, the glitch may be captured and its value propagated.

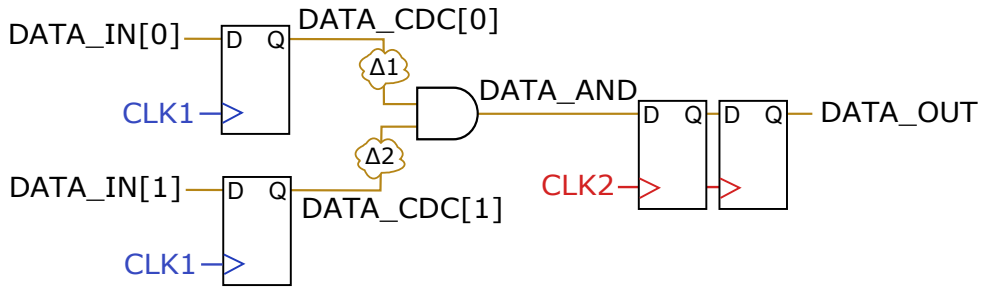


Figure 1.5: CDC convergence

For instance, in Figure 1.5, two signals from the source domain (DATA_CDC[0] and DATA_CDC[1]) converge to an AND-gate. Because of physical constraints, the value from DATA_CDC takes a delay Δ to propagate from the register to the AND-gate. Since this delay cannot be guaranteed to be exactly the same for all converging paths, a glitch may appear on DATA_AND (see Figure 1.6). It is then possible that the destination register captures this glitch, hence propagating an incorrect value.

To guarantee that a glitch cannot appear on this gate, the two input signals should never toggle in the same cycle. In other words, the control logic which generates these signals must ensure that only one signal can toggle at a time.

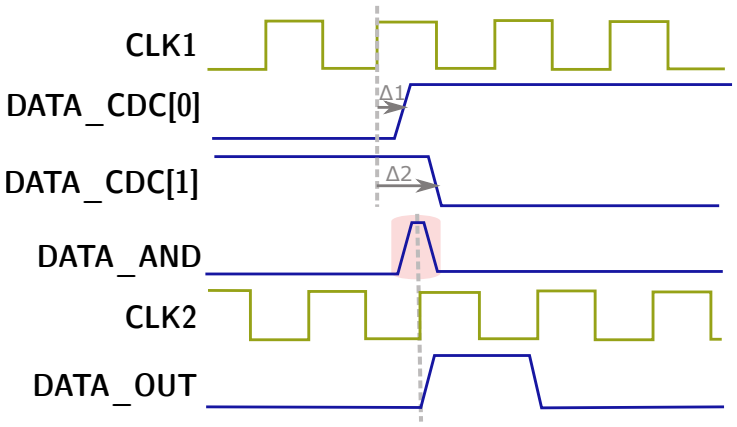


Figure 1.6: Waveform of a captured glitch

1.2.3 Bus Incoherency

When synchronizing buses, there can be *coherency* issues: if some bits of a bus have separate multi-flop synchronizers (see Figure 1.7), 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 (glitch) may even be generated.

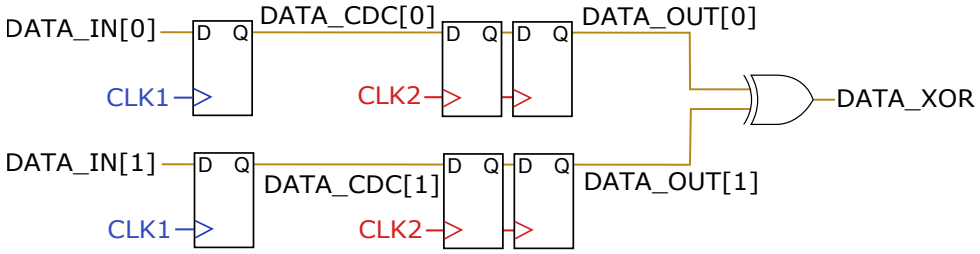


Figure 1.7: Bus synchronization

For instance, in Figure 1.8, 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 [25] (Gray-encoding in the case of a counter, or mutual exclusion in other cases). Even if the multi-flops stabilize this toggling signal with different latencies, the bus output value will be correct regarding either the previous or the next cycle. Thus, no false value is propagated. This can create some data loss, but avoids incoherency.

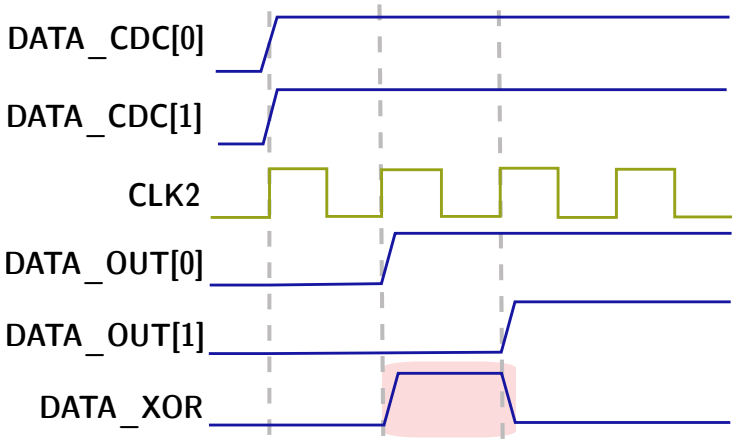


Figure 1.8: Waveform of a bus incoherence with glitch

An alternative solution is using a control signal which stops the data from propagating to the destination registers (Figure 1.9). This *CTRL* signal is set to '1' only after *DATA_IN* stabilizes. Hence, no metastability is propagated in the destination domain, and there is no need for further resynchronization[25]. Also, since the resulting bus is always coherent, a glitch cannot be generated. 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). (In Figures 1.9, 1.10 and 1.11, clock domains are shown in blue or red, data is in green and control logic is in yellow or purple.)

Note that different synchronization schemes are derived from this structure. The control signal is here controlling a recirculation mux of the destination flop, but it could also be connected to a clock-gate enable, an AND-gate or an OR-gate on the data path.

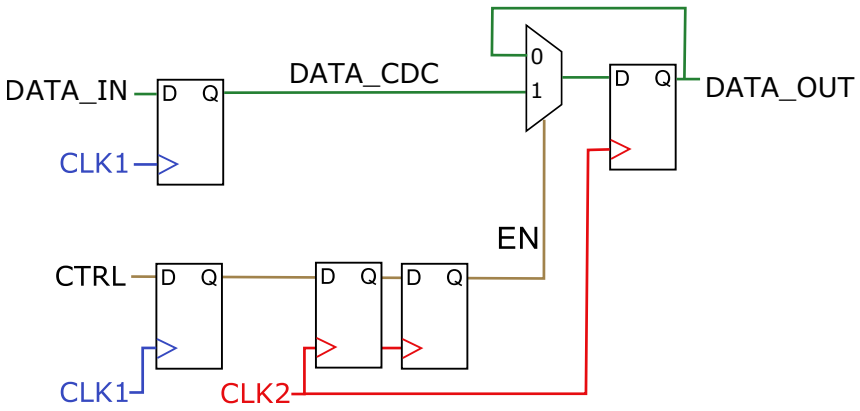


Figure 1.9: Enable-based synchronization

1.2.4 Data Loss

The enable-based synchronizer structure only ensures to propagate 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 Figure 1.10. Using a multi-flop on the control signal implies a cycle of uncertainty, during which the control value is unpredictable. This is not a problem here, as it would only delay a new data to be sent.

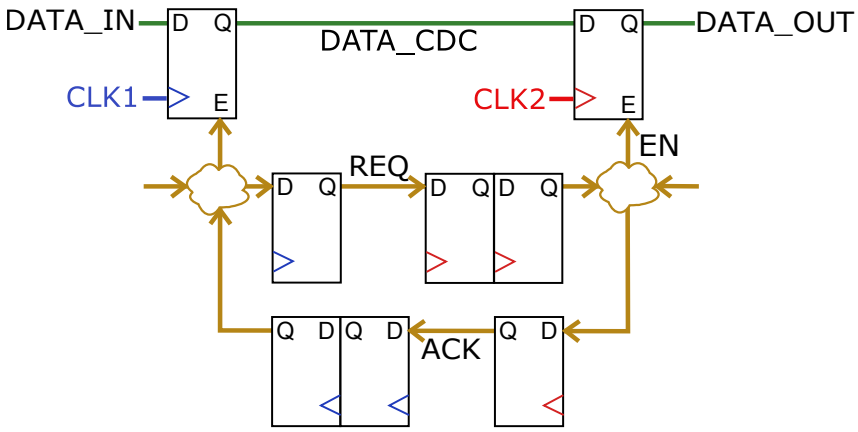


Figure 1.10: Handshake synchronization

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 Figure 1.11). 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 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.

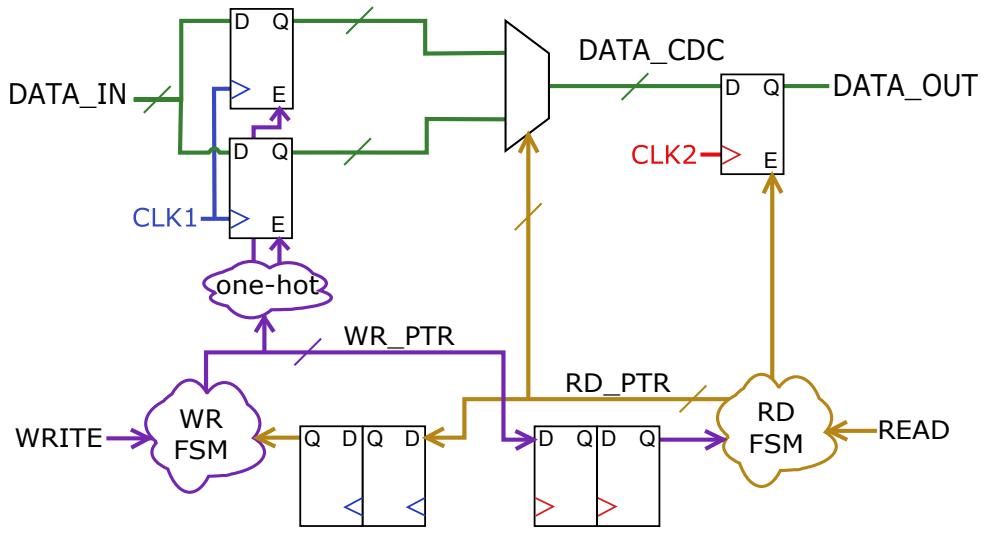


Figure 1.11: FIFO synchronization

1.3 Verification of CDCs

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. Static timing analysis tools do not handle asynchronous paths, hence cannot verify that a metastability would be resolved [19, 55]. To guarantee the absence of CDC issues in an RTL design, leading EDA vendors provide static analysis tools: Synopsys SpyGlass CDC [77], Mentor Questa CDC [60], Real Intent Meridian CDC [73], ... They all propose a similar verification flow (see Figure 1.12). First, a synchronizer pattern is identified on each CDC. Then, the corresponding protocol is checked using formal properties.

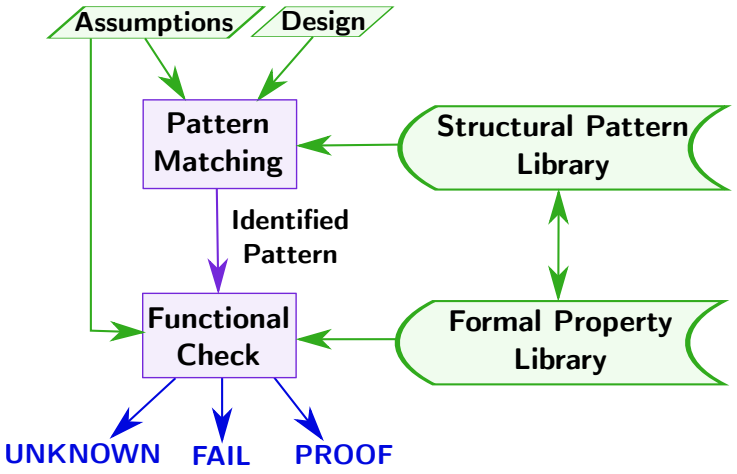


Figure 1.12: CDC verification methodology

1.3.1 Structural Analysis

From a netlist design, it is fairly easy to structurally detect two registers receiving different clocks, 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, it could be stored in a pattern library and identified by a proper pattern matching [43, 49, 50, 54]. 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.

1.3.2 Functional Analysis

Multiple solutions can be considered to functionally check a synchronizer. Recently, Tarawneh proposed a new model for the destination flop of a CDC, which randomly outputs metastability as an X-value [78]. The design is then exercised (either with simulation or formal methods) and the X-value is propagated. The robustness of the design against this metastability is then checked. This approach, however correct, combines a pessimistic random generation of X-value, and an over-approximating ternary execution. Many false violations are then expected to be reported by this methodology.

Burns et al. proposed a novel verification flow using xMAS models, which requires to know the exact boundaries of the synchronizer [15]. These boundaries correspond to the synchronizer patterns, and contain a sufficient part of the design to guarantee the functional correctness of the synchronizer. As previously mentioned, the variety of custom synchronizers avoids automatically detecting the synchronizer boundaries, which makes this approach not scalable.

In practice, most industrial CDC tools reuse structural information to run functional checks. Once a synchronizer structure is recognized, corresponding formal safety properties are generated [41, 43, 49, 54], e.g. in PSL [38] or SVA [39]. For instance, the property corresponding to the bus synchronization of Figure 1.7 verifies its Gray-encoding [50]. Coherency is then ensured since at most one bit can change at a time in `DATA_CDC`. Note that the convergence pattern of Section 1.2.2 is verified using the same property.

Coherency:

```
stable(DATA_CDC) or onehot(DATA_CDC xor prev(DATA_CDC))    @(posedge CLK1)
```

The property corresponding to the enable-based structure of Figure 1.9 checks that the destination register only propagates `DATA_CDC` when it is stable [50]:

Stability:

```
EN -> stable(DATA_CDC)    @(posedge CLK1 || posedge CLK2)
```

Formal properties are generated from all synchronizer patterns, and can then be verified at RTL level using simulation or formal methods (see Figure 1.13). However, the CDC issues are by nature very rare since they depend on physical timing constraints. While simulation scales for large designs, it is only able to show the absence of functional errors in a subset of the full design behavior. A simulation environment, even with good coverage, is then very likely to miss a glitch or metastability. Consequently, in the functional verification of CDC, using model checking prevails.

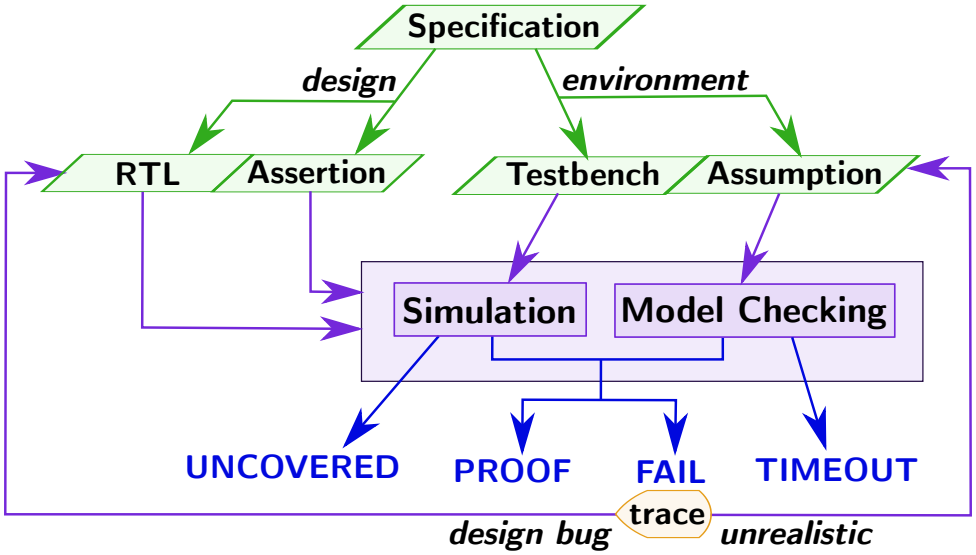


Figure 1.13: Generic RTL functional verification flow

1.4 Thesis Structure

While model checking is precise, it runs into timeouts on complex systems. It then takes the work of an expert verification engineer to create a more precise model of the environment, and to tune the engines in order to obtain a conclusive result: a proof or a failure. Instead of improving the engines, the approach which is taken in this thesis is to simplify the problem. This leads us to tackle multiple issues in the CDC formal verification flow, including:

- Setting up the design in a realistic mode;
- Writing protocol assumptions modeling the environment;
- Facing state-space explosion;
- Analyzing counter-examples.

We provide practical solutions for the verification engineer to overcome these issues, along with their theoretical principles. In particular:

- Chapter 1 motivates the formal verification of CDCs in hardware designs;

- Chapter 2 provides the formalism for a structural and a functional hardware model, along with its model checking;
- Chapter 3 focuses on the identification of clock configuration modes, with the intent of reaching a realistic clock setup;
- Chapter 4 tackles the state-space explosion problem, while providing guidance for counter-example analysis;
- Chapter 5 aims at creating missing protocol assumptions for under-specified environments;
- Chapter 6 concludes on the CDC formal verification flow, and provides perspectives for future works.

CHAPTER 2

Formal Verification of a Hardware Model

Among all following notations, vector indexes are super-scripted; time and context are sub-scripted. Notation $\mathbb{B} = \{\text{true}, \text{false}\}$ denotes the Boolean set, and $\mathbb{N}$ denotes the set of integers including zero.

The Kleene closure of the Cartesian product of a set $\mathcal{A}$ is referred to as $\mathcal{A}^*$. The cardinality of a finite set $\mathcal{A}$ is expressed as $|\mathcal{A}|$ (e.g., $|\mathbb{B}| = 2$). Note that vectors are considered as ordered sets. For any vector $\omega = \langle \omega^1, \omega^2, \ldots, \omega^n \rangle$, we respectively define a prefix, suffix and sub-sequence with the following notations:

$$\forall 0 \leq i \leq n, \omega^{\cdot\cdot i} = \langle \omega^1, \ldots, \omega^{i-1}, \omega^i \rangle$$
$$\forall 0 \leq i \leq n, \omega^{i\cdot\cdot} = \langle \omega^i, \omega^{i+1}, \ldots, \omega^n \rangle$$
$$\forall 0 \leq i_1 \leq i_2 \leq n, \omega^{i_1 \cdot\cdot i_2} = \langle \omega^{i_1}, \omega^{i_1+1}, \ldots, \omega^{i_2-1}, \omega^{i_2} \rangle$$

As they are foundations of this thesis, it is expected from the reader to be familiar with the following notions:

- Hardware description languages such as VHDL [37] or Verilog [36];
- Formal specification languages such as LTL [69], PSL [38] or SVA [39];
- The main principles of model checking [6, 22, 24].

2.1 Hardware Modeling

Hardware is often described using a so-called hardware description language. These languages are represented by both a structural semantics and a functional semantics. To model this duality, we use two different graphs: a vertex-labeled directed graph (structural model) and a Moore machine (functional model).

2.1.1 Informal Introduction: the Handshake Example

In many hardware designs, data transfer across modules is using the handshake protocol. The 4-phase handshake transaction includes a *request* and an *acknowledge* signal, as shown in Figure 2.1. We will use this protocol as an example to illustrate the following sections.

Using an additional input signal α to model the variable duration of the phases 2 and 4, the netlist description in Figure 2.2 shows a potential circuit generating the acknowledge control signal.

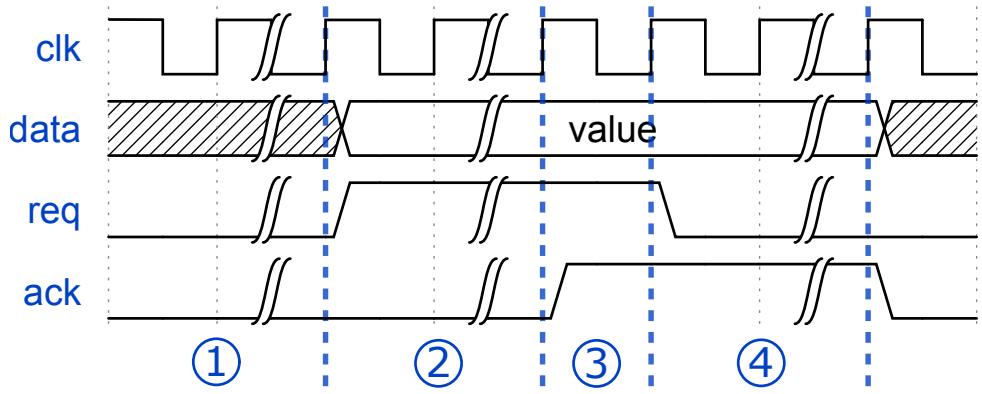


Figure 2.1: Waveform of a standard handshake transaction

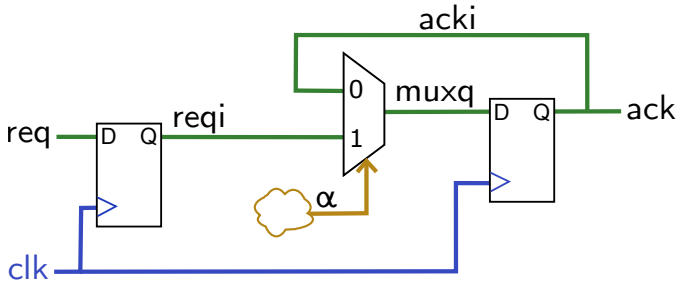


Figure 2.2: Handshake netlist

The corresponding functional model of this handshake design is represented as a state transition graph in Figure 2.3. Even though they are part of the model, the brown dotted arrows represent transitions which should not happen in a correct handshake protocol.

2.1.2 Netlist Model

When RTL is synthesized into a netlist, it creates a graph of components among: primary input, primary output, constant zero, constant one, inverter, and-gate, or-gate and sequential operator. For instance, Figure 2.4 exhibits the structural graph model of the handshake netlist defined in Figure 2.2, where each operator type is represented with a different symbol. All these operators have one or more inputs, and a single output. In the following, we shall call *signal* the output of an operator instance in a design, and we shall use the word *operator* when we refer to the functionality.

2.1.2.1 Formal Representation

The overall structural design model is defined using a directed graph with labeled vertices $\mathcal{D} = \langle \mathcal{A}, \mathcal{E}, \mathsf{Type} \rangle$, where:

- $\mathcal{A}$ is the set of vertices (also called signals);

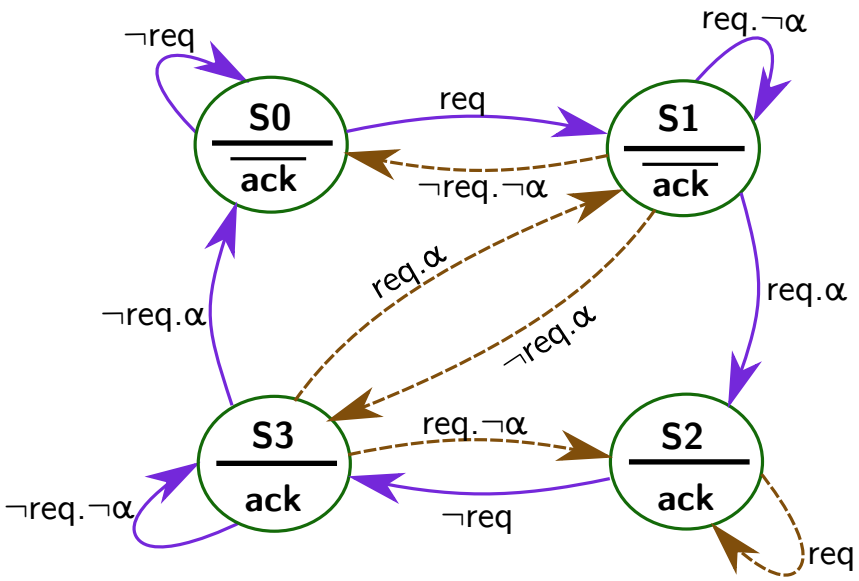


Figure 2.3: Handshake state transition graph

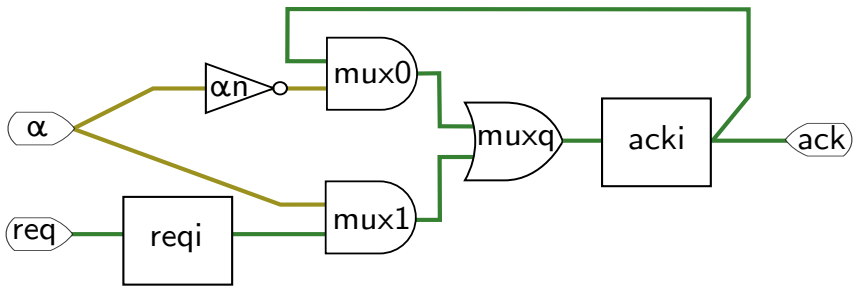


Figure 2.4: Structural graph of the handshake design

- $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$ is the set of edges;
- $\mathsf{Type} : \mathcal{A} \to \mathcal{T}$ is the signal-labeling function, with $\mathcal{T} = \{\mathcal{T}_{\mathsf{in}}, \mathcal{T}_{\mathsf{out}}, \mathcal{T}_{\mathsf{zero}}, \mathcal{T}_{\mathsf{one}}, \mathcal{T}_{\mathsf{not}}, \mathcal{T}_{\mathsf{and}}, \mathcal{T}_{\mathsf{or}}, \mathcal{T}_{\mathsf{seq}}\}$.

For the ease of notation, the subset of signals of a type $t \in \mathcal{T}$ is defined as:

$$\mathcal{A}_t = \{\alpha \in \mathcal{A} \,|\, \mathsf{Type}(\alpha) = t\}$$

We can then define: $\mathcal{A}_{\mathsf{in}}$, $\mathcal{A}_{\mathsf{out}}$, $\mathcal{A}_{\mathsf{zero}}$, $\mathcal{A}_{\mathsf{one}}$, $\mathcal{A}_{\mathsf{not}}$, $\mathcal{A}_{\mathsf{and}}$, $\mathcal{A}_{\mathsf{or}}$ and $\mathcal{A}_{\mathsf{seq}}$. We consider that all the signals in a set $\mathcal{A}_t$ are ordered. For instance, the notation $\mathcal{A}^i_{\mathsf{in}}$ represents the i^{th} primary input, where: $1 \leq i \leq |\mathcal{A}_{\mathsf{in}}|$.

As an example, the handshake design model of Figure 2.4 is defined by:

- $\mathcal{A} = \{\alpha, \alpha\mathsf{n}, \mathsf{req}, \mathsf{reqi}, \mathsf{mux0}, \mathsf{mux1}, \mathsf{muxq}, \mathsf{acki}, \mathsf{ack}\}$
- $\mathcal{E} = \{\langle\alpha, \alpha\mathsf{n}\rangle, \langle\alpha, \mathsf{mux1}\rangle, \langle\alpha\mathsf{n}, \mathsf{mux0}\rangle, \langle\mathsf{mux1}, \mathsf{muxq}\rangle, \langle\mathsf{mux0}, \mathsf{muxq}\rangle,$ $\langle\mathsf{muxq}, \mathsf{acki}\rangle, \langle\mathsf{acki}, \mathsf{ack}\rangle, \langle\mathsf{acki}, \mathsf{mux0}\rangle, \langle\mathsf{req}, \mathsf{reqi}\rangle, \langle\mathsf{reqi}, \mathsf{mux1}\rangle\}$
- $\mathcal{A}_{\mathsf{in}} = \{\mathsf{req}, \alpha\}$
- $\mathcal{A}_{\mathsf{out}} = \{\mathsf{ack}\}$
- $\mathcal{A}_{\mathsf{not}} = \{\alpha\mathsf{n}\}$
- $\mathcal{A}_{\mathsf{or}} = \{\mathsf{muxq}\}$
- $\mathcal{A}_{\mathsf{and}} = \{\mathsf{mux1}, \mathsf{mux0}\}$
- $\mathcal{A}_{\mathsf{seq}} = \{\mathsf{reqi}, \mathsf{acki}\}$

The *successors* of a signal in model $\mathcal{D}$ are the signals directly succeeding it:

$$\forall\alpha \in \mathcal{A}, \mathrm{Succ}(\alpha) = \{\alpha' \in \mathcal{A} \,|\, \langle\alpha, \alpha'\rangle \in \mathcal{E}\}$$

The *predecessors* of a signal in $\mathcal{D}$ are the signals directly preceding it (also called inputs of the operator):

$$\forall\alpha \in \mathcal{A}, \mathrm{Pred}(\alpha) = \{\alpha' \in \mathcal{A} \,|\, \langle\alpha', \alpha\rangle \in \mathcal{E}\}$$

Note that the $\mathcal{T}_{\mathsf{seq}}$, $\mathcal{T}_{\mathsf{out}}$ and $\mathcal{T}_{\mathsf{not}}$ operators only have one predecessor; the operators $\mathcal{T}_{\mathsf{in}}$, $\mathcal{T}_{\mathsf{zero}}$ and $\mathcal{T}_{\mathsf{one}}$ have no predecessor; the operator $\mathcal{T}_{\mathsf{out}}$ has no successor:

- $\forall\alpha \in (\mathcal{A}_{\mathsf{seq}} \cup \mathcal{A}_{\mathsf{out}} \cup \mathcal{A}_{\mathsf{not}}), |\mathrm{Pred}(\alpha)| = 1$
- $\forall\alpha \in (\mathcal{A}_{\mathsf{in}} \cup \mathcal{A}_{\mathsf{zero}} \cup \mathcal{A}_{\mathsf{one}}), |\mathrm{Pred}(\alpha)| = 0$
- $\forall\alpha \in \mathcal{A}_{\mathsf{out}}, |\mathrm{Succ}(\alpha)| = 0$

A *structural path* from α to α' is a vector of successive signals $\pi_{\alpha..\alpha'} = \langle\pi^1, \pi^2, \ldots, \pi^n\rangle \in \mathcal{A}^*$, where:

- $\pi^1 = \alpha$
- $\pi^n = \alpha'$

- $\forall 1 \leq i < n, \langle \pi^i, \pi^{i+1} \rangle \in \mathcal{E}$

We call *cone of influence* (or fan-in) of a signal α' the set of signals from which there exists a structural path to α':

$$\mathrm{COI}_{\alpha'} = \{\alpha \in \mathcal{A} \,|\, \exists \pi_{\alpha..\alpha'}\}$$

A structural path π is called *combinational* iff it does not include any sequential operators, i.e.:

$$\forall 1 \leq i \leq |\pi|, \pi^i \notin \mathcal{A}_{\mathsf{seq}}$$

Throughout this thesis, we assume that the graph has no combinational loop: there is no repeating signal in a combinational path. More precisely, it means that $\mathcal{D}$ has a finite number of combinational paths.

2.1.2.2 Complex Operators Modeling

The design graph $\mathcal{D}$ is composed of only eight types of simple components. More complex operators like the multiplexer, the latch or the flip-flop are then modeled by combining these simple components. For instance, a multiplexer (mux) can be modeled by two AND-gates, an OR-gate and an inverter, as shown in Figure 2.5.

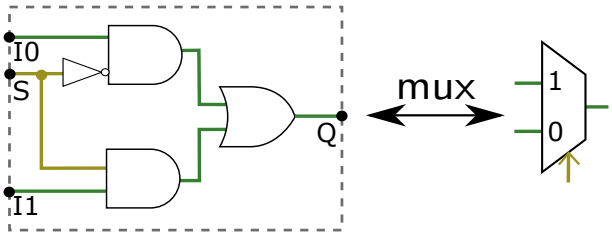


Figure 2.5: Multiplexer model

Also, if all the sequential elements in the design are flip-flops and are sampling on one common clock, the $\mathcal{T}_{\mathsf{seq}}$ component (called the *simple flop model* hereafter) can be used to model all of them. Otherwise, in case the design contains latches or multiple clocks, more complex models are necessary. A latch is modeled with a multiplexer and a sequential operator (see Figure 2.6).

Our flip-flop model (also called register) is composed of latches enabled on different polarities of the clock (see Figure 2.7). The model is also enhanced with an asynchronous reset to force the output to a constant zero or one.

To facilitate the manipulation of complex operators, we introduce notations for the signals and edges related to the multiplexer, flop and latch models. The set $\mathcal{A}_{\mathsf{mux}}$ contains all signals which are the outputs of a multiplexer. Similarly, we define $\mathcal{A}_{\mathsf{flop}}$ and $\mathcal{A}_{\mathsf{latch}}$. With the following notations, the input signals of these operators are identified :

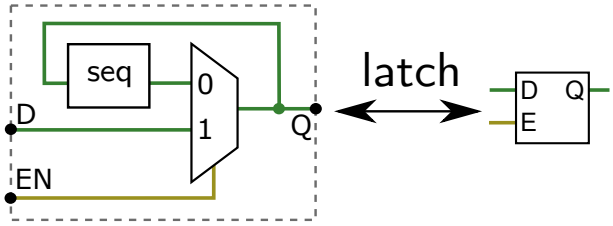


Figure 2.6: Latch model

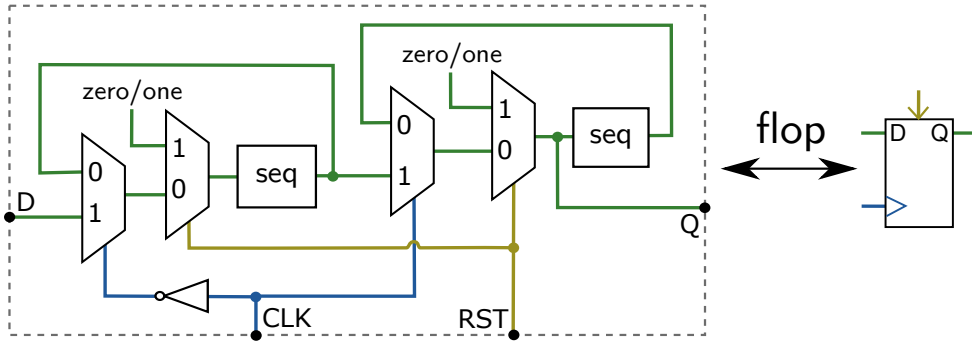


Figure 2.7: Flop model

- $\mathcal{A}_{\mathsf{mux,I0}}$, $\mathcal{A}_{\mathsf{mux,I1}}$, $\mathcal{A}_{\mathsf{mux,S}}$ represent the multiplexer input signals;
- $\mathcal{A}_{\mathsf{flop,D}}$, $\mathcal{A}_{\mathsf{flop,CLK}}$, $\mathcal{A}_{\mathsf{flop,RST}}$ represent the flop input signals;
- $\mathcal{A}_{\mathsf{latch,D}}$, $\mathcal{A}_{\mathsf{latch,EN}}$ represent the latch input signals.

The relation between a flop output and its clock input is given by function:

$$\mathsf{getClockOf} : \mathcal{A}_{\mathsf{flop}} \rightarrow \mathcal{A}_{\mathsf{flop,CLK}}$$

In the case of the handshake from Figure 2.4, the complex operators modeling allows to see the netlist as in Figure 2.2. Because there is only one deterministic clock, the simple flop model $\mathcal{T}_{\mathsf{seq}}$ is used, and the clock is abstracted. The complex operator sets are:

$$\mathcal{A}_{\mathsf{mux}} = \{\mathsf{muxq}\} \qquad \mathcal{A}_{\mathsf{flop}} = \{\mathsf{reqi}, \mathsf{acki}\} \qquad \mathcal{A}_{\mathsf{latch}} = \varnothing$$
$$\mathcal{A}_{\mathsf{mux,I0}} = \{\mathsf{acki}\} \qquad \mathcal{A}_{\mathsf{flop,D}} = \{\mathsf{req}, \mathsf{muxq}\} \qquad \mathcal{A}_{\mathsf{latch,D}} = \varnothing$$
$$\mathcal{A}_{\mathsf{mux,I1}} = \{\mathsf{reqi}\} \qquad \mathcal{A}_{\mathsf{flop,CLK}} = \varnothing \qquad \mathcal{A}_{\mathsf{latch,EN}} = \varnothing$$
$$\mathcal{A}_{\mathsf{mux,S}} = \{\alpha\} \qquad \mathcal{A}_{\mathsf{flop,RST}} = \varnothing$$

2.1.3 Functional Model

2.1.3.1 Formal Representation

To build the functional model $\mathcal{M}$ upon $\mathcal{D}$, each signal is associated to a binary value. The semantics $\mathcal{M}$ is described as a deterministic finite state machine where each state

corresponds to a combination of sequential operator values. The transition from one state to the next happens after evaluating the combinational logic, and updating the individual values of all sequential elements. This Moore machine models the design as $\mathcal{M} = \langle \Sigma_s, s_0, \Sigma_l, \Sigma_g, \delta, \lambda \rangle$, where:

- $\Sigma_s = \mathbb{B}^{|\mathcal{A}_{\mathsf{seq}}|}$ is the set of state values;
- $\Sigma_{\mathrm{s},0} \subseteq \Sigma_s$ is the set of initial states;
- $\Sigma_l = \mathbb{B}^{|\mathcal{A}_{\mathsf{in}}|}$ is the input alphabet;
- $\Sigma_g = \mathbb{B}^{|\mathcal{A}_{\mathsf{out}}|}$ is the output alphabet;
- $\delta : \Sigma_s \times \Sigma_l \to \Sigma_s$ is a total state transition function;
- $\lambda : \Sigma_s \to \Sigma_g$ is a total output function.

The global alphabet on the valuations of input, output and sequential signals is:

$$\Sigma = \Sigma_l \times \Sigma_g \times \Sigma_s$$

A letter of this alphabet is called a *configuration* of the machine $\mathcal{M}$, and is represented by a triplet $c = \langle l, g, s \rangle \in \Sigma$, such that:

$$\lambda(s) = g$$

At a time point $t \in \mathbb{N}$, a configuration is defined by $c_t = \langle l_t, g_t, s_t \rangle$, where:

- $l_t = \langle l_t^1, l_t^2, \ldots, l_t^{|\mathcal{A}_{\mathsf{in}}|} \rangle \in \Sigma_l$ is the input letter
- $g_t = \langle g_t^1, g_t^2, \ldots, g_t^{|\mathcal{A}_{\mathsf{out}}|} \rangle \in \Sigma_g$ is the output letter
- $s_t = \langle s_t^1, s_t^2, \ldots, s_t^{|\mathcal{A}_{\mathsf{seq}}|} \rangle \in \Sigma_s$ is the state

An *execution* of the machine $\mathcal{M}$ is defined as a (finite or infinite) sequence of configurations $\omega = \langle c_0, c_1, c_2, \ldots \rangle$ starting from an initial state, such that:

- $s_0 \in \Sigma_{\mathrm{s},0}$
- $\forall t \in \mathbb{N}, \delta(s_t, l_t) = s_{t+1}$

The set of all executions is a subset of Σ^*, denoted Ω. A configuration c is called *reachable* if it is part of an execution of the machine.

2.1.3.2 Signal Valuation

The relation between the structure and the functionality of an RTL design is implied by the valuation function Val_ω:

$$\mathrm{Val}_\omega : \mathcal{A} \times \mathbb{N} \longrightarrow \mathbb{B}$$

Within the context of an execution $\omega \in \Omega$, Val_ω gives the value of a signal at a time point.

$$\forall t \in \mathbb{N}, \begin{cases} \forall 1 \leq i \leq |\mathcal{A}_{\mathsf{in}}|, \mathrm{Val}_\omega(\mathcal{A}^i_{\mathsf{in}}, t) = l^i_t \\ \forall 1 \leq i \leq |\mathcal{A}_{\mathsf{out}}|, \mathrm{Val}_\omega(\mathcal{A}^i_{\mathsf{out}}, t) = g^i_t \\ \forall 1 \leq i \leq |\mathcal{A}_{\mathsf{seq}}|, \mathrm{Val}_\omega(\mathcal{A}^i_{\mathsf{seq}}, t) = s^i_t \end{cases}$$

The type of an operator defines a relation between its value and the values of its inputs (if any). The properties of the valuation function Val_ω follow:

$$\forall t \in \mathbb{N}, \begin{cases} \forall \alpha \in \mathcal{A}_{\mathsf{zero}}, & \mathrm{Val}_\omega(\alpha, t) = \mathrm{false} & \\ \forall \alpha \in \mathcal{A}_{\mathsf{one}}, & \mathrm{Val}_\omega(\alpha, t) = \mathrm{true} & \\ \forall \alpha \in \mathcal{A}_{\mathsf{out}}, & \mathrm{Val}_\omega(\alpha, t) = \mathrm{Val}_\omega(\alpha', t) & \text{with } \{\alpha'\} = \mathrm{Pred}(\alpha) \\ \forall \alpha \in \mathcal{A}_{\mathsf{not}}, & \mathrm{Val}_\omega(\alpha, t) = \neg \mathrm{Val}_\omega(\alpha', t) & \text{with } \{\alpha'\} = \mathrm{Pred}(\alpha) \\ \forall \alpha \in \mathcal{A}_{\mathsf{and}}, & \mathrm{Val}_\omega(\alpha, t) = \bigwedge_{\alpha' \in \mathrm{Pred}(\alpha)} \mathrm{Val}_\omega(\alpha') & \\ \forall \alpha \in \mathcal{A}_{\mathsf{or}}, & \mathrm{Val}_\omega(\alpha, t) = \bigvee_{\alpha' \in \mathrm{Pred}(\alpha)} \mathrm{Val}_\omega(\alpha') & \\ \forall \alpha \in \mathcal{A}_{\mathsf{seq}}, & \mathrm{Val}_\omega(\alpha, t+1) = \mathrm{Val}_\omega(\alpha', t) & \text{with } \{\alpha'\} = \mathrm{Pred}(\alpha) \end{cases}$$

The only non-determinism of Val_ω lies in the values of type $\mathcal{T}_{\mathsf{in}}$ and of type $\mathcal{T}_{\mathsf{seq}}$ at time point 0.

For instance, a correct handshake transaction – as in Figure 2.1 – reduced to signals req and ack is:

$$\begin{aligned} \omega &= \Big\langle \big\langle \mathrm{Val}_\omega(\mathrm{req}, 0), \mathrm{Val}_\omega(\mathrm{ack}, 0) \big\rangle, \ldots, \big\langle \mathrm{Val}_\omega(\mathrm{req}, 6), \mathrm{Val}_\omega(\mathrm{ack}, 6) \big\rangle \Big\rangle \\ &= \Big\langle \langle 0,0 \rangle, \langle 1,0 \rangle, \langle 1,0 \rangle, \langle 1,1 \rangle, \langle 0,1 \rangle, \langle 0,1 \rangle, \langle 0,0 \rangle \Big\rangle \end{aligned}$$

2.2 Formal Verification

2.2.1 Satisfiability

The formal verification of a model is based on formal properties which define a correct design behavior. The property formalism we use is the *linear temporal logic* (LTL). We build an LTL formula over the model $\mathcal{M}$ by considering all signals in $\mathcal{A}$ as atomic propositions. In particular, we are interested in the subset $\mathcal{P}$ of safety LTL formulas [47, 57].

As proved in [76], a safe LTL property is constructed in a positive normal form, using the usual Boolean logic and the temporal operators X (next), U (weak until) and G (always). The syntax of an LTL property in $\mathcal{P}$ is recursively defined as follows:

- $\mathrm{true} \in \mathcal{P}$
- $\mathrm{false} \in \mathcal{P}$
- $\forall \alpha \in \mathcal{A}, \alpha \in \mathcal{P}$

- $\forall \alpha \in \mathcal{A}, \neg\alpha \in \mathcal{P}$
- $\forall \phi \in \mathcal{P}, \mathsf{X}\phi \in \mathcal{P}$
- $\forall \phi \in \mathcal{P}, \mathsf{G}\phi \in \mathcal{P}$
- $\forall (\phi, \phi') \in \mathcal{P}, \phi \wedge \phi' \in \mathcal{P}$
- $\forall (\phi, \phi') \in \mathcal{P}, \phi \vee \phi' \in \mathcal{P}$
- $\forall (\phi, \phi') \in \mathcal{P}, \phi \,\mathsf{U}\, \phi' \in \mathcal{P}$

The semantics of the LTL language are represented by the satisfaction – or failure – of a property. We use the notation $\omega \vDash \phi$ to indicate that a property $\phi \in \mathcal{P}$ is satisfied in the sequence of configurations $\omega \in \Sigma^*$. The satisfaction relation $\vDash$ and its negation $\nvDash$ are inductively defined as:

- $\omega \vDash \text{true}$
- $\omega \nvDash \text{false}$
- $\forall \alpha \in \mathcal{A}, \forall i \geq 1,\ \omega^{i..} \vDash \alpha \iff \text{Val}_\omega(\alpha, i-1)$
- $\forall \alpha \in \mathcal{A}, \forall i \geq 1,\ \omega^{i..} \vDash \neg\alpha \iff \neg\text{Val}_\omega(\alpha, i-1)$
- $\forall \phi \in \mathcal{P},\ \omega \vDash \mathsf{X}\phi \iff \omega^{2..} \vDash \phi$
- $\forall \phi \in \mathcal{P},\ \omega \vDash \mathsf{G}\phi \iff \forall i \geq 1, \omega^{i..} \vDash \phi$
- $\forall (\phi, \phi') \in \mathcal{P},\ \omega \vDash \phi \wedge \phi' \iff (\omega \vDash \phi) \wedge (\omega \vDash \phi')$
- $\forall (\phi, \phi') \in \mathcal{P},\ \omega \vDash \phi \vee \phi' \iff (\omega \vDash \phi) \vee (\omega \vDash \phi')$
- $\forall (\phi, \phi') \in \mathcal{P},\ \omega \vDash \phi \,\mathsf{U}\, \phi' \iff \begin{array}{l}(\exists j \in \mathbb{N}, (\omega^{j..} \vDash \phi') \wedge (\omega^{..j-1} \vDash \mathsf{G}\phi)) \\ \vee (\omega \vDash \mathsf{G}\phi)\end{array}$

When all possible executions of a given design model satisfy a property ϕ, we say that the model satisfies the property:

$$\mathcal{M} \vDash \phi \iff \forall \omega \in \Omega, \omega \vDash \phi$$

A property that is used to verify the behavior of the model is called an *assertion*. Conversely, an *assumption* χ (also called constraint) is a property that specifies the design environment, and which restricts the set of admissible input value sequences. In effect, an assumption restricts the behavior of the model, which is strongly related to the assume-guarantee paradigm from [23, 70]. When $\mathcal{M}$ is constrained by χ, then it is denoted $\mathcal{M}|\chi$ and is defined by:

$$\mathcal{M}|\chi \vDash \phi \iff \forall \omega \in \Omega, \omega \vDash \chi \rightarrow \omega \vDash \phi$$

When model $\mathcal{M}$ constrained by χ does not satisfy property ϕ, then we say that the property is failed. Note that an execution exhibiting a failure of ϕ (also called a *counter-example*, or CEx) will satisfy χ.

$$\mathcal{M}|\chi \nvDash \phi \iff \exists \omega \in \Omega, \omega \vDash \chi \wedge \omega \nvDash \phi$$

2.2.2 Property Monitors

In the context of hardware verification, properties are usually written in the *Property Specification Language* PSL [38] or in SystemVerilog Assertions SVA [39], which provide a different syntax to LTL formulas. Throughout this thesis, we will use the PSL language. For instance, consider the following PSL property ϕ and its LTL equivalent, built over the signals of the handshake design from Section 2.1.1:

$$\begin{array}{ll} \mathit{PSL}: & \textbf{always}\ \mathrm{ack} \rightarrow \textbf{next}\ !\,\mathrm{req} \\ \mathit{LTL}: & \mathsf{G}(\neg \mathrm{ack} \vee \mathsf{X} \neg \mathrm{req}) \end{array}$$

Property ϕ semantics can be described as a Moore machine $\mathcal{M}_\phi$ [16, 80, 81] the input alphabet of which is in $\mathbb{B}^{|\mathcal{A}|}$, and output alphabet is $\mathbb{B}$. The output letter of this machine indicates whether the property is satisfied or failed. Hence, we call this machine the *property monitor*.

As demonstrated in [2, 11, 13, 30, 62] such machine is *synthesizable* into hardware. The corresponding structural representation $\mathcal{D}_\phi$ takes all signals of $\mathcal{D}$ as inputs, and outputs a single new signal denoted α_ϕ. Within the context of an execution, the positive (respectively negative) value of α_ϕ represents the failure (respectively satisfaction) of property ϕ. Thus, execution ω satisfies property ϕ up to a time point if the value of α_ϕ stays false:

$$\omega \vDash \phi \iff \forall t \in \mathbb{N}, \neg \mathrm{Val}_\omega(\alpha_\phi, t)$$

When the model is constrained, the satisfaction of the assertion is then defined as:

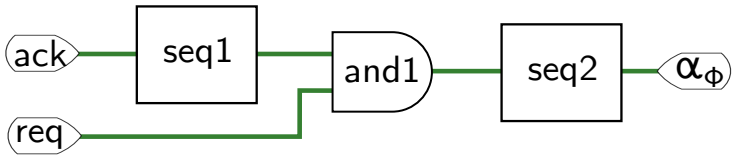
$$\mathcal{M}|\chi \vDash \phi \iff \forall \omega \in \Omega, (\exists t \in \mathbb{N}, \mathrm{Val}_\omega(\alpha_\chi, t)) \vee (\forall t \in \mathbb{N}, \neg \mathrm{Val}_\omega(\alpha_\phi, t))$$

Taking the PSL property written above, the corresponding circuit monitor and state graph are represented in Figure 2.8. As we consider a Moore machine (and not a Mealy one), the final value of the monitor is latched. Here, Σ_s is the set of valuations of $\langle \mathsf{seq1}, \mathsf{seq2} \rangle$, and:

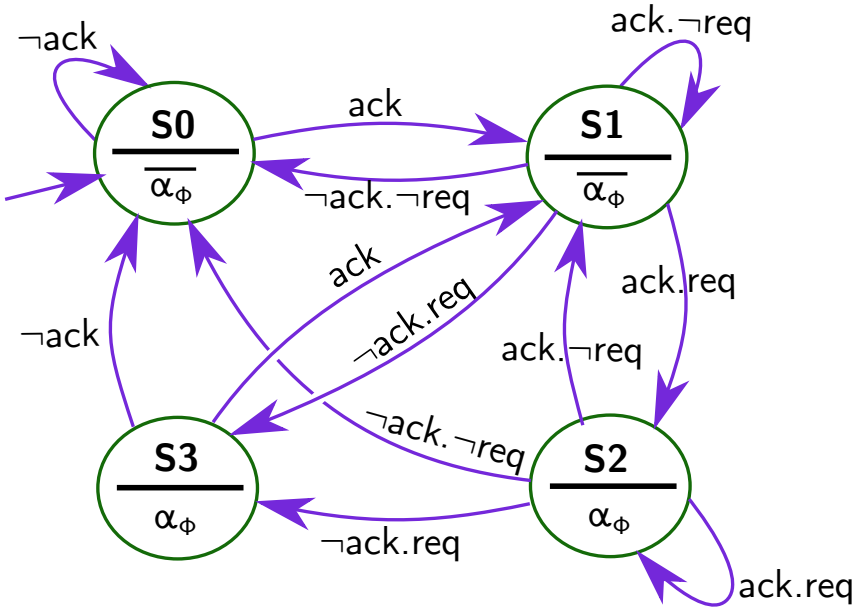
$$\mathsf{S0} = \langle 0, 0 \rangle \qquad \mathsf{S1} = \langle 0, 1 \rangle \qquad \mathsf{S2} = \langle 1, 0 \rangle \qquad \mathsf{S3} = \langle 1, 1 \rangle$$

2.2.3 Model Checking

Traditional model checkers verify the satisfaction of property ϕ by connecting its monitor $\mathcal{D}_\phi$ to $\mathcal{D}$. The corresponding state machine is the product of $\mathcal{M}_\phi$ and $\mathcal{M}$, and the states in which the value of α_ϕ is true are denoted *failing states*. If a failing state is reachable in an execution of the machine, then the model does not satisfy the property. Thus, verifying a safety property boils down to identifying the set of executions Ω (which implies the reachable state space).. Model checking can then be reduced to a *reachability* problem, searching for a failing state in a state transition graph.



(a) Structural representation of ϕ



(b) Functional representation of ϕ

Figure 2.8: Semantics for a property $\phi = \mathsf{G}(\neg\mathsf{ack} \vee \mathsf{X}\neg\mathsf{req})$

The failure of a property can be checked by a forward reachability analysis: starting from the initial states, the set of reachable states is computed cycle-after-cycle, until identifying a failing state. Conversely, a backward co-reachability check can also be performed: starting from the failing states, we iteratively compute the set of states from which a failing state is reachable, until identifying an initial state. In both approaches, whenever an execution is not found to reach a failing state, the property is reported as proved. Note that model checkers also combine reachability and co-reachability for performance improvement.

To compute the state-space, model checking methods rely either on the enumeration of states or on their symbolic representation, using techniques like binary decision diagrams (BBDs) or SAT-solvers. As enumeration and BDDs are not scalable on large designs, modern model checkers consider the symbolic computation based on SAT. The values of the (non-deterministic) primary inputs and the signals in $\mathcal{A}_{\mathsf{seq}}$ are represented as Boolean variables, and a SAT-solver computes the relation between their current value and their next value. In a sense, it corresponds to a symbolic definition of the transition function δ.

2.2.4 Inputs Modeling

Model checkers consider that primary inputs of the model take non-deterministic values, and try to verify the satisfiability of a property. When an assumption χ restricts the behavior of a primary input, the model $\mathcal{D}$ itself can be modified to include this behavior. First, a circuit modeling the assumption is created to output the intended signal value. This is known as Assertion-Based Synthesis [40, 63, 71]. Then, this circuit is appended to the model by connecting its output to the specified signal.

Note that this process cannot be applied on an internal signal. Indeed, within the process, the predecessors and the type of the signal are modified, which breaks causality. The values of the original predecessors could become inconsistent with the new signal behavior. This process is then only applied to signals in $\mathcal{A}_{\mathsf{in}}$ (i.e., primary inputs), since they have no predecessor.

This technique is in particular useful in case some inputs are specified to have a deterministic behavior: constants, clocks, and resets. Also, we consider this technique when inputs are described to be synchronized with a specific clock.

Constant Modeling. When a primary input is specified as a constant, then the model is modified: the type of the input is changed from $\mathcal{T}_{\mathsf{in}}$ to $\mathcal{T}_{\mathsf{zero}}$ or $\mathcal{T}_{\mathsf{one}}$.

Clock Modeling. Current model checkers only consider one overall clock of the state machine. When the design has multiple clock inputs with different periods, state-machines are created to model each clock as if they derived from one master clock (Figure 2.9). The outputs of the corresponding netlists are connected to the clock signals of the design. The overall model can then be clocked by a unique master clock, whose period is the greatest common divisor of the modeled clock periods. For instance, if the designer specifies two

signals as clocks of period 2ns and 5ns, then the master clock is considered to have a period of 1ns.

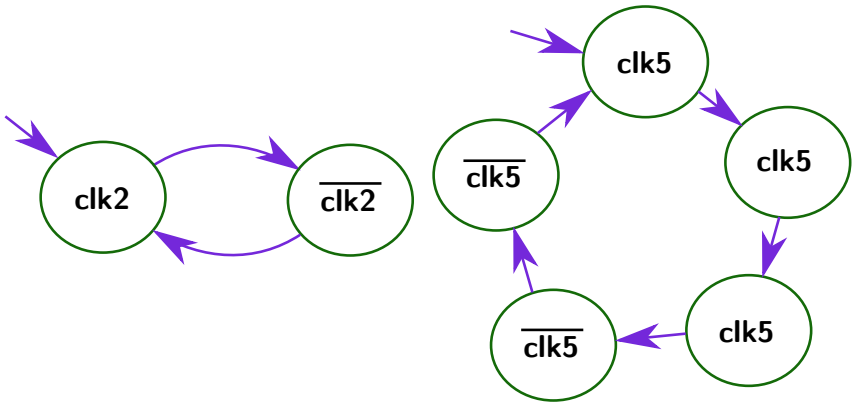


Figure 2.9: Example of state-machines for clock periods 2 and 5

Reset Modeling. In practice, the initial state s_0 has to be reached after powering up a circuit by an initialization phase, which may be potentially complex. If no specific initialization execution is predefined, it is then inferred by symbolically evaluating the system: all the resets are kept enabled (replaced by constants) until all register values are stable. This allows resets to propagate through registers, until a global stable state is reached. We assume the sequential values of this state as the initial state (which restricts the original $\Sigma_{s,0}$). In our context, properties should not be verified during the initialization phase. During all following formal checks, reset signals are replaced by constants corresponding to their disabled value.

Synchronous Input Modeling. When an input is specified to be synchronous to a given clock, then it is modeled with a flip-flop sampling on this clock. Thus, the D-input of this flop is a new primary input to the model.

CHAPTER 3

Reaching a Complete Design Setup

3.1 Context

3.1.1 Influence of Design Setup on Verification

In any verification flow, the first step is to setup the design so that the verification tools analyze the system in a realistic context. The verification engineer is required to provide information about the system environment, based on the design specification. These can be assumptions on the primary inputs (as seen in Section 2.2.4), on internal signals, or higher-level information (e.g. about a blackboxed internal module). For a CDC design, the crucial setup factor is to provide a correct configuration of the clocks.

Because the verification flow starts in parallel with the design specification, and because modules are reused from incompletely documented legacy components, we observe that the specification of the clock configurations is often not fully available. Also, when reusing predefined modules, designers do not know the exact clock configuration components. Thus, they typically miss some assumptions on configuration signals. As a consequence, the design is verified in a pessimistic mixed-mode, where controls toggle incoherently and unrealistically. With this incomplete setup, protocols may behave spuriously, leading to worthless verification results. The verification of protocols then requires a realistic and complete clock setup.

As an example, consider a design implementing the FIFO of Figure 1.11 (on page 9), where the source and the destination clocks are the output of two clock-gating cells. In a correct setup, the control signals of the clock-gates are assumed to be up, hence always enabling the clock to propagate. After writing the correct assertion, a model checker takes 13 seconds to guarantee that the FIFO cannot overflow. We then remove the assumption on the clock-gate controls, hence making them fully non-deterministic. Functionally, the clock can be enabled or disabled at any time. The same overflow property now takes 192 seconds to be proven (15 times more).

The reason for this runtime increase is an explosion in the design state-space. Indeed, the overall state machine is clocked by the same single virtual clock (see Section 2.2.4), but the sampling point of each register must now consider non-determinism. This exponentially increases the number of reachable states, hence making the formal verification harder.

When we consider that modern designs include thousands of clock-gating structures along with complex clock switching, running formal checks with an incomplete setup sounds intractable. The verification engineer must assume one configuration mode in which to verify the design. Several configurations can iteratively be checked, but since

a control protocol usually does not change between modes, verifying it in one may be enough. Finally, having a good understanding of the clock network and its modes helps provide a realistic setup to the tool very efficiently.

3.1.2 Clock Networks

To better understand clock networks, we need to define their constituents. We consider a clock input to be a periodic binary signal, used to indicate the sampling point of sequential operators. It is characterized by a period, duty cycle and phase.

Between these primary clock inputs and the sequential operators, designers implement complex clock control logic to manage the different modes of execution. Depending on the operation mode and the required performance, the clocks frequencies are then optimized [7, 83] using selection and transformation logic. This clock distribution network [28] is commonly referred to as the *clock tree*[1]. We distinguish four most common operations on a clock tree:

- Selection: choosing to propagate one clock or another;
- Blending: combining two clocks to generate a new one;
- Gating: propagating or stopping a clock;
- Shaping: modifying the period, duty cycle or phase of a clock.

Globally, a clock network considers two kinds of inputs: clocks and configuration signals (see Figure 3.3 for an generic overview). As an example, a typical clock network is shown in Figure 3.1. Here, CLK1 and CLK2 are the input clocks; CFG, SEL and EN are the configuration signals. This network includes a clock divider (shaping), a clock multiplexer (selection) and a clock-gate (gating). We may imagine that CLK1 is a test clock (with an external source) and CLK2 is the mission clock (with a PLL source). Signal SEL indicates if the design is in test mode or in mission mode. Depending on the power mode, signal CFG configures the division factor of the period of CLK2. When no data needs to be propagated, signal EN is set low, hence disabling the clock.

Note that the gating stage of this example is sequential as it includes a latch. This is a common clock-gate structure which avoids propagating a glitch on the clock rising edge (indeed, here the clock can only be disabled when it is low). Using two clock-gates with inverted control signals, a sequential mux could easily be implemented (with the same intent of glitch avoidance). Thus, we understand that the operations on a clock network can be implemented with a variety of custom combinational or sequential structures.

A clock is called a *primary clock* when it is a primary input of the design or an output of internal analog modules (e.g. a PLL or a crystal, which is then seen as a blackbox). We consider these clocks to be periodic and deterministic. On a clock network, the signals after selection, blending, gating or shaping, are called *derived clocks*. Note that a clock

[1] "clock tree" is misleading, as its formal representation is a directed graph.

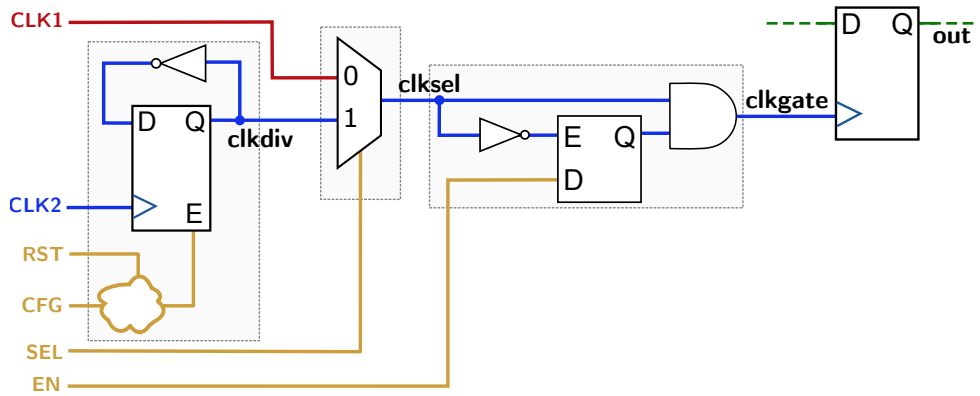


Figure 3.1: Typical clock network

can derive from a primary clock (e.g., CLKDIV derives from CLK2) or from another derived clock (e.g., CLKGATE derives from CLKSEL).

The *configuration signals* are mostly all non-clock inputs to the clock network. However, as the clock control becomes more and more complex, modern designs include configuration FSMs which gather information from both the environment and the ongoing tasks of the system in order to compute configuration values. Thus, configuration signals are identified as either primary inputs or internal registers.

We define an *operation mode* by the set of configuration signal values. Within such operation mode, configuration signals are then considered constant. Figure 3.2 exhibits an operation mode where CLK2 is selected after being divided by 4. Note that in a realistic execution, the system may dynamically switch between modes, depending on the required performance.

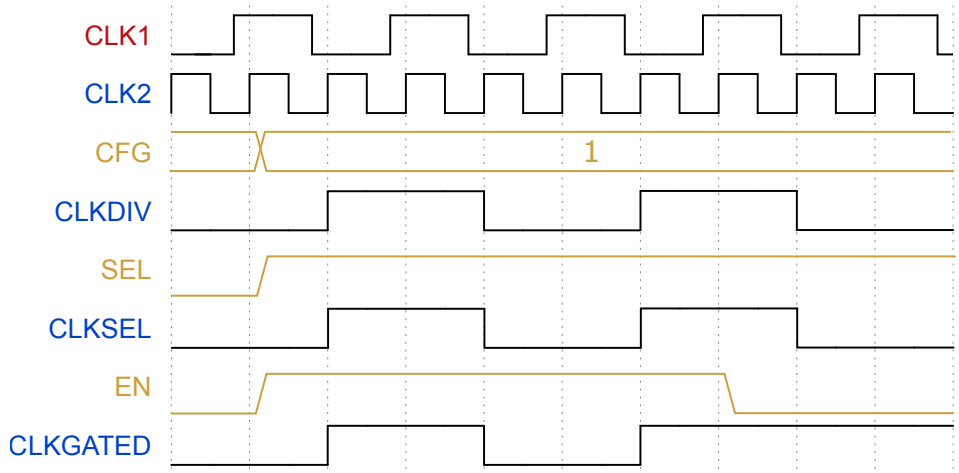


Figure 3.2: Clock network waveform

Considering the complexity and heterogeneity of clock networks, a purely structural analysis does not provide enough information on the actual operations. We need to

provide a functional analysis of the clock network and of its modes. As a result of this analysis, protocols will be verified in a realistic and complete design setup.

3.2 Clock Verification

3.2.1 Identification of the Clock Network

To verify the setup of a clock network, we need to identify its global architecture (Figure 3.3). First, the clock signals are determined. Then, the (external and internal) configurations signals are identified.

However, a clock network includes sequential transformations, and control loops, which makes its boundaries not trivial to find. For instance, structurally, there is no way to differentiate a configuration register from a register which is part of the shaping logic (the cloud in Figure 3.1). Thus, the derived clocks cannot be restricted to the clock pins of the registers, and the configuration signals cannot be restricted to the primary inputs. The identification of the clock paths and the control paths need to rely on heuristics.

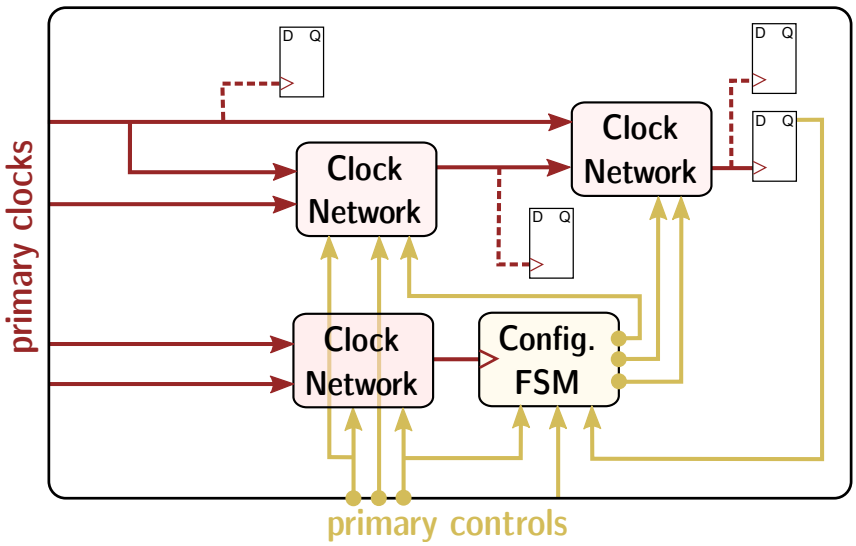


Figure 3.3: Global clock network

3.2.1.1 Clock Propagation

3.2.1.2 Configuration Signals

Configuration signals are identified as the inputs to the clock network which are not on the clock path. A first approximation would be to consider them as:

$$\bigcup_{\alpha_c \in \mathcal{A}_{clk}} \{\alpha \in \mathcal{A}_{in} \cap \mathrm{COI}(\alpha_c) \mid \alpha \notin \mathcal{A}_{clk}\}$$

However, as seen in 3.1.2, internal signals may also configure the clock network. We could then consider that the first sequential operators on the side of the clock path are configuration registers. This is incorrect, as there can be sequential logic (or even an FSM) between a configuration signal and the clock path.

SYNOPSYS CONFIDENTIAL

3.2.2 Liveness Property

SYNOPSYS CONFIDENTIAL

3.2.3 Parametric Analysis

SYNOPSYS CONFIDENTIAL

3.3 Mode Generation

SYNOPSYS CONFIDENTIAL

3.3.1 Justification

SYNOPSYS CONFIDENTIAL To compute this relevance, we use the justification algorithm of Mishchenko [61]. Within the context of a CEx ω exhibiting a failure of property ϕ, the justification function $J_{\omega,\phi}$ is recursively defined on the design netlist:

$$J_{\omega,\phi} : \mathcal{A} \times \mathbb{N} \longrightarrow \mathbb{B}$$

If signal α is relevant to the failure at time point t, then $J_{\omega,\phi}(\alpha, t)$ is true. Otherwise, it means that the signal value is a don't care value. The computation of function $J_{\omega,\phi}$ starts by considering that the property monitor signal α_ϕ is relevant whenever its value is true. Then, it traverses the netlist backwards, and for each gate, decides which of its inputs are relevant. For instance, if the output of an AND-gate is 0, and one of its inputs is 0, then this input is relevant. When the traversal reaches a relevant register, it continues on its

inputs but now justifying the previous time point. The justification process terminates when the traversal reaches the primary inputs at the first time point.

$$\begin{aligned}\forall \alpha \in \mathcal{A},\ \forall t \in \mathbb{N},\ J_{\omega,\phi}(\alpha,t) \iff & (\alpha = \alpha_\phi) \wedge \mathrm{Val}_\omega(\alpha,t) \\ & \vee \exists \alpha' \in \mathrm{Succ}(\alpha) \cap \mathcal{A}_{\mathsf{not}},\ J_{\omega,\phi}(\alpha',t) \\ & \vee \exists \alpha' \in \mathrm{Succ}(\alpha) \cap \mathcal{A}_{\mathsf{out}},\ J_{\omega,\phi}(\alpha',t) \\ & \vee \exists \alpha' \in \mathrm{Succ}(\alpha) \cap \mathcal{A}_{\mathsf{and}},\ J_{\omega,\phi}(\alpha',t) \wedge \mathrm{Val}_\omega(\alpha',t) \\ & \vee \exists \alpha' \in \mathrm{Succ}(\alpha) \cap \mathcal{A}_{\mathsf{and}},\ J_{\omega,\phi}(\alpha',t) \wedge \neg\mathrm{Val}_\omega(\alpha,t) \\ & \vee \exists \alpha' \in \mathrm{Succ}(\alpha) \cap \mathcal{A}_{\mathsf{or}},\ J_{\omega,\phi}(\alpha',t) \wedge \neg\mathrm{Val}_\omega(\alpha',t) \\ & \vee \exists \alpha' \in \mathrm{Succ}(\alpha) \cap \mathcal{A}_{\mathsf{or}},\ J_{\omega,\phi}(\alpha',t) \wedge \mathrm{Val}_\omega(\alpha,t) \\ & \vee \exists \alpha' \in \mathrm{Succ}(\alpha) \cap \mathcal{A}_{\mathsf{seq}},\ J_{\omega,\phi}(\alpha',t+1)\end{aligned}$$

A *justified execution* $\widehat{\omega}$ is then obtained using a new valuation function $\widehat{\mathrm{Val}_\omega}$ which replaces each non-justified signal value by a *don't care* value.

$$\begin{array}{rccl}\widehat{\mathrm{Val}_\omega}: & \mathcal{A}\times\mathbb{N} & \longrightarrow & \mathbb{B}\cup\{-\} \\ & \alpha,t & \mapsto & \begin{cases}\mathrm{Val}_\omega(\alpha,t) & \text{if } J_{\omega,\phi}(\alpha,t) \\ - & \text{otherwise}\end{cases}\end{array}$$

3.3.2 Blocking Condition

A model checking engine being deterministic, only one CEx can be generated for a triplet: model, assumptions and assertion. In order to generate a new CEx, a *blocking condition* is added. A blocking condition is a special kind of assumption which ensures that the same failing execution cannot happen in the newly constrained model.

Assume $\mathcal{M} \not\models \phi$ for some $\phi \in \mathcal{P}$. Let ω be a CEx that exhibits the failure of ϕ. A blocking condition χ_ω will force ω to be removed from the set of admissible executions[2] of $\mathcal{M}$:

$$\chi_\omega \subseteq \Omega - \{\omega\}$$

Checking the property under the blocking condition will result in a new CEx. In order to generate many CEx, a new blocking condition is iteratively created using the latest obtained CEx, and added to the global set of assumptions χ_{all}. Let $\chi_{\mathsf{all},i}$ be the set of assumptions at step i, and assume that ω_i is the CEx of $\mathcal{M}|\chi_{\mathsf{all},i} \not\models \phi$. Then,

[2]For ease of notation, we shall not distinguish a property and the set of executions on which the property is satisfied.

we can define the successive blocking conditions, starting with the original user-defined assumptions, and terminating at step k when $\mathcal{M}|\chi_{\mathsf{all},k} \vDash \phi$:

$$\begin{cases} \chi_{\mathsf{all},0} = \chi_{\mathsf{user}} \\ \forall i \in \mathbb{N}, i < k, \chi_{\mathsf{all},i+1} \subseteq \chi_{\mathsf{all},i} - \{\omega_i\} \end{cases}$$

SYNOPSYS CONFIDENTIAL Function Lit_ω is defined to represent the LTL literal corresponding to the valuation of a signal:

$$\begin{array}{rccl} \mathrm{Lit}_\omega : & \mathcal{A} \times \mathbb{N} & \longrightarrow & \mathcal{P} \\ & \alpha, t & \mapsto & \begin{cases} \alpha & \text{if } \mathrm{Val}_\omega(\alpha, t) \\ !\,\alpha & \text{otherwise} \end{cases} \end{array}$$

SYNOPSYS CONFIDENTIAL

3.3.3 Algorithm

SYNOPSYS CONFIDENTIAL

3.4 Clock Network Functional View

3.4.1 Table of Operation Modes

SYNOPSYS CONFIDENTIAL

3.4.2 Functional Schematics

SYNOPSYS CONFIDENTIAL

3.5 Experimental Results

In all following experiments, the formal verification back-end is the open source model checker ABC [14], used with the *BMC3* engine [10]. We run the experiments on a workstation with 20 Intel Xeon E5-2640 v4 at 2.4GHz CPUs and 250GB of memory.

3.5.1 Simple Clock network

As a proof of concept, this first experiment considers the simple clock tree from Figure 3.1. The logic on signal CFG includes a 2-bit counter (register and adder) whose value is compared to CFG. The complete Verilog description of this clock tree is given in Appendix C.

Signals CLK1 and CLK2 are defined as primary clocks, with respective periods of 10ns and 15ns. No other assumption is given on the design (in particular, no constants on the other inputs). The asynchronous reset signal rst, which is used in the counter, stays unconstrained.

SYNOPSYS CONFIDENTIAL

Interestingly, while the reset was left unconstrained, our clock analysis found out that it was required to be disabled for modes 3,4,5, and that its value does not matter for modes 1,2. Table 3.1 represents these modes with their respective source clock, shaping factor and configuration values. Indeed, within this clock network, the reset only affects the counter, which is not selected in mode 1 and always enables the division in mode 2. This observable non-determinism in the reset values will lead the verification engineer to provide a correct reset setup. Then, this clock analysis not only leads to a complete setup of the clock tree, but also of the reset.

Derived: clkgate		# 1	# 2	# 3	# 4	# 5
Source		CLK1	CLK2	CLK2	CLK2	CLK2
Frequency		-	1/2	1/4	1/6	1/8
Configurations	CFG	-	00	01	10	11
	SEL	0	1	1	1	1
	EN	1	1	1	1	1
	RST	-	-	0	0	0

Table 3.1: Operation modes for the simple clock network

Using these results, an engineer can review the potential modes of his design, and selects one (or more) for further verification steps. For instance, if one crucial mission mode to check is mode 2, the CFG, SEL and EN will be respectively assumed to be constants “00”, 1 and 1. In parallel, an overview of the clock network is given in Figure 3.4. Note that the table of operation modes and the functional schematics can be directly used to create or review the specification of the clock network and of the design modes.

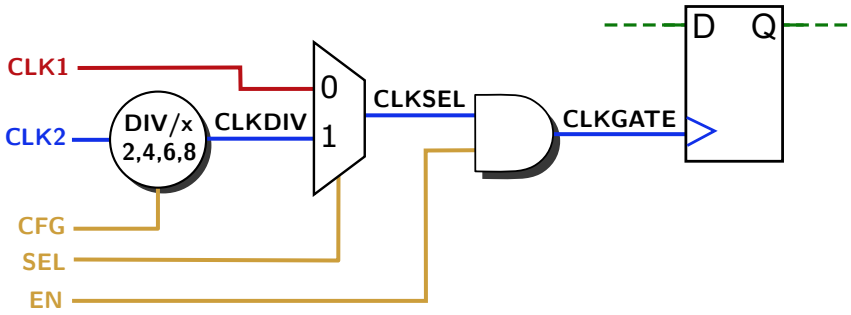


Figure 3.4: Schematic view of the simple clock network

3.5.2 OpenSPARC Clock Control Unit

The Oracle's OpenSPARC T2 is an open-source hardware including the UltraSPARC T2, a 64-bit 8-core multi-threaded microprocessor which is intended for servers [64]. The architecture encompasses a Clock Control Unit (CCU) which programs the PLL and outputs multiple clocks with various frequencies and shifts. As the PLL and its programming logic are using analog blocks and timing, it cannot be considered for our formal analysis. However, as per the design specification, the ccu_divider module is supposed to divide the programmed PLL clock with a factor of 2.

First, this single module is analyzed by considering that its clkin input is a primary clock. 6 configuration signals are identified on its inputs: div[4:0] and rst_n. In 28sec, 32 modes are generated for the clkout output clock. All modes exhibit a division of the input clock frequency (shaping operation), where signal div gives the value of the division factor, as represented in Table 3.2. Also, as we could expect, rst_n is a reset with an active-low value, which should be constrained as such.

Derived: clkout		#1	#2	#3	#4	...	#6	#7
Source		clkin	clkin	clkin	clkin	...	clkin	clkin
Frequency		1/2	1/3	1/4	1/5	...	1/32	1/64
Configuration signals	div	1	2	3	4	...	31	0
	rst_n	1	1	1	1	...	1	1

Table 3.2: Operation modes for the OpenSPARC

For another experiment at the hierarchical level of the CCU module, the primary clock is considered to be the output of the PLL/VCO module (signal l2clk). The propagation of this clock leads to gathering 9 configuration signals and 457 derived clocks, among which the output of the ccu_divider. In the verification of this signal (ccu_io2x_out), 136 modes are generated, all exhibiting a division by 2 of the primary clock. After running the Quine-McCluskey optimization, it appears that any combination of the configuration signals leads to a correct clock behavior. Hence, the specification has been formally proved to correctly define the behavior of clock ccu_io2x_out: a deterministic division by 2 of clock l2clk (see Figure 3.5).

3.5.3 Industrial CPU Subsystem

To challenge the scalability of our approach and the heuristics provided (in particular for the configuration signals detection), we consider a complex SoC hardware from an industrial partner, which has been recently designed for a gaming system. It has a low-power architecture, with a state-of-the-art quad-core CPU and many different interfaces (see Figure 3.6). 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

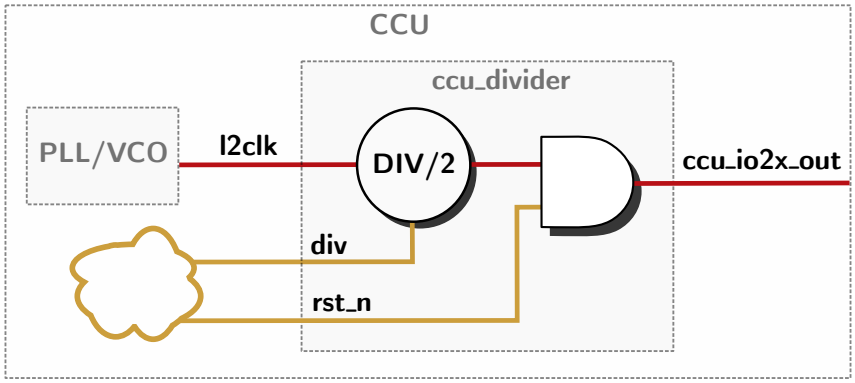


Figure 3.5: OpenSPARC clock network

for the whole system among its 38 primary clocks and 17 primary resets. Because this control module was reused from previous projects, and because an exhaustive specification was not available, the clock configurations and clock tree components were not perfectly understood by the verification engineers.

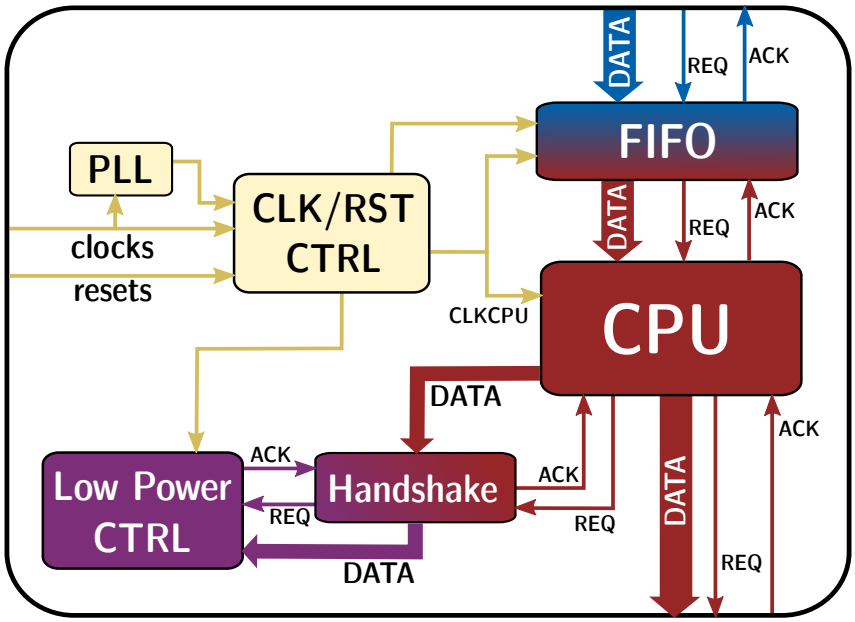


Figure 3.6: Overview on the synchronizers at the interface of the CPU

Primary clocks and resets are defined on the primary inputs of the design. The clock propagation leads to a global clock path of 450 sequential elements (most of them covered by 2 mission clocks). 134 configuration signals are identified, among which primary inputs and internal signals. Interestingly, 109 of these signals are identified to be the outputs of only 10 different modules. After review with designers, these modules were confirmed to be configuration FSMs. The remaining signals are either primary inputs or internal registers inside the clock control module. These observations give a strong confidence in

our heuristics to infer configuration registers.

To focus on the analysis of one derived clock, we select the clock signal feeding the CPU, as it is expected to contain the most complex logic. Indeed, the cone-of-influence of this derived clock (up to the configuration signals) contains the majority of the clock network: 403 clocks derived from 3 primary clocks. In 7171 seconds, 68 modes are identified to lead to a correct clock behavior on CLKCPU.

The abstract schematic of these modes is represented in Figure 3.7. It includes 4 selections, 1 divider, 0 shift, 3 clock-gates and 0 blending. Despite the fact that many clock-gates are inserted on the clock path in the RTL (over 15), they are actually part of other components. Hence, they are subsumed by these other operators, and are not represented in the schematics. For instance, in the RTL, two gating operators converging on an OR-gate are found multiple times. Although they don't share the same control signal, our analysis identified this scheme as a sequential multiplexer. After review with the designer, these structures were confirmed to be glitch-free multiplexers intended for a dynamic clock switching.

Note that CLK3, one of the three primary clocks propagating to CLKCPU, is not identified as a source in any mode. However, in the RTL, a sequential multiplexing stage is manually identified on CLKINT to select between CLK1 and CLK3. Actually, CLK3 only propagates to CLKINT in a specific boot-up phase of the system. In this phase, only CLK2 propagates to the CPU, and never CLK3. Hence, the multiplexing stage of CLKINT is not identified by our clock analysis (though, for clarity reasons, it has been manually added with dotted borders in Figure 3.7).

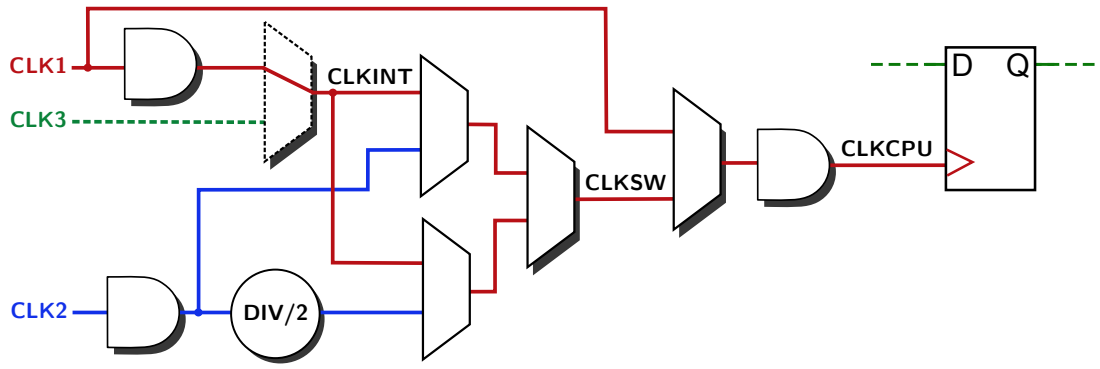


Figure 3.7: Industrial design clock network

After running Quine-McCluskey on the 68 modes, 52 modes and 20 configuration signals remain. The full table containing all modes and signals is available in Appendix D. As this single table containing all modes is too large for an easy review, we decide to represent the modes after a projection on one operator. With this intent, only the justified configuration signals in the cone-of-influence of the operator are kept. The modes of CLKCPU relatively to CLKSW and CLK1 are reported in Table 3.3, along with the signals configuring the corresponding multiplexer and clock-gate. For confidentiality reasons, the names of the signals are obfuscated.

Derived: CLKCPU		#1	#2	#3	#4
Source		CLK1	CLKSW	CLKSW	CLKSW
Frequency		-	-	-	-
Configuration	GE	-	1	1	1
	O2	1	0	0	0
signals	O39	-	0	0	0
	O40	-	0	-	-
	O30	-	-	0	0
	S0	-	-	1	-
	EM	-	-	-	1

Table 3.3: Partial operation modes for the industrial design

Two kinds of configuration signals are observed. Selecting signals (like O2) lead to propagating one clock or another, depending on their values. Enabling signals (like GE, O39, O40, O30, S0, EM) must be set to a constant value in order to lead to a correct clock behavior. Among these, some signals are reviewed by the designer as reset (like O39 and O40), and must be constrained as such.

Using the full table of modes and the clock network schematics, the user would be able to select one or multiple modes, hence generating the constraints for a correct and complete design setup.

3.6 Conclusion

The main contribution of this chapter is a fully automatic flow to formally analyze clock networks. We provide algorithms to detect the major components of a clock network and infer its functional modes. This work is derived from a broader work in the context of the verification of so-called *generated clocks* in the Synopsys Design Constraints (SDC) format, which is currently undergoing a provisional patent application by Synopsys Inc. [74].

SYNOPSYS CONFIDENTIAL

Using these modes and a structural analysis of the netlist, a functional overview of the whole clock network is reported. Thanks to the schematic overview and the table of modes, designers may notice false or redundant logic, and optimize the clock network. Verification engineers can then use these results to provide the complete and realistic setup to the design, and proceed with further formal verification checks. Also, these reports can be made available as a functional specification of the clock network.

CHAPTER 4

Avoiding State-Space Explosion

4.1 Context

4.1.1 Limitations of Model Checking

Functional checks are a mandatory step for an exhaustive verification of hardware protocols. When running a formal verification, as described in Chapter 1, a *conclusive result* is expected: either a proof or a failure of the assertion. However, experience shows that model checking a property is not scalable on large hardware designs: for some properties, one may not achieve a conclusive result, even after several days.

Indeed, within the context of a CDC design, each transition of the state machine depends on the values of all the clocks (see Section 2.2.4 on page 24). The complexity of the state machine is then largely influenced by the number of clocks and their periods (see Section 3.1.1 on page 27). Also, modern designs contain hundreds of thousands of registers, large-size data buses, and deal with concurrent protocols. With this increasing system complexity comes an increasing state space. Thus, model checking sequential protocols in large hardware designs often leads to inconclusive results. When this occurs, no information is returned, and the verification engineer is left with no clue on how to proceed.

More precisely, CDC synchronization verification focuses on safety properties (see Section 1.3.2 on page 10). Model checking a given safety property over a given hardware design boils down to solving the unreachability problem over a succinctly represented directed graph (such as a Moore machine), which is well-known to be PSPACE-complete [56]. Unfortunately, PSPACE-complete problems are intrinsically intractable in the sense that there does not exist a deterministic algorithm that is guaranteed to solve every instance in polynomial time.[1] That is, there will always be some problem instances that cause an exponential blow-up in the analysis, irrespective of how well the model checking algorithm is engineered. Even worse: since both NP and co-NP are most probably strictly contained in PSPACE, there can always be cases in which executions that prove or disprove a given property are of exponential length, independent of any representation. This is also known as the *state-space explosion* problem [21, 59].

4.1.2 Tackling the State-Space Explosion

In practice, the typical manual approach to overcome inconclusive results is to narrow the model to the logic that is relevant to each property. The verification is then focused

[1] Unless PSPACE=P which is commonly regarded as highly unlikely.

on just a small but relevant part of the design: the synchronizer boundary. However, this approach comes with the following issues:

- First, synchronization protocols are often split between modules, so the synchronizer boundaries are not contained in a small hierarchical module;
- Then, the verification engineer needs to have a very good understanding of the underlying design, which is not realistic for large RTL models;
- Also, 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.

This abstraction of the design space uses a technique called *localization reduction* which has been pioneered by Kurshan in the 1980s (and eventually published in 1994 [48]). It was also considered by Clarke et al. [20, 21] in the context of over-approximating abstractions defined through state-space partitioning. Using *counter-example guided abstraction refinements* (CEGAR), successive localization abstractions lead to an identification of the boundaries that are relevant to the property logic. The works by Andraus et al. [4, 5] propose a similar approach for data-paths in hardware designs. 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. To verify CDC protocols, Li and Kwok [52] described a flow using abstraction refinement along with synthesis to prove some properties, but the underlying techniques were not explained in detail. After running their flow, properties that are still inconclusive are *promoted* to the top-level, which require the user to use another methodology to proceed with the formal verification.

In the context of CDC verification, other approaches have been recently researched. Burns et al. [15] proposed a novel verification flow using xMAS models. While the verification of the synchronizer protocol is correct, the user needs to define its boundaries, which is not scalable. Similarly, Kebaili [43] proposed to push some workload from the functional to the structural checks: the main control signals of the synchronizer are structurally detected, and this information is used to create more precise assertions. The properties to be verified would only rely on these control signals (with a handshake-based protocol), hence avoiding the state-space explosion caused by the design complexity in the data path. However, it relies on the correct and exhaustive identification of a synchronizer "meta-model". To the best of our knowledge, the practical feasibility of this identification has not been proved nor implemented in industrial CDC tools.

4.1.3 Spurious Behaviors

The architecture of an SoC design includes many different operation modes (mission, test, scan, etc) which are controlled by configuration signals. Those signals are either primary inputs of the design, or set during an initialization phase by an externally-controlled FSM. We can distinguish three main categories of such signals:

- **Static signals**:
  Usually setup during an initialization phase, they remain constant during all executions (e.g., scan or test enables).
- **Quasi-static signals**:
  Their values change from time to time (e.g., for dynamically switching between power modes), but they remain mostly constant. In our context, these signals can be considered constant as we verify the design in a static mode.
- **Protocol signals**:
  Their behaviors are dependent on the values of other signals.

For instance, in the industrial CPU subsystem presented in Section 3.5.3, quasi-static configuration values are internally set by the software program which is expected to run on the CPU. We observe that verification engineers often do not constrain such configuration signals because it is not required outside of functional checks. Since the model checker has no information about the validity of a software program, it exhaustively explores all possible states and transitions. This often leads to a timeout, or in the best case spurious results. By pruning these spurious behaviors from the state machine, realistic results might actually be reached.

However, providing configuration information involves a complex and tedious work from the verification engineer. Even after providing it, a model checker may still not reach a conclusive result, due to the design complexity. Tackling the state-space explosion then requires to consider both the complexity issue and the missing assumptions on configuration signals. Our goal is then to automate both the boundary identification and the configuration debug, in order to reach realistic conclusive results.

4.2 State-Space Pruning

Instead of improving the performance of model checking engines, we prune irrelevant states from the machine in order to reduce the global state space. Two automated techniques are used: propagation of constant signal values, and localization abstraction.

4.2.1 Constant Propagation

SYNOPSYS CONFIDENTIAL

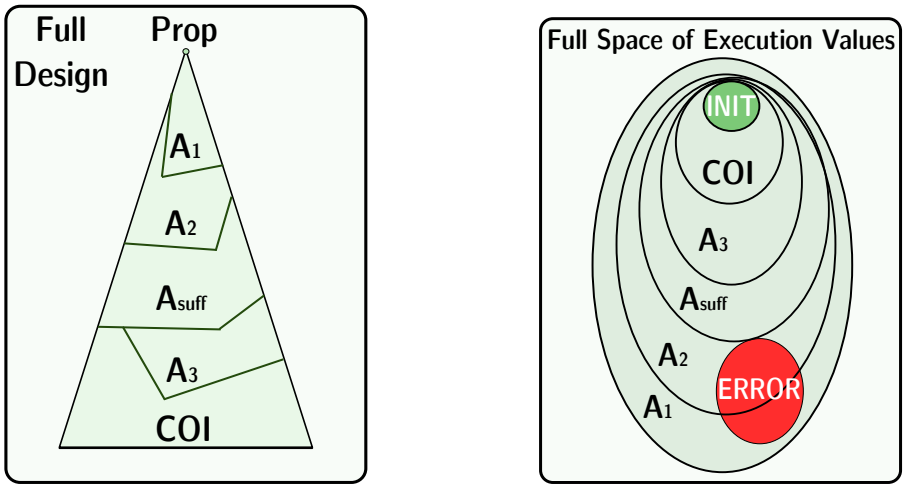
4.2.2 Localization Abstraction

One way to prune the state space is to operate via *localization reduction* [48]. First, we select signals (denoted *cut-points*) in the cone-of-influence of the property. Then, these signals are temporarily considered as primary inputs: their respective types are replaced by $\mathcal{T}_{\mathsf{in}}$. Localization reduction is pruning the structural design model $\mathcal{D}$, while increasing

the size of the reachable state space. The number of variables (as mentioned in 2.2.3) is greatly decreased, hence reducing the complexity of the state space exploration for the model checker. The new cone-of-influence of the property is called the *focus*, which represents an abstraction of the complete state space.

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 the correctness of synchronizing protocols should have a small focus that suffices to either prove the property or to reveal bugs.

Figure 4.1 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 boundary signals free, the set of reachable executions is enlarged (i.e., it represents an over-approximation). As a result, executions in which the property can fail, initially unreachable, may become reachable (A_1). The goal is to find the ideal abstraction which prunes away a sufficient part of the design ($\mathsf{A}_{\mathsf{suff}}$) without spuriously making any such execution reachable.



(a) Localization abstractions on $\mathcal{D}$

(b) Corresponding sets of reachable executions in $\mathcal{M}$

Figure 4.1: Various abstractions for a given design and property

Note that an abstraction A_2 is more precise than an abstraction A_1 if the cone-of-influence (with cut-points) of the property in A_2 is an extension of the one in A_1. Expanding the boundaries of the focus from A_1 to A_2 is called *abstraction refinement*. In this context, refining the abstraction consists in iteratively adding back some of the removed circuit. As a consequence, the reachable states-space and execution space is reduced (from A_1 to A_2) to a more realistic one.

In order to reach the optimal abstraction $\mathsf{A}_{\mathsf{suff}}$, one technique is to successively try different abstractions. First, we generate an *abstract counter-example* that is reachable with the last abstraction. Then, we analyze it to find the cut-points which are causing the failure. Since the over-approximation of these signals may be causing the failure, we

refine the abstraction in their cone-of-influence. This process is known as *counter-example guided abstraction refinement* (CEGAR) [20].

4.3 Counter-Example Debug

The analysis of an abstract counter-example gives which signals are relevant to the property failure, and should then be refined. While this process is time-consuming for an engineer, automatic structural heuristics provide valuable information which may be sufficient in our context. Along with the user guidance, two heuristics are considered: classification of data/control signals and analysis of signal values.

4.3.1 Signals Classification

SYNOPSYS CONFIDENTIAL

4.3.2 Failing Cause Analysis

SYNOPSYS CONFIDENTIAL

4.3.3 User Feedback Loop

While an automatic heuristic-based CEGAR eventually converges on a sufficient abstraction, a concise design insight may be crucial to lower the number of iterations. Indeed, the designer is the best source of information to know which part of the design has been implemented as the synchronization protocol. After identifying relevant cut-points in $\mathcal{J}_\omega$ and before refining them, the user is asked for his insight in a guided way. *SYNOPSYS CONFIDENTIAL*

Whenever an assumption is accepted, the corresponding signal is removed from the set $\mathcal{J}_\omega$. Rejecting all these assumptions implies that the fan-in of the signal contains relevant logic, and the signal is kept for automatic refinement. The focus and all subsequent abstraction will then be more precise, as they additionally comprise this relevant signal.

SYNOPSYS CONFIDENTIAL

4.4 Global Flow

Technically, our underlying framework is a CEGAR algorithm: a localization abstraction of the design is incrementally made more precise in a sequence of *refinement rounds* (see Section 4.2.2). In each round, the safety property is checked on the focus: if the property is satisfied, the algorithm terminates with result “proof”; if a counter-example is found,

structural heuristics identify relevant signals for the user to review. Either new assumptions are made, or the abstraction is refined, or the counter-example is *concretizable*, i.e. also valid for the full design, and the flow terminates with Result "fail". In contrast to fully automatic CEGAR approaches, the user here influences the refinement process. We therefore call our approach *User-aided CEGAR* (UCEGAR). Figure 4.2 gives an overview on the semi-automatic algorithm in the context of the overall methodology.

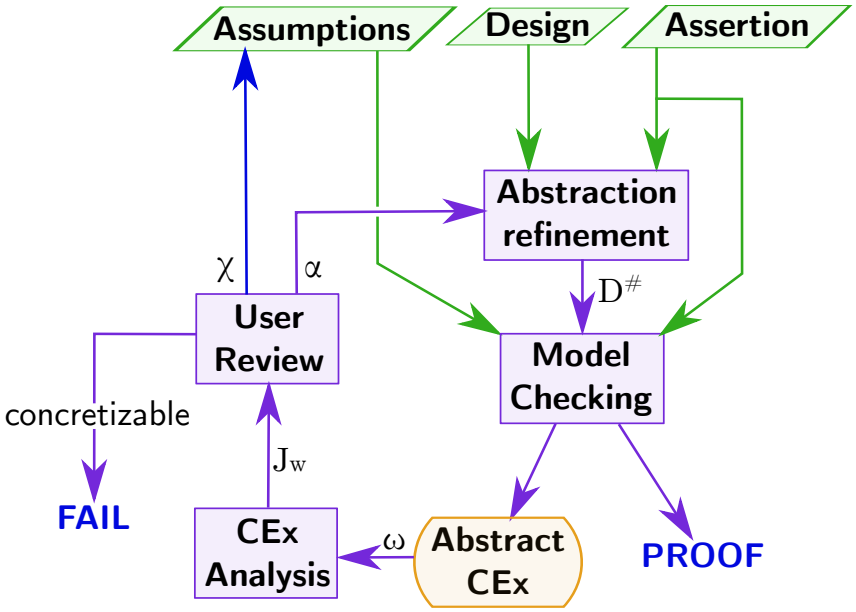


Figure 4.2: Global abstraction flow

4.4.1 Abstraction Refinement

Once an abstract counter-example is obtained and analyzed, the abstraction is refined to include more logic. From the set of relevant cut-points, we have to determine how much of their cone-of-influence should be added back. In other words, we have to find new cut-points in the fan-in of the previous ones. If too much of the COI is refined, then we will face a state-space explosion. If too few signals are refined, then we will again generate an abstract counter-example, hence leading to a new refinement round. Since each round calls a formal check, we need to optimize the number of iterations needed to reach $\mathsf{A}_{\mathsf{suff}}$.

SYNOPSYS CONFIDENTIAL

4.4.2 Core Algorithm

Algorithm ?? represents the global flow including the abstraction refinement and the counter-example debug. Any notation which is superscripted with # is considered on the design after abstraction. All other operations are considered on the original design $\mathcal{D}$.

The set Cut contains cut-points; Ref represents the signals to be refined; χ_{user} is the list of assumptions that the user originally defined. We assume that constant propagation is already processed, and signal classification returns the set $\mathcal{A}_{\mathsf{data},\phi}$.

Starting from the property output signal, Function refine runs the traversal defined in Algorithm ?? to identify the first cut-points (line 4). Function abstract creates a duplicate design from $\mathcal{D}$ where the type of these cut-points is replaced by $\mathcal{T}_{\mathsf{in}}$ (line 5). The resulting design model $\mathcal{D}^{\#}$ is our focus, on which we run the formal check (line 6). In the first round, only χ_{user} is considered in the list of assumptions χ_{all} that is given to the the model checker. As the abstraction is an over-approximation, a proof which is valid for $\mathcal{D}^{\#}$ will also be valid for $\mathcal{D}$, and is immediately reported (line 8).

SYNOPSYS CONFIDENTIAL

The failure is concretizable on the whole design when no signal is to be refined, and the user does not add any new assumption (line 24); a failure is then immediately reported. Otherwise, new assumptions χ_{add} are added to the set of user assumptions χ_{user}, and will be considered in the next rounds (for this property and the next ones).

SYNOPSYS CONFIDENTIAL

4.4.3 The Case of a Timeout

SYNOPSYS CONFIDENTIAL

4.4.4 Soundness, Completeness, Validity

The algorithm terminates if either the model checker reports a *proof* or if no new assumptions or signals 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 $\omega^{\#}$ only depends on signals within the focus, i.e., on signals that were not abstracted out or for which the user provided assumptions. Hence, $\omega^{\#}$ remains a valid counterexample for any greater set $\mathcal{D}' \supset \mathcal{D}^{\#}$, and in particular, on the full design $\mathcal{D}$.

In every non-terminating round, we either monotonically make the abstraction more precise or constrain the design behavior. Indeed, the traversal inside function refine starts from a cut-point, and eventually reaches either a primary input, or a cut-point candidate in its cone-of-influence. All traversed signals will then be added to the focus. 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., $\mathcal{D}^{\#} = \mathcal{D}$.

When manually adding assumptions one runs the risk of over-constraining the design's behavior, which can lead to vacuous (i.e., spurious) proofs. However, our methodology is designed to minimize the risk of over-constraining. The setup assumptions 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, cut-points are conservative in the sense that they lead to an over-approximation that preserves all safety properties.

4.5 Experimental Results

4.5.1 Experimental Setup

The following experiments compare the standard model checking with its integration into our global flow. We focus on three performance criteria:

- the number of assertions proved;
- the runtime to reach conclusive results;
- the workload on the user.

For each design, the structural CDC analysis is done by a specialized industrial tool, which returns assertions to prove the functionality of the synchronizers. As defined in Section 1.3.2 (page 10), two kind of assertions are verified: the stability of a data inside a CDC, and the coherency of a synchronized bus.

To verify them, the open source model checker ABC [14] is used with the engines *PDR* [27] and *BMC3* [10] in parallel. For each property, the runtime limit for timeout is set to 15 minutes. When the model checker reaches this timeout without returning any result, it is denoted *T/O*. We run the experiments on a workstation with 20 Intel Xeon E5-2640 v4 at 2.4GHz CPUs and 250GB of memory.

To analyze the influence of abstraction and user involvement on the results, we consider four different verification scenarios:

1. **Standard:** Model-checking each property on the full (non-abstracted) design.
2. **CEGAR:** A UCEGAR variant where the user rejects all assumptions. It can be seen as a reduction of UCEGAR to standard CEGAR.
3. **UCEGAR:** The full semi-automatic algorithm presented in Section 4.4 including automatic refinement and assumption inference from the user.
4. **Standard w/ assumptions:** A repeated run of the standard scenario, with all the accepted assumptions from the UCEGAR scheme.

4.5.2 Asynchronous FIFO

4.5.2.1 Design Presentation

This hardware design includes a FIFO similar to the one presented in Figure 1.11 on page 9. 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 128-bit counter, along with some non-deterministic control logic. Also, the source and destination clocks are enabled by sequential clock-gates, controlled by two independent non-deterministic 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.

4.5.2.2 Results

Using the industrial CDC tool, 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 Section 1.2.3 on page 6).

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 focus on the data stability property (see the results in Table 4.1).

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 minutes (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 makes 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 UCEGAR run, the enables of the clock-gates are justified and proposed for review to the user. Because having a non-deterministically enabled clock is not a realistic design behavior, we decide to accept the assumption that these signals should be constant ones. As a result, the stability property is solved in all 35 variations of the design within 10 seconds each (Column "UCEGAR"). 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 assumptions on the standard scheme, not all properties can be solved (Column "Standard w/ assumptions"). Also in this case, by comparing with column "UCEGAR", we notice that the runtime is always higher than when using both the inferred assumptions and CEGAR. This observation along with Figure 4.3 points out the importance of using both abstraction and involving the user in order to reach a conclusive result. Note that the workload on the user was small: only two clock controls had to be assumed to be constant ones.

FIFO depth	Data width	Standard	CEGAR	UCEGAR	Standard w/ assumptions
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

Table 4.1: Proof CPU runtime (in sec) on the asynchronous FIFO

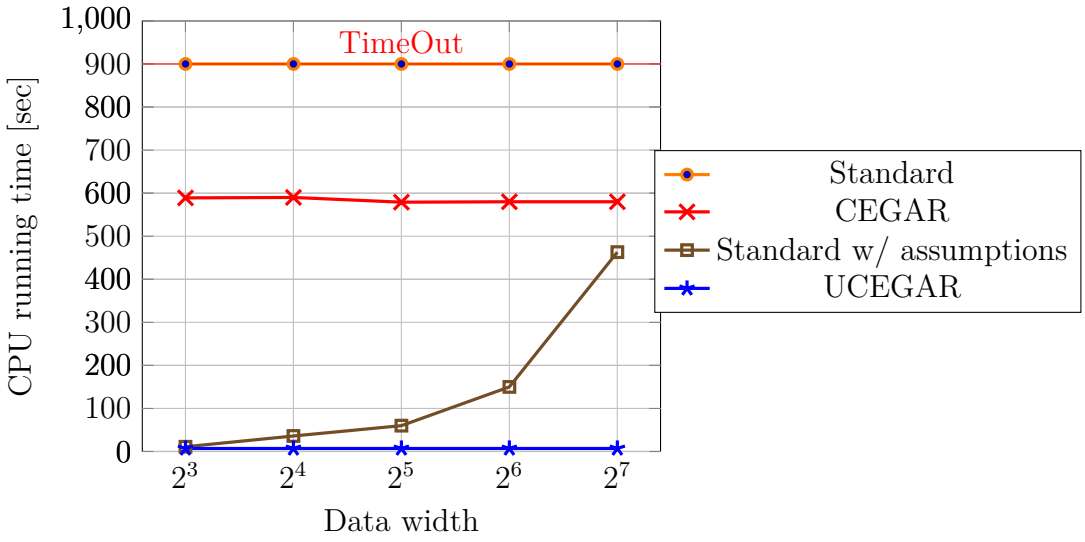


Figure 4.3: Performance comparison for FIFO depth 8

4.5.3 Industrial CPU Subsystem

4.5.3.1 Design Presentation

The second case study is the industrial hardware design which was already introduced in Section 3.5.3 (page 35). The setup of the clock and reset control module was made by the verification engineers. It is configured for a structural CDC analysis: all clock multiplexers are constrained so that one single clock propagates to each register. Also, static primary inputs were constrained to the value given in the design specification (subsystem configuration). However, many configuration signals (such as clock-enable signals, or internal resets which are controlled by the CPU software) are not constrained. As explained in Section 2.2.4 (page 24), all external resets are exercised to define a global initial state.

Since the design has a Globally Asynchronous Locally Synchronous (GALS) intent, CDC signals are always synchronized in the destination module. Figure 3.6 on page 36 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 Section 1.2.4 (page 8), with additional low power and performance optimizations. Only one communication is shown in Figure 3.6, 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.

Note that many synchronizers (mostly FIFO protocols) are split between modules of this subsystem. Consequently, the boundaries of each synchronizer are not trivial to define, and we cannot proceed in a module-by-module CDC analysis. Working at this hierarchy level is then particularly relevant for us.

4.5.3.2 Results of the UCEGAR Approach

An industrial CDC tool is used to structurally analyze the industrial design. 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. Each extracted property is verified in the same four schemes that were presented previously.

		Standard	CEGAR	UCEGAR 1^{st} run	**UCEGAR 2^{nd} run**	Standard w/ assumptions
Stability	# Proof	29	31	52	**78**	43
	# Fail	0	0	26	**0**	0
	# Inconclusive	49	47	0	**0**	35
	CPU time [min]	771	734	436	**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

Table 4.2: Results on the CDC stability and Gray-encoding properties

Table 4.2 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 UCEGAR run, 40 setup assumptions are automatically inferred and are all easily accepted. These include global interface enables (scan or test enables, internal configuration signals, ...), and also internal software resets and clock-gate enables which were missing in the original setup. It leads to 94 proved properties and provides counterexamples for the remaining 31. 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: on each interface, a global enable signal is toggling. 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 the designers, we decide to constrain these signals to a realistic behavior. Here, the worst case would be

to set them to '1' which would mean the CPU always transfers data. Twenty two Boolean assumptions are then added to enable the handshakes. We stress the fact that no deep design knowledge is needed during this process, and the assumptions represent a realistic design behavior.

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

Finally, the last column shows that having the proper assumptions is not sufficient to get proofs; the efficiency of UCEGAR indeed relies on the *combination* of automatic CEGAR and manual assumption 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 Section 4.1.2. 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.

4.5.3.3 Discussion on the Heuristics Efficiency

SYNOPSYS CONFIDENTIAL

4.6 Conclusion

UCEGAR is a complete formal verification flow for conclusively proving or disproving CDC assertions on industrial-scale SoC hardware designs. This work has been presented in the industrial conference SNUG [44], in the international conference VLSI-SoC [67] and was extended as a chapter for the book edition of the conference [68]. The key idea is to use counter-example-guided abstraction refinement (CEGAR) as the algorithmic backend, while the user influences the course of the algorithm based on information extracted from intermediate abstract counterexamples.

The efficiency of our approach had been demonstrated on an industrial 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 accept or reject assumptions that refer to single signals. 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 most of our predefined assumptions do not depend on a certain property but provide a general design setup. Therefore, they remain globally valid. This does not only speed-up the overall CDC verification time, when assumptions are reused while verifying multiple properties, it also helps further functional verification steps in the VLSI flow.

CHAPTER 5

Fixing Under-Constrained Designs

5.1 Context

5.1.1 Counter-Example Debug

While formal verification is precise, it takes a lot of effort and the work of an expert verification engineer to converge on a realistic result. After solving the potential state-space explosion problem, the two expected outputs are either a proof or a failure of the assertion. In the failing case, the designer has to debug the counter-example (CEx) of the failure, and decide if the failure is due to a design error, or a specification error (see Figure 1.13 on page 11). In the first case, the designer must fix the design. In the second case, the failure might be caused either by an ill-written assertion or an insufficiently specified environment of the design. In this chapter, we focus on this second aspect of specification debugging. More precisely, we are concerned by a failure caused by outside-the-specification input values. The verification engineer then has to figure out which assumption to add in order to make the inputs behavior more precise.

In large hardware designs, the specification complexity is increased to the point where it becomes unrealistic to manually create a perfect behavioral model of the primary inputs. When formal verification engineers face a false failure, identifying the missing assumption becomes a major challenge. Indeed, they do not usually have a sufficient design knowledge and time to review the counter-example from the incorrect failures. This root cause analysis stands as the main task of the verification engineers. It is closely linked to writing formal assumptions which avoid the incorrect input behavior. Typically, engineers perform *what-if analysis* [72], which consists in manually writing different assumptions in order to generate new CEx right away.

While this iterative process is likely to lead to a realistic environment model for a particular assertion, it is time-consuming and requires property-writing skills. The need for automatic root cause analysis and assumption generation arises.

5.1.2 Handshake Example

As an example, we consider the handshake protocol described in Section 2.1.1 on page 13. This protocol is fully formalized in the standard temporal *Property Specification Language* PSL[38] with the following safety properties:

1. assert **always** (! req && ! ack) → **next** ! ack
2. assume **always** (req && ! ack) → **next** req

3. assert **always** (req && ack) → **next** ack
4. assume **always** ack → **next** !req

Here, we consider a design under verification (DUV) which correctly implements the protocol on the receiving side (see Figure 5.1). Block *receiver* receives the data and the request, and answers with an acknowledge. Since we focus on the control protocol, we abstract out the data.

Figure 5.1: Handshake overview

All these properties use values at the current cycle in order to deduce a behavior at the next cycle. When this deduced behavior is an input, the property is an assumption that models the environment; in the example, properties (2) and (4) are assumptions. When this deduced behavior is an output, the property is an assertion that has to be verified; in the example, properties (1) and (3) are assertions. Using these four properties, a model checker will be able to verify the functionality of the handshake.

Figure 2.3 on page 15 displays the state diagram of Block *receiver*. In the absence of the two assume properties of the protocol, all transitions are possible. If the two assumptions are added, the brown dotted transitions are removed. For instance, it becomes impossible to directly reach S3 from S1.

If assumptions 2 and 4 are given to a formal engine, then assertions 1 and 3 are proven. If only assumption 4 is given, then assertion 1 fails after reaching state S2 from S3.

Among all counter-examples that the formal engine can generate, three are shown in Figure 5.2. Note that this is theoretical, as current formal engines only report one counter-example (usually the one with the smallest sequential depth).

After debugging these counter-examples, the designer will infer that the *request* signal is not following the protocol specification. The verification engineer will then have to figure out how to write the missing assumption.

5.1.3 Generating Properties from a Design

The automatic generation of missing assumptions was investigated with the objective of debugging specifications to make them *realizable* [1]. Informally, "a specification is unrealizable if there is no implementation that can produce outputs to satisfy the specification given all possible inputs that can be generated by the environment" [53]. We share a common aim with the authors who address the case when specifications are unrealizable due to an under-constrained environment, and want to generate additional environment assumptions [18, 33, 53]. The above authors represent the specifications as an automaton,

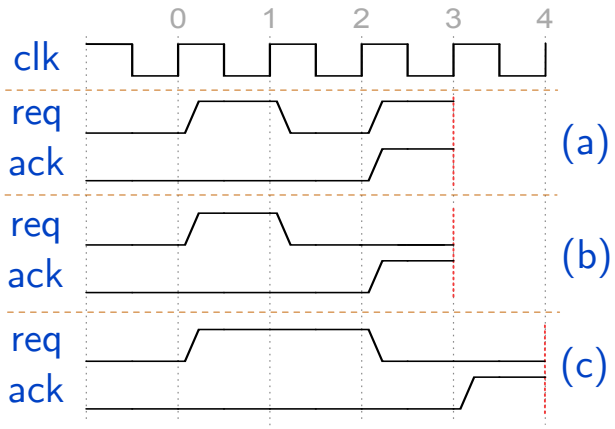


Figure 5.2: Counter-examples for the handshake with a missing assumption

and apply the two-players game procedure by which the automaton built for the environments elaborates a strategy (sequence of inputs) to prevent the system from satisfying the specifications, a procedure that is known to time-out on large benchmarks (its time complexity is cubic). As a matter of fact, no execution time is provided on the practical results of the above references. We differ in the fact that we work on an already designed system and a failing assertion, while the above consider no pre-existing design and a set of properties.

Property mining techniques have been successful in generating formal properties from a model, for verification purposes. It may be used to help identify non-trivial corner-cases and improve the coverage for regression tests. In the GoldMine tool [35, 82] a large set of patterns are simulated on a design, which generates thousands of traces for Boolean signals. Property mining tools extract from the traces thousands of assertion candidates which are model checked on the design to filter out spurious and failing assertions. Static analysis helps improve the percentage of mined assertions that pass the model checking step by computing the cones of influence.

Mining of properties built over arithmetic variables is reported in [8, 26]. The method repeatedly invokes the *Daikon* tool [79] that returns relations over arithmetic variables at a time step. Then the authors build temporal properties according to predefined patterns based on the *implication*, *next*, *until* and *alternating* operators. The complexity is cubic in the number of variables, and proportional to the length of the trace and the number of considered implication antecedents, which limits the use of the tool to a limited set of traced variables.

The work of Keng et al. [45, 46], like ours, addresses the hypothesis that an assertion fails in a correct design due to missing assumptions, and aims at automatically suggesting some assumptions to the designer as an aid to elaborate the missing one. Their flow is based on two successive iterations: the first one generates multiple counter-examples and multiple corrective assumptions on the design inputs. The second one prunes the generated assumptions to reduce the number of those effectively returned to the user. The technology underlying Keng's method is entirely built over a SAT engine for the

generation and filtering of candidate assumptions, at the cost of processing time. The generated assumptions are unit cycle, which cannot express *temporal causality between distinct signals*, a must for assumptions on protocols.

Our main contribution is an efficient algorithm which automatically infers missing assumptions in the context of a property failure. First, we generate multiple distinct counter-examples which are guaranteed not to be just delayed one from the other by some cycles (in contrast to the results discussed in [46]). Then, we extract common root causes of the failure from the set of counter-examples, using mining techniques combined with a structural analysis of the netlist. Finally, we generate realistic temporal assumptions for the user to review.

5.2 Generation of Counter-Examples

Using only one counter-example, it is not trivial to debug the reason of the failure. Our objective is to generate many counter-examples, so that it is easier to single out under-constrained inputs of the design.

5.2.1 Blocking Relevant Values

As already introduced in Section 3.3.2 on page 32, we will use a blocking condition in order to generate multiple counter-examples. The main issue here is to find a blocking condition which generates a CEx with a different root cause. If the blocking condition is removing too few input behaviors, then the new CEx will be very close to the previous one (e.g., CEx will be delayed by one idle cycle). If the blocking condition is removing too many input behaviors, no new CEx will be possible and the assertion will be spuriously proved (e.g., if the blocking condition is stopping the clock).

The blocking conditions are generated in PSL, using *Sequential Extended Regular Expressions* (SEREs). Similarly to Section 3.3.2, function Lit_ω is used to represent the LTL literal corresponding to the valuation of a signal at a time point.

The simple blocking condition χ_ω, based on the CEx ω, precludes the sequence of configuration values $\langle c_0, \ldots, c_{|w|-1} \rangle$ on all signals. Since the state and output values are completely determined by the input values (using δ and λ), this blocking condition is reduced to preclude the input values only:

$$\chi_\omega = ! \left\{ L_0 \,;\, L_1 \,;\, \ldots \,;\, L_{|\omega|-1)} \right\}$$
$$\text{with } L_t = \bigwedge_{i=1}^{|\mathcal{A}_{\mathsf{in}}|} \mathrm{Lit}_\omega(\mathcal{A}^i_{\mathsf{in}}, t)$$

With this approach, the three blocking conditions corresponding to the three CEx on Figure 5.2 are:

(a) assume ! {! req; req; ! req; req}

(b) assume ! {! req; req; ! req; ! req}

(c) assume ! {! req; req; req; ! req; ! req}

The blocking condition can be very complex if $|\omega|$ and $|\mathcal{A}_{\mathsf{in}}|$ are high. Also, only the behavior leading to the failure should be disabled, so that a different path is exercised in the next round. In order to improve the formal engine performance and the diversity of the causes of failure among CEx, some abstraction is applied. For this purpose, we use the justification function $J_{\omega,\phi}$ and the associated literal function $\widehat{\text{Lit}}_\omega$, as already defined in Section 3.3.1 on page 31. The resulting blocking condition is simpler because it only constrains relevant input values:

$$\chi_\omega = ! \left\{ L_0 \,;\, L_1 \,;\, \ldots \,;\, L_{|\omega|-1)} \right\}$$
$$\text{with } L_t = \bigwedge_{i=1}^{|\mathcal{A}_{\mathsf{in}}|} \widehat{\text{Lit}}_\omega(\mathcal{A}^i_{\mathsf{in}}, t)$$

For instance, in CEx (a) and (b) of Figure 5.2, the value of req at cycle 3 is irrelevant to the property, so both CEx exhibit the exact same failure. The corresponding blocking condition will not block this value, hence avoiding to generate CEx (b):

$$\text{assume ! \{! req; req; ! req; true\}}$$

5.2.2 Blocking Stuttering

An execution ω is stuttering at time $t \in \mathbb{N}$ if it exhibits repeating configurations: $c_t = c_{t+1}$. In the following example, ω is stuttering from time 2 to 4, and ω' is stuttering from time 6 to 7:

time	0	1	2	3	4	5	6	7	8	9
ω	s_0	s_1	$\lvert\mathbf{s_2}$	$\mathbf{s_2}$	$\mathbf{s_2}\rvert$	s_3	s_4	s_5	$\lvert\mathbf{s_6}\rvert$	s_7
ω'	s_0	s_1	$\lvert\mathbf{s_2}\rvert$	s_3	s_4	s_5	$\lvert\mathbf{s_6}$	$\mathbf{s_6}\rvert$	s_7	

An execution starts stuttering at time t_1 if:

$$(c_{t_1} = c_{t_1+1}) \wedge (\forall t < t_1, c_t \neq c_{t+1})$$

Let ω be an execution that starts stuttering at time t_1. There exists an execution ω' which is not stuttering at t_1, such that:

- $\omega'_{0..t_1} = \omega_{0..t_1}$
- $\exists t_2 > t_1, \begin{cases} \forall c \in \omega_{t_1..t_2-1}, c'_{t_1} = c \\ \omega'_{t_1+1..} = \omega_{t_2..} \end{cases}$

We say that ω' is equivalent to ω after removing the stuttering at t_1. By induction, we generate a sequence of equivalent executions whose stutterings are removed one by

one. Since the counter-examples are of finite length, an equivalent execution is eventually generated without any stuttering. Thus, two executions ω and ω' are said to be equivalent if they can be reduced to the same non-stuttering equivalent execution.

During the generation of counter-examples, we want to avoid generating many equivalent CEx because they do not reveal a new cause of failure. However, gathering a few equivalent CEx gives significant information, as it implies that the stutter is irrelevant for the failure. Let *maxTD* be the maximum temporal depth of the assumptions we plan to generate. For instance, in the case of the handshake, all assumptions are 2-cycles deep. Whenever *maxTD* CEx have been generated with the same stutter, we create a blocking condition to preclude further executions with the same stutter. Assuming that the stutter is detected at time point k, the blocking condition becomes[1]:

$$\chi_\omega = ! \left\{ L_0 \,;\, \ldots \,;\, L_k[+] \,;\, \ldots \,;\, L_{|\omega|-1)} \right\}$$
$$\text{with } L_t = \bigwedge_{i=1}^{|\mathcal{A}_{\mathsf{in}}|} \widehat{\mathrm{Lit}}_\omega(\mathcal{A}^i_{\mathsf{in}}, t)$$

For instance in Figure 5.2, cycles 1 and 2 of CEx (c) are repeating cycle 1 of CEx (a). The corresponding blocking condition ends up to become:

$$\text{assume } ! \{ ! \, \text{req}; \text{req}[+]; ! \, \text{req}; \text{true} \}$$

To improve the performance of the blocking condition in the formal check, a custom property monitor is created. Each cycle to block is represented with a monitor state. When the final state is reached, it means that the sequence of inputs has been identified. Then, the monitor outputs false and the assumption is violated.

Figure 5.3 shows the example of a monitor blocking the CEx (a) of Figure 5.2. Here, the states of the monitor are encoded in one-hot. At time t, when the inputs have the expected value, $\mathsf{AND_t}$ is true (here, there is only one input: req). If the input sequence has been identified up to time t, then $\mathsf{SEQ_t}$ is true. Whenever both $\mathsf{SEQ_{t-1}}$ and $\mathsf{AND_t}$ are true, then we continue to the next state. To consider the stuttering at time t, if both $\mathsf{SEQ_t}$ and $\mathsf{AND_t}$ are true, then we stay in this same stuttering state.

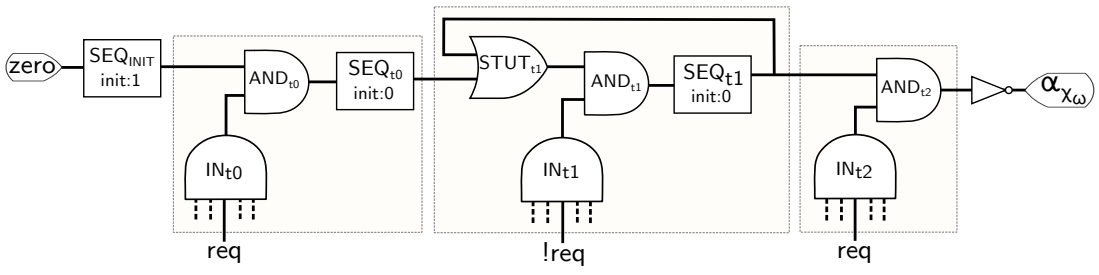


Figure 5.3: Blocking condition monitor

[1]In PSL, $\alpha[+]$ denotes one or more repetitions of α.

5.2.3 Algorithm

The flow which generates multiple CEx is presented in Algorithm 1. During each round, the model checking engine checks the assertion with a set of assumptions χ_{all}. If a CEx ω is found, the blocking condition χ_ω is created on the justified CEx. Then, the next round starts with an updated set of assumptions. The algorithm terminates either when assertion ϕ is proved on $\mathcal{M}$, or when the number of CEx reaches a predefined limit *maxLimit*. Then, we obtain a set of CEx $\mathcal{W}$.

```
1  χ_all ← χ_user // Initial assumptions
2  W ← ∅ // Set of CEx
3  while (size(W) < maxLimit) do
4  |   ω ← check(D, φ, χ_all)
5  |   if (ω = ∅) then
6  |   |   return W
7  |   else
8  |   |   ω̂ ← justify(ω, φ)
9  |   |   W ← W ∪ {ω̂}
10 |   |   χ_ω ← disable(ω̂, W)
11 |   |   χ_all ← χ_all ∧ χ_ω
12 |   end
13 end
14 return W
```

Algorithm 1: Generate multiple CEx for assertion ϕ on model $\mathcal{M}$ with user assumptions χ_{user}

Function $\mathsf{check}(\mathcal{M}, \phi, \chi_{\mathsf{all}})$ calls a formal engine on the design model $\mathcal{M}$ to verify an assertion ϕ with given assumptions χ_{all}. It returns a CEx if the assertion fails.

Function $\mathsf{justify}(\omega)$ extends $J_{\omega,\phi}$ on all signals and time points of an execution ω. It returns a justified execution, as defined in Section 3.3.1 (page 31).

Function $\mathsf{disable}(\omega, \mathcal{W})$ returns the blocking condition corresponding to a CEx ω, after considering stuttering executions in $\mathcal{W}$.

Note that when a proof is obtained (lines 5-6 of Algorithm 1), it does not mean that we reached a corrected model, it only means that no more CEx can be generated.

5.3 Extraction of Assumptions

We assume that all the generated CEx reveal an incorrect input behavior, which should be constrained. We use property mining in order to identify this common root cause of the failures. Then, we generate assumptions avoiding each mined behavior. Finally, we apply several criteria to find a realistic assumption leading to a proof of the original assertion.

Note that, in this section, we work on signals which are in the same clock domain. Indeed, all the protocol assumptions that we are aware of in modern design only refer to one single clock. Thus, the missing assumptions we are interested in contain signals from one single domain.

5.3.1 Property Mining

If a missing assumption χ is the cause of a failing execution ω, then: $\omega \not\models \chi$ (which is equivalent to $\omega \models \neg\chi$). By mining all the properties that are satisfied by the execution, we know that the negation of one of them will be the missing assumption we are looking for.

Assumptions are mostly created by verification engineers following standard templates $\mathrm{T_C}$, which map a list of signals to a formal property.

$$\mathcal{T}_\mathcal{C} \subset \{\mathrm{T_C} : \mathcal{A}^* \to \mathcal{P}\}$$

Here, we limit our scope to the set of templates $\mathcal{T}_\mathcal{C}$ described in Column "Assumption Template" of Table 5.1. This scope is motivated by the handshake specification, and corresponds to the common templates of property mining.

Assumption Template $\mathrm{T_C}$		Input	Order
$\lambda\alpha_1.$	**always** α_1	α_1	$-$
$\lambda\alpha_1\alpha_2.$	**always** $\alpha_1 \rightarrow \alpha_2$	α_2	$\alpha_1 \prec \alpha_2$
$\lambda\alpha_1\alpha_2.$	**always** $\alpha_1 \rightarrow$ **next** α_2	α_2	$-$
$\lambda\alpha_1\alpha_2\alpha_3.$	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow \alpha_3$	α_3	$\alpha_1 \prec \alpha_2 \prec \alpha_3$
$\lambda\alpha_1\alpha_2\alpha_3.$	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow$ **next** α_3	α_3	$\alpha_1 \prec \alpha_2$
$\lambda\alpha_1\alpha_2\alpha_3.$	**always** $(\alpha_1 \,\&\&$ **next** $\alpha_2) \rightarrow$ **next** α_3	α_3	$\alpha_2 \prec \alpha_3$

Table 5.1: Assumption writing templates

For brevity, Table 5.1 only shows the positive occurrences of the parameters. The table we use in practice holds all combinations. For instance, the second template of the table can be instantiated as:

- $\lambda\alpha_1\alpha_2.$ **always** $\alpha_1 \rightarrow \alpha_2$
- $\lambda\alpha_1\alpha_2.$ **always** $\alpha_1 \rightarrow \neg\alpha_2$
- $\lambda\alpha_1\alpha_2.$ **always** $\neg\alpha_1 \rightarrow \alpha_2$
- $\lambda\alpha_1\alpha_2.$ **always** $\neg\alpha_1 \rightarrow \neg\alpha_2$

In contrast to the usual property mining which exhibits circuit properties that the user is looking for, the kind of mining we are performing on CEx exhibits behaviors that should be forbidden. As a consequence, we wish to find in the CEx some behaviors to be

negated by the missing assumption. In other words, the type of property we mine are the negations of the templates of Table 5.1.

We therefore give a 1-to-1 mapping between each assumption template $\mathrm{T_C}$ of Table 5.1 and a mining template $\mathrm{T_M}$ under the format given in Table 5.2, such that:

$$\forall \mathrm{T_M} \in \mathcal{T}_{\mathcal{M}},\ \exists \mathrm{T_C} \in \mathcal{T}_{\mathcal{C}},\ \neg \mathrm{T_M} \implies \mathrm{T_C}$$

Mining Template $\mathrm{T_M}$	
$\lambda\alpha_1.$	**eventually** $\neg\alpha_1$
$\lambda\alpha_1\alpha_2.$	**eventually** α_1 && $\neg\alpha_2$
$\lambda\alpha_1\alpha_2.$	**eventually** α_1 && **next** $\neg\alpha_2$
$\lambda\alpha_1\alpha_2\alpha_3.$	**eventually** $(\alpha_1$ && $\alpha_2)$ && $\neg\alpha_3$
$\lambda\alpha_1\alpha_2\alpha_3.$	**eventually** $(\alpha_1$ && $\alpha_2)$ && **next** $\neg\alpha_3$
$\lambda\alpha_1\alpha_2\alpha_3.$	**eventually** $(\alpha_1$ && **next** $\alpha_2)$ && **next** $\neg\alpha_3$

Table 5.2: Property mining templates

For instance, all counter-examples of Figure 5.2 from Section 5.1.2 exhibit:

$$\omega \vDash \textbf{eventually}\,(!\,\mathrm{ack}\,\&\&\,!\,\mathrm{req}\,\&\&\,\textbf{next}\,(\mathrm{req}))$$

and we generate from it the corresponding assumption:

$$\mathrm{T_C}(\mathrm{req},\mathrm{ack}) = \textbf{always}\,(!\,\mathrm{ack}\,\&\&\,!\,\mathrm{req}) \rightarrow \textbf{next}\,(!\,\mathrm{req})$$

Since we are checking for behaviors relevant to the assertion, our property mining runs on the list of justified counter-examples $\mathcal{W}$. After mining for a behavior following template $\mathrm{T_M} \in \mathcal{T}_{\mathcal{M}}$ on CEx $\omega \in \mathcal{W}$, we obtain a list $\mathcal{P}^{\omega}_{\mathrm{T_M}}$ of mined properties:

$$\mathcal{P}^{\omega}_{\mathrm{T_M}} = \{\ \mathrm{T_M}(\alpha_1,\alpha_2,\dots) \mid \exists \langle \alpha_1,\alpha_2,\dots\rangle \in (\mathcal{A}_{\mathsf{in}} \cup \mathcal{A}_{\mathsf{out}})^*, \widehat{\omega} \vDash \mathrm{T_M}(\alpha_1,\alpha_2,\dots)\ \}$$

From this set of mined properties, we infer the set of potentially missing assumptions $\mathcal{P}^{\omega}_{\mathrm{T_C}}$ using the 1-to-1 correspondence from $\mathrm{T_M}$ to $\mathrm{T_C}$.

5.3.2 Filtering Heuristics

Using only property mining, the number of generated assumptions is huge. Most of them are, of course, not actual missing assumptions. We use multiple lightweight criteria to filter out irrelevant assumptions.

5.3.2.1 Input Condition

In our context, we are aiming at modeling an external module connected to the inputs and outputs of our design. We are thus searching for missing assumptions on inputs, and not on outputs. If a generated assumption restricts the behavior of an output, then it is discarded. For instance, "$\alpha_1 \rightarrow$ **next** α_2" is only valid if α_2 is an input. Using the reasoning explained in Section 5.1.2, we specify, for each template, which parameter needs to be an input (see Column 2 of Table 5.1). In effect, for the restricted list of Table 5.1, the last parameter of a template function should always be an input.

5.3.2.2 Uniqueness

Distinct instantiations of templates may be semantically identical (for instance $\alpha_1 \rightarrow !\,\alpha_2$ and $\alpha_2 \rightarrow !\,\alpha_1$). This happens when two Boolean propositions are referring to the same time point. To avoid redundancy, we attach a strict order condition to each template (Column 3 in Table 5.1). Every generated assumption can then be written in a unique way.

For two signals α_1 and α_2, the strict order "$\alpha_1 \prec \alpha_2$" is defined as complying with one of the following (mutually exclusive) conditions, where $ID(\alpha_1)$ defines a unique identifier of the signal and identifiers are lexically ordered:

- $(\alpha_1 \in \mathcal{A}_{\mathsf{out}}) \wedge (\alpha_2 \in \mathcal{A}_{\mathsf{in}})$
- $(\alpha_1 \in \mathcal{A}_{\mathsf{in}}) \wedge (\alpha_2 \in \mathcal{A}_{\mathsf{in}}) \wedge (ID(\alpha_1) < ID(\alpha_2))$
- $(\alpha_1 \in \mathcal{A}_{\mathsf{out}}) \wedge (\alpha_2 \in \mathcal{A}_{\mathsf{out}}) \wedge (ID(\alpha_1) < ID(\alpha_2))$

5.3.2.3 Common Cause

Since we assume that all the generated CEx show the same erroneous behavior, the root cause behavior can be found in every CEx. Thus, we only keep the assumptions that are common to all the CEx:

$$\mathcal{X} = \bigcap_{\omega \in \mathcal{W}} \left(\bigcup_{\mathrm{T_C} \in \mathcal{T}_\mathcal{C}} \mathcal{P}^{\omega}_{\mathrm{T_C}} \right)$$

If we assume multiple missing assumptions, then this criterion can be easily weakened by replacing the intersection by a threshold of occurrences, e.g., properties occurring in at least 50% of the generated CEx.

5.3.2.4 Implication

An assumption χ implies another assumption χ' when: $\chi' \subseteq \chi$. We deduce from this implication:

$$\begin{aligned}
&\mathcal{M}|\chi \vDash \chi' \\
&\forall \phi \in \mathcal{P}, (\mathcal{M}|\chi' \vDash \phi) \rightarrow (\mathcal{M}|\chi \vDash \phi) \\
&\forall \phi \in \mathcal{P}, (\mathcal{M}|\chi \nvDash \phi) \rightarrow (\mathcal{M}|\chi' \nvDash \phi)
\end{aligned}$$

For instance, $\chi' \subseteq \chi$ with:

$$\chi = \textbf{always}\ \alpha_1 \rightarrow \textbf{next}\ \alpha_3$$
$$\chi' = \textbf{always}\,(\alpha_1\ \&\&\ \alpha_2) \rightarrow \textbf{next}\ \alpha_3$$

Whenever such kind of assumptions are generated, they are stored in an *implication set*:

$$\text{Impl} = \{\langle \chi, \chi' \rangle \,|\, \chi' \subseteq \chi\}$$

As we know the structure of all generated assumptions, determining the implications does not need a formal check but only a lightweight syntactical one. The exhaustive list of template implications is available in Appendix E.

5.3.2.5 Triviality

It may happen that the user assumptions imply a generated assumption: $\chi \subseteq \chi_{\mathsf{user}}$. Then, the generated assumption is considered trivially true and is discarded from $\mathcal{X}$. The same deduction applies to all assumptions $\chi' \subseteq \chi$ which are identified with the set Impl. Conversely, if an assumption χ is not considered trivial, then all assumptions χ' which imply χ are also not trivial. Note that we cannot use a SAT-solver as the assumptions are temporal. We use model checking bounded with the highest sequential depth of the assumptions *maxTD*.

5.3.2.6 Consistency

Each generated assumption is checked for consistency with the user-defined assumptions, and the inconsistent ones are discarded from the set $\mathcal{X}$. An inconsistent assumption is defined by: $(\chi_{\mathsf{user}} \wedge \chi) \equiv \text{false}$.

5.3.2.7 Vacuity

A generated assumption may over-constrain the model, which leads to a vacuous proof of the assertion ϕ that the user wants to check. For that purpose, we verify that all signals directly involved in the assertion can toggle, using the assertion:

$$\text{assert}\ \textbf{eventually}\,(\alpha = \textbf{next}\ \neg\alpha)$$

Also, if the assertion is formulated with an implication, we check if the left-hand side α_1 is not always false, and if the right-hand side α_2 is not always true:

$$\phi = \text{assert}\ \textbf{always}\ \alpha_1 \rightarrow\ \alpha_2$$
$$\text{assert}\ \textbf{eventually}\ \alpha_1$$
$$\text{assert}\ \textbf{eventually}\ !\,\alpha_2$$

Assumptions complying with the *vacuity* criterion are discarded from the set $\mathcal{X}$. The same action holds for all generated assumptions which imply a vacuous assumption. Conversely, if an assumption is not vacuous, then all implied assumptions are also not vacuous.

5.3.2.8 Assertion Check

After applying the above filters on $\mathcal{X}$, the assertion ϕ is checked on the model with each remaining assumption χ.

$$\mathcal{M}|\chi_{\wedge}\chi_{\mathsf{user}} \vDash \phi$$

This step occurs at the end of all filters because it is expected to be the most expensive. If the assertion is proved, then assumption χ stands out as a very good candidate to be the missing one, and should be reported to the user with a high priority. The same conclusion holds for all assumptions which imply χ.

To order the assumptions by priority, they are labeled with a quality metric μ. If the assertion is non-vacuously proved, then this assumption gets the highest value. If the assertion is still failing, the assumption's μ is the relative augmentation of the sequential depth of the new CEx ω_1 (obtained when $\chi_{\wedge}\chi_{\mathsf{user}}$ is considered) with respect to the original CEx ω_0 (when only χ_{user} is considered).

$$\begin{array}{rccl} \mu: & \mathcal{X} & \longrightarrow & [0,1] \\ & \chi & \mapsto & \begin{cases} 1 & \text{if } \mathcal{M}|\chi_{\wedge}\chi_{\mathsf{user}} \vDash \phi \\ \frac{|\omega_1|-|\omega_0|}{|\omega_1|} & \text{otherwise} \end{cases} \end{array}$$

It is possible that none of the generated assumptions leads to a proof. Several explanations exist: either our list of templates is not exhaustive, or a second assumption is missing, or the design exhibits a bug. In the last case, the assertion should always fail. However, if the depth of the new failure is greater than the original one, then it was more difficult for the model checker to find it. Thus, we infer that this assumption is somehow relevant – at least more than if the two CEx were identical – and can be helpful in debugging the design.

After ranking all assumptions according to their metric μ, they are reported to the user for review. The implication map is also used in the reporting, as an aid to the user. For usability purposes, all assumptions with a quality metric of 0 (or lower than a user-defined value) are discarded from the set $\mathcal{X}$. In case the user discards all proposed assumptions, because they don't fit the specification, he may deduce that there is a real design bug. The list of justified counter-examples and the mined properties – revealing a potential root cause of the failure – will then contribute greatly to debugging.

5.4 Experimental Results

The overall flow combines both the CEx generation (Rectangle 1) and the assumptions extraction (Rectangle 2), as shown in Figure 5.4. From the user perspective, a list of CEx is created and available on-the-fly for debugging. At the end of the flow, a succinct list of potentially missing assumptions is available for review.

In the following experiments, we consider designs for which an assertion is originally proved. We remove one assumption from the formal specification, which leads to a failure of the assertion. The flow is then evaluated based on three quality criteria:

- the missing assumption is proposed to the user;
- all the proposed assumptions are realistic;
- the algorithms performance is scalable.

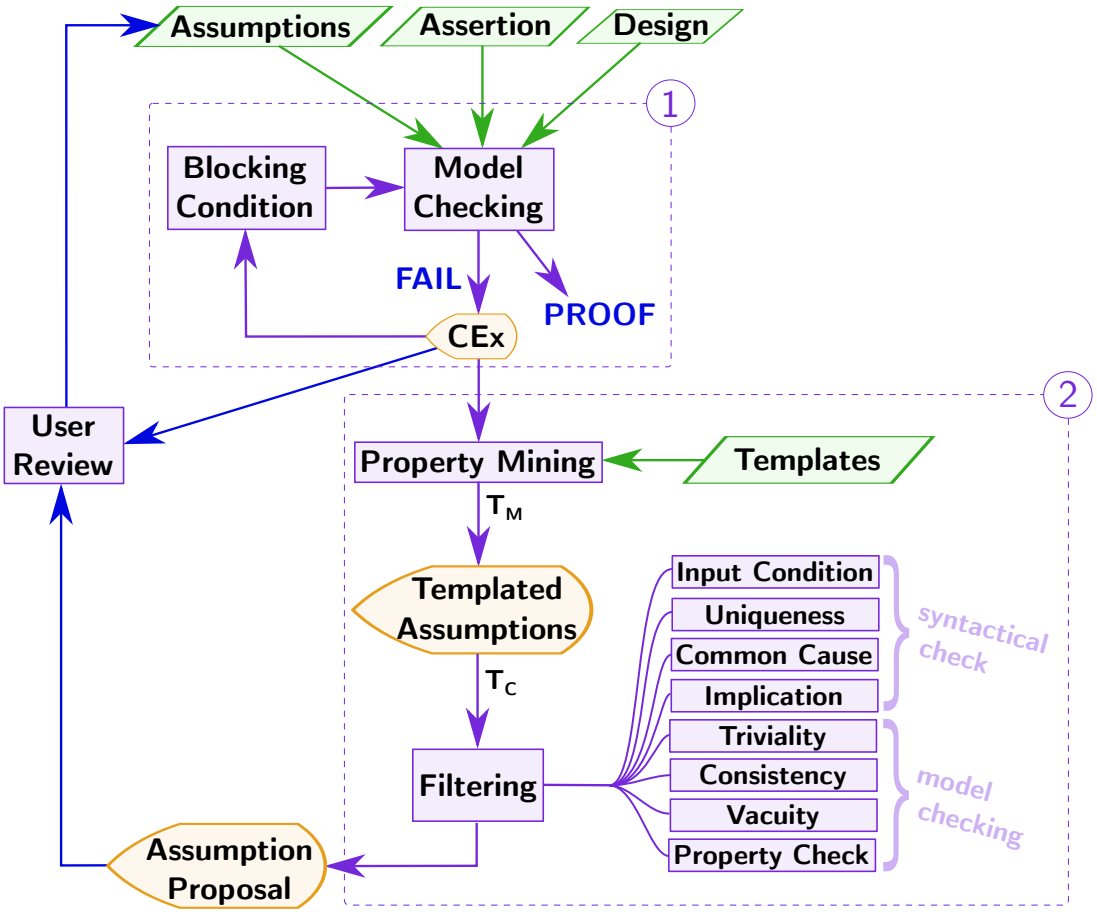


Figure 5.4: Assumption generation flow

5.4.1 Handshake

As a proof of concept, we run our flow on the handshake described in Section 5.1.2. We use the model checker ABC [14] with the engines *PDR* [27] and *BMC3* [10] in parallel. The CPU is an Intel Xeon E5-2640 v4 at 2.40GHz with 250GB of memory.

Algorithm 1 generates 10 counter-examples in 2 seconds. The first two are CEx (a) and (c) from Figure 5.2. 77 assumptions are then extracted from these 10 CEx, and filtered with the syntactical criteria: *input condition*, *uniqueness*, *common cause*. The

resulting assumptions are presented in Column "Instantiated assumption" of Table 5.3. Column "Status" represents the final result of the assumption, which is either discarded by a filtering criterion, or accepted as a proof.

	Instantiated assumption	Status
1.	req	Discarded by: Consistency
2.	! req	Discarded by: Vacuity
3.	! ack → req	Discarded by: Consistency
4.	! ack → ! req	Discarded by: Vacuity
5.	! ack → **next** req	Proof μ=1
6.	! ack → **next** ! req	Discarded by: Vacuity
7.	req → **next** req	Discarded by: Consistency
8.	! req → **next** ! req	Discarded by: Vacuity
9.	(! ack && req) → **next** req	Proof μ=1
10.	(! ack && ! req) → **next** ! req	Discarded by: Vacuity
11.	(req && **next** ! ack) → **next** req	Proof μ=1
12.	(! req && **next** ! ack) → **next** ! req	Discarded by: Vacuity
13.	(! ack && **next** ! ack) → **next** req	Proof μ=1
14.	(! ack && **next** ! ack) → **next** ! req	Discarded by: Vacuity

Table 5.3: Assumptions extracted from the handshake CEx

After automatically removing inconsistent assumptions and the ones leading to a spurious proof, only four assumptions are left for the user to review. Note that the full flow (CEx generation, property mining and filtering) completes within 8 seconds.

Actually, all four assumptions lead to a correct handshake protocol, but some are more restrictive than others. Indeed, the implication set shows that assumption 5 implies assumptions 9 and 13. We notice that assumptions 5 and 13 force the protocol to never be idle: as soon as the acknowledge is falling, a new request rises. Assumption 11 describes a behavior similar to Assumption 9, but the control logic on the sender side is combinational rather than sequential. As they are all realistic, we cannot automatically infer which of the four assumptions is the missing one. Only a verification engineer, using a design specification, can identify Assumption 9 as the correct one.

5.4.2 Generalized Buffer

The generalized buffer (GenBuf) is an arbiter which queues data between two senders and two receivers (see Figure 5.5). The design specification was created by IBM [34] and it has been exercised multiple times in the formal verification literature [12, 63]. While its specification seems simple, its formal complexity is greater than the industrial AMBA bus from ARM that will be considered in the next subsection.

The GenBuf contains an internal FIFO of depth 4. Two internal signals, fifo_full and fifo_empty, respectively represent the full and empty states of the FIFO. All senders are

equivalent, as are the receivers. The interface between each sender and the buffer uses a handshake protocol (these signals are prefixed with "src_"). The interface between each sender and the buffer also uses a handshake protocol (these signals are prefixed with "dst_").

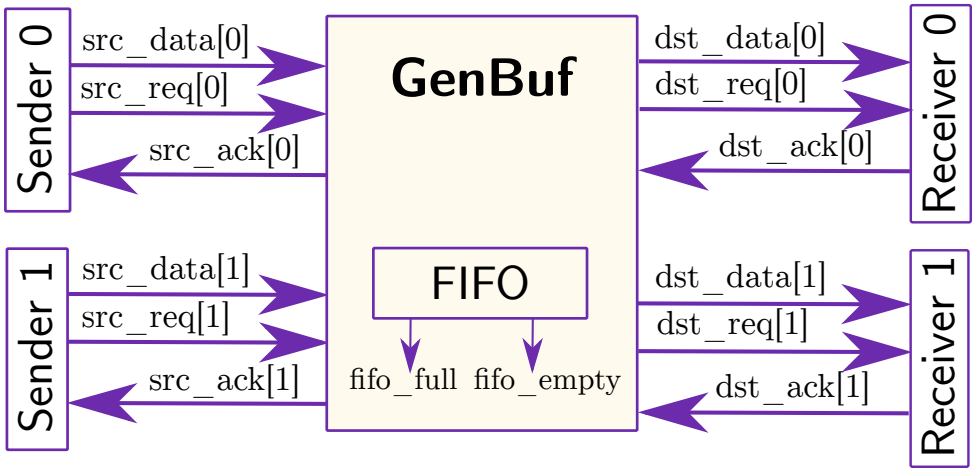


Figure 5.5: Generalized Buffer overview

Consequently, the formal specification of the GenBuf control interface consists of four sets of handshake properties. For instance, the handshake between the GenBuf and Receiver 0 is specified as:

(R0.1) assume **always** (! dst_req[0]) → **next** ! dst_ack[0]

(R0.2) assert **always** (dst_req[0] && ! dst_ack[0]) → **next** dst_req[0]

(R0.3) assume **always** (dst_req[0] && dst_ack[0]) → **next** dst_ack[0]

(R0.4) assert **always** dst_ack[0] → **next** ! dst_req[0]

The FIFO only reads one data, and writes another data. Then, the GenBuf only receives data from one sender at a time, and sends data to one receiver at a time. Properties 5 and 6 assert the mutual exclusion between the acknowledgments to the senders, and between the requests to the receivers. To complete the formal specification, properties 7 and 8 assert that the internal FIFO cannot overflow nor underflow:

(P5) assert **always** ! (src_ack[0] && src_ack[1])

(P6) assert **always** ! (dst_req[0] && dst_req[1])

(P7) assert **always** fifo_full → **next** src_ack = 00

(P8) assert **always** fifo_empty → **next** dst_req = 00

When all 8 assumptions are given (2 assumptions for each sender and each receiver), the 12 asserted properties are proved (2 assertions per sender and receiver, plus assertions (P5) to (P8)). When removing assumption (R0.1), then assertion (R0.4) fails. The following experiment will aim at inferring this removed assumption.

5.4.2.1 Results on the Generation of 10 CEx

In this experiment, we limit to 10 the number of CEx to be generated (setting the *maxLimit* parameter in Algorithm 1). The CEx generation step completes in 4 seconds. After mining assumptions from the CEx, Table 5.4 shows how many assumptions are discarded by each step of the flow, and how many assumptions remain.

Flow step	# Assumptions		CPU runtime (sec.)	
	Discarded	Remaining	Total	Per assumption
Property mining	-	1413	0.3	-
Input condition	757	656	<0.01	<0.01
Uniqueness	196	460	<0.01	<0.01
Common cause	228	232	<0.01	<0.01
Triviality	0	232	11	0.06
Consistency	42	190	24	0.13
Vacuity	134	56	117	0.72
Assertion check	50	6	8	0.18

Table 5.4: Extraction results on the GenBuf

We observe that, among the 1413 mined assumptions, 84% are discarded within a second by the first 3 syntactical criteria. Among the remaining 232 assumptions, 56 are neither inconsistent nor vacuous, and only 6 finally lead to a proof of the assertion. Note that 26 of these checks were not run, as their result was inferred from the implication criteria. Since we get some non-vacuous proofs, we discard the assumptions with a quality metric less than 1 (when the assertion fails). So, from the 1413 mined assumptions, only 6 are proposed to the user for review (see Table 5.5).

Assumption	Implied by
1. ! dst_req[0] → **next** ! dst_ack[0]	-
2. ! dst_req[0] → ! dst_ack[0]	-
3. (! dst_req[0] && **next** ! dst_req[0]) → **next** ! dst_ack[0]	1
4. (! dst_req[0] && **next** ! dst_req[1]) → **next** ! dst_ack[0]	1
5. (! dst_req[0] && ! dst_req[1]) → ! dst_ack[0]	2
6. (! dst_req[1] && **next** ! dst_req[0]) → **next** ! dst_ack[0]	2

Table 5.5: Assumptions extracted from the GenBuf

Assumption (1) is the one we removed from the initial problem. Assumption (2) is similar, but for a combinational control logic on the receiver side. Assumption (3) corresponds to a more specific variation of Assumption (1). The last three assumptions involve the request signals of both receivers. These assumptions may look spurious, but are actually more specific variations of the first three. Indeed, Assumption (1) implies

Assumption (4), and Assumption (2) implies Assumptions (5) and (6). Consequently, a verification engineer can easily review these assumptions and accept the one corresponding to the design specification.

On the performance side, the last two columns of Table 5.4 show that the formal checks (consistency and vacuity) take most of the runtime. In a previous version of the tool, we had assertion check performed before vacuity: model checking needed 418 seconds for running these filters on 190 assumptions, hence 3.3x more than with the current filter order. This motivates the importance of using lightweight filtering criteria in order to prune the set of assumptions from spurious ones.

5.4.2.2 Discussion on the Number of CEx

Under the hypothesis that only one assumption is missing, all the counter-examples that are generated exhibit the faulty behavior. Thus, using only one CEx ($|\mathcal{W}| = 1$), our assumption extraction will propose the correct missing assumption to the user, but it will be reported among many others. The advantage of using several CEx is to consider the common behavior of all of them, which also contains the faulty behavior. The assumptions proposed to the user, by application of the formula of Section 5.3.2.3, is therefore the intersection of the generated assumption sets for each CEx (and decreases when increasing $|\mathcal{W}|$). How many CEx should be generated in order to optimize performance, and to avoid proposing too many assumptions to the user?

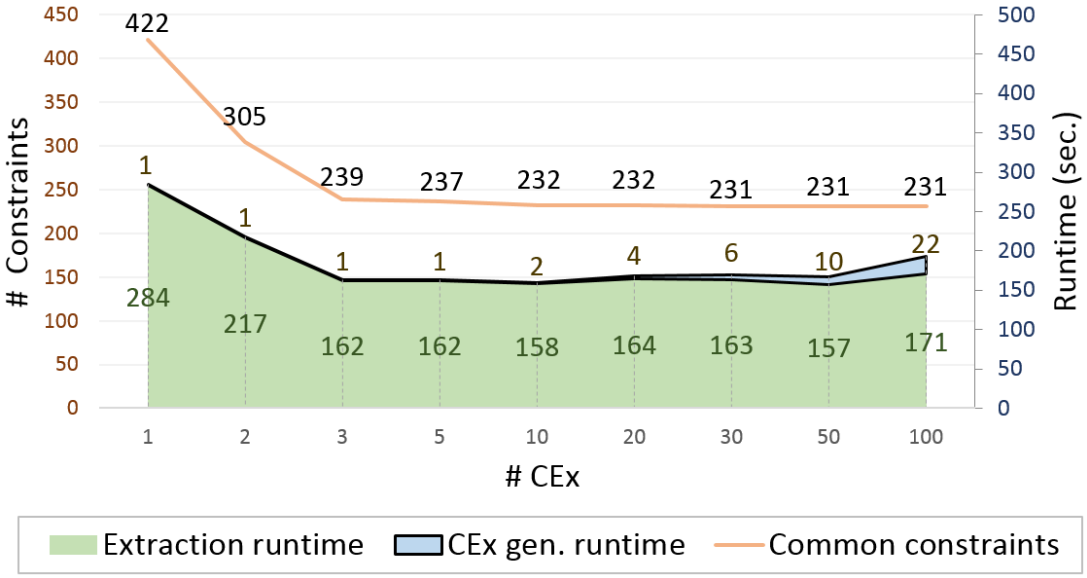


Figure 5.6: GenBuf runtime evolution with the CEx generation *maxLimit*

Figure 5.6 shows the evolution of the runtime and the number of returned assumptions, depending on the number of CEx being generated (maxLimit in Algorithm 1). The upper blue-filled part represents the runtime of the CEx generation. The lower green-filled part represents the runtime of the assumption extraction. The orange line represents the

number of generated assumptions after the "Common cause" filter. Note that, whatever the number of CEx, the same final 6 assumptions were eventually returned.

The total runtime is the sum of the CEx generation runtime (box 1 in Figure 5.4) and the extraction runtime (box 2 in Figure 5.4). It is optimal for 10 CEx in Figure 5.6. As expected, the orange line shows that increasing the number of CEx decreases the number of filtered common assumptions. However, this lower number does not compensates for the CEx generation time: between 10 and 100 CEx, the total runtime is increased by 21%, while only 1 assumption is removed. The same experiments have been repeated by removing different assumptions one by one from the GenBuf specification, and they show identical results. In all cases, a very small number of CEx (between 5 and 10) suffices to produce the optimal result and running time.

Note that the filtering of one assumption is independent from the next one. While the extraction runtime is high, it is highly parallelizable. Ideally, all assumptions can be filtered in parallel, which would reduce the total runtime to just a few seconds.

5.4.2.3 Discussion on the Efficiency of Justification

The usefulness of justification is another topic to be evaluated. For that purpose, we disabled justification, so that all the values of the primary inputs/outputs in the cone of influence of the assertion are considered during property mining. After running assumption extraction, we measured that the number of assumptions increases significantly compared to Table 5.4: 2543 assumptions are mined (1.8x), among which 549 are common to all CEx (2.4x). Since the filtering runtime is proportional to the number of common assumptions, it increases by 2.5x (470 seconds) when justification is disabled.

Note that, as justification involves a lightweight linear traversal of the cone of influence of the assertion, its computation time is very low – less than one second. Static design analysis is thus inexpensive and key to property mining quality.

5.4.2.4 Discussion on the Scalability

To exercise the scalability of our approach, we run the same experiment as before with various numbers of senders and receivers. The effect will be to increase the complexity of the design and of the formal specification. For the results to be comparable, the same assumption (R0.1) on receiver 0 is removed in each variation of the design.

As we can see in Table 5.6, increasing the complexity of the design generally increases both the number of generated assumptions and the runtime. Observe that increasing the number of senders from 2 to 5 only increases the runtime from 190 to 768 seconds. However, increasing the number of receivers multiplies the runtime by 92. As the removed assumption is on the receiver side, and the request logic for all receivers is strongly connected (via the round-robin), this complexity increase is expected.

In all obtained results, the correct assumption was generated in less than 40 minutes. This exhibits the scalability of our approach with an increasing complexity of the design and of its formal specification.

#Senders	#Receivers	#Assumptions Initial	#Assumptions Final	CPU runtime (sec.)
2	2	1413	6	190
2	3	2135	18	705
2	4	3047	57	4009
2	5	3049	127	17414
3	2	2476	8	378
4	2	2933	8	679
5	2	5844	8	768
4	4	9485	36	2413

Table 5.6: Extraction results on GenBuf variations

5.4.2.5 Discussion on Multiple Missing Assumptions

In all the experiments above, we assumed that only one assumption was missing. In this new experiment, we remove not one but two assumptions from the specification, and see how the filtering criteria can be tuned to still generate them. Among the possible benchmarks, we choose the GenBuf because it has the most complex specification due to the interaction between the protocols of the various modules. We consider the simple GenBuf with 2 senders and 2 receivers, and we remove the following assumptions:

- assume **always** (src_req[1] && ! src_ack[1]) → **next** src_req[1]
- assume **always** (! dst_req[0]) → **next** ! dst_ack[0]

10 counter-examples are generated from the failures of multiple different properties. Some CEx exhibit a failure which would be solved by one of the missing assumptions. Other failures occurred because both assumptions were removed.

As a proof of concept, the filtering criteria are tuned to better solve this problem. As mentioned in Section 5.3.2.3, the *Common cause* criterion is weakened by filtering out assumptions which occur in less than 50% of the CEx (instead of 100% in all previous experiments). Since multiple properties were originally failing, we also consider that an assumption passes the *Assertion check* if it leads to prove all these properties.

As shown in Table 5.7, from the original 1898 assumptions, only 56 reach the final filter. A first observation is that, as expected, no generated assumption achieves to prove all properties. They are then ordered depending on the number of properties that were proved, and depending on the quality metric μ. Actually, most of the generated assumptions had no effect on the failures, and led to a metric of 0. In the end, only 16 assumptions led to a proof of some properties or to a positive quality metric. Among these, the two removed assumptions are identified.

Note that the overall runtime is lower than in Table 5.4. This is explained because most of the results were failures, hence they were easier to find for the model checker.

Flow step	# Assumptions		CPU runtime (sec.)	
	Discarded	Remaining	Total	Per assumption
Property mining	-	1898	0.2	-
Input condition	1025	873	<0.01	<0.01
Uniqueness	282	591	<0.01	<0.01
Common cause	516	75	<0.01	<0.01
Triviality	0	75	3	0.06
Consistency	14	61	8	0.13
Vacuity	9	52	45	0.72
Assertion check	52	0	52	0.18
$\mu > 0$	36	16	-	-

Table 5.7: Extraction results on the GenBuf with 2 missing assumptions

This experiment exhibits the adaptability of the filtering criteria to solve variations of the problem with multiple missing assumptions.

The practical question that remains to be answered is: how to help the user find how many and which assumptions need to be included in the specification? We do not have a definitive answer to propose, but we can provide some hints. It is easy to know that more than one assumption is missing since adding only one does not prove all assertions. We identified two main situations:

- The missing assumptions involve signals that are independent. For instance, a same assumption is forgotten on all the instances of a module. This case is relatively simple, because a divide-and-conquer strategy can be applied to come back to the single missing assumption case.
- The missing assumptions are related and are needed together for the proof of one of more properties. This is a more complex case in which grouping strategies such as implication can be used.

5.4.3 AMBA from an Industrial Design

In order to verify the scalability of our method in an industrial design, we run our flow on the complex SoC design introduced in Section 3.5.3 on page 35. Using stuck-at assumptions, the design is fully configured in a mission mode: all the clock gates are enabled, and only one main clock is propagated to the registers. As explained in Section 2.2.4 (page 24), all external and internal/software resets are exercised to define a global initial state. Also, static primary inputs are constrained to the value given in the design specification.

The design includes a custom system bus with a power-optimization architecture, to interface between the CPU and external modules. In the following, without loss of relevance, we shall not disclose the actual bus signals, but shall reason as if it were an

AMBA AXI. We focus on one of the bus channels, the "read data" channel. This channel uses a variation of the handshake protocol, defined as follows:

1. assume **always** (ready && ! valid) → **next** ready
2. assert **always** (! ready && valid) → **next** valid
3. assume **always** (ready && valid) → **next** ! ready
4. assert **always** (ready && valid) → **next** ! valid

Two buses are studied:

- BUS1: the system control interface, by which the system reads data from the environment.
- BUS2: the system interface by which the interruptions are transmitted to the CPU.

Due to the size of the design, we need some abstraction to prune the state-space, for the model checker to give a conclusive result. We use our UCEGAR algorithm, as described in Chapter 4. Using the two assumptions (Properties 1 and 3 above), the model checker proves both assertions (Properties 2 and 4).

On each bus, we perform two experiments:

- We remove Property (3). Property (4) fails.
- We remove a reset assumption. Property (2) fails.

We then try to infer the missing assumption on the four experiments, with a limit of 10 CEx to be generated. The obtained results are given in Table 5.8.

Experiment	Bus	Removed assumption	# Assumptions Initial	# Assumptions Final	CPU runtime (sec.)	After abstraction # Latches	After abstraction # Signals
1	BUS1	Prop. 3	813	15	967	1177	8437
2	BUS1	Reset	705	1	376	1075	7731
3	BUS2	Prop. 3	68	5	26	476	2612
4	BUS2	Reset	27	1	33	255	1518

Table 5.8: Extraction results on the industrial design

Observe that the number of generated assumptions, and thus the execution run time, are significantly higher for BUS1 than for BUS2. This is due to the cone of influence of the signals named in the properties *after abstraction*. The numbers are given by columns *# Latches* and *# Signals* in Table 5.8. Initially, for experiments 1 and 3, these cones of influence were: 15,658 nets and 2,054 latches for Bus1; 343,769 nets and 42,134 latches for Bus2. The position of a same block in the design greatly influences the results, which was expected.

It is interesting to note that, for a given assertion, the final number of returned assumptions bears some dependency on the initial number of generated assumptions. For

experiments 1 and 3, we go from 813 down to 15 for Bus1, from 68 down to 5 for Bus2. After analysis of the returned assumptions, we arrive at the same conclusions as for the GenBuf: some assumptions are assuming a sequential control logic, others a combinational logic; some imply the missing assumption, and this missing assumption is among the final set. For experiments 2 and 4, the filtering iterations are specially effective, since a single missing assumption is returned to the user, and it is the exact assumption on the reset. Finally, these results confirm that the CPU run time is proportional to the initial number of generated assumptions.

If we compare the initial numbers of generated assumptions in Table 5.6 (GenBuf) and Table 5.8, the much larger numbers of Table 5.6 are an indication that the protocols in the small GenBuf are much more complex. This is why we did not perform the removal of more than one assumption on this industrial design.

This experiment exhibits the scalability of our approach: even for a large design, generating multiple CEx and filtering the set of assumptions down to a modest returned set takes less than half an hour.

5.5 Conclusion

Our main contribution is a novel and efficient flow to automatically produce missing assumptions on a design environment. This work has been presented in the international conference MEMOCODE [65] and was extended for a journal publication in ACM TECS [66].

As a first step, our flow generates multiple distinct counter-examples for the failing assertion (counter-examples that are not equivalent after stuttering removal). We showed that a small number of counter-examples is enough to contain relevant and sufficient information. The efficiency of this generation relies on the application of the structural justification method, which brings significant improvements compared to the cone of influence analysis alone.

As a second step, the counter-examples are mined for properties which could reveal the cause of the failure. Corresponding assumptions avoiding these behaviors are generated in the form of multi-cycle temporal properties. To be useful to the verification engineer, the returned assumptions must be few, relevant and realistic. We implemented a series of filtering criteria, and showed their practical efficiency.

In case the failing assertion cannot be corrected by a single assumption, the generated assumptions are ranked according to their influence on the formal check. Using both the justified counter-examples and the proposed assumptions, a verification engineer is able to debug the failure and provide correct assumptions on the design environment.

CHAPTER 6

Conclusion & Outlook

6.1 Enhanced Formal Verification Flow

The formal verification of complex designs *integrating multiple clocks* is too hard a problem for brute force model checking. In this thesis, our purpose was not to improve model checking engines, but to provide practical solutions that make the problem tractable. We adopted a divide-and-conquer strategy that can easily be incorporated in the standard verification flow of Figure 1.13 on page 11. More precisely, we addressed in isolation three well identified verification steps: the clock setup, the clock-domain crossing functional check and the debugging of incomplete specifications. The combination of our proposed solutions results in an enhanced formal verification flow, as presented in Figure 6.1.

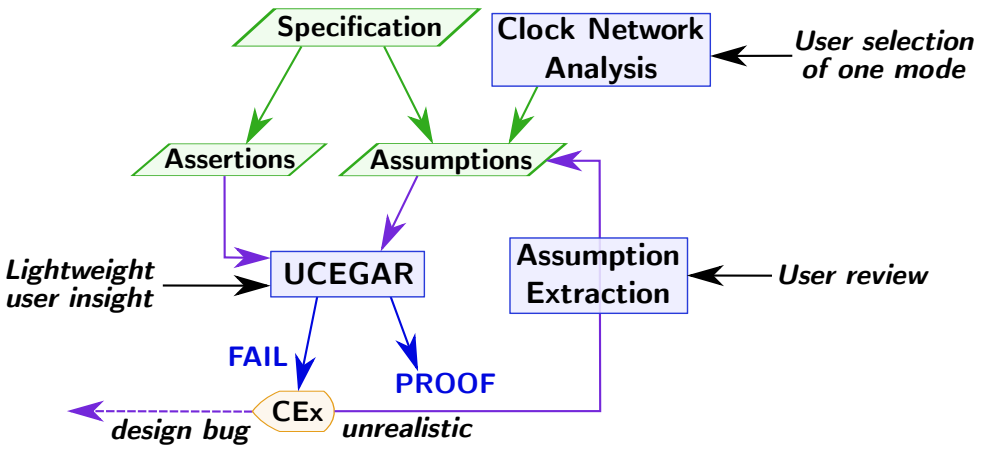


Figure 6.1: Enhanced formal verification flow

First, using the design specification, engineers are used to describe the behavior of some primary inputs with formal assumptions. Then, our clock network analysis identifies all potential behaviors of each internal clock (Chapter 3). The definition of these operation modes, using formal assumptions, results in a realistic and complete setup of the clocks in the design, which is a mandatory step for any formal verification flow.

The formal assertions to be verified are generated after performing a structural analysis of the design which detects CDC synchronization patterns. During the formal check of these assertions, our user-guided counter-example abstraction refinement (UCEGAR) tackles the state-space explosion problem using both a structural analysis and lightweight feedback from the user (Chapter 4). The model checking of the CDC synchronizer properties then always provides a conclusive result: either guaranteeing the absence of bug,

or returning a localized counter-example. Using justification, the latter is pruned from signal values which are irrelevant to the failure of the assertion, so that the engineer easily analyzes it.

If this failure is considered to be unrealistic, the reason is most likely a missing protocol assumption on the design environment. To complete this debugging step, we provide an algorithm which generates multiple counter-examples for the same assertion, then mines their common behaviors (Chapter 5). Using this information, a succinct list of realistic missing assumptions is reported to the user for review.

Overall, the key ingredient leading to conclusive results appeared to be the combination of inexpensive structural analysis along with exhaustive formal analysis. The joint formalization of both the netlist graph and the state machine allows the consideration of this duality in many parts of our flow. Thus, even on modern large industrial designs, our enhanced verification flow ensures the absence of CDC issues while setting up the design in a realistic mode.

6.2 Outlook

As a future work, all our techniques could be brought together into one unified flow. First, the result of the clock network analysis could be considered as an input to the assertion check. Indeed, while selecting one operation mode makes the clocks to behave deterministically, the whole logic stays in the model. As seen with the constant propagation in Chapter 4, even fully deterministic logic influences the complexity of formal verification. By replacing the internal clocks with deterministic clock monitors, the clock network would be pruned from the design, hence decreasing the formal verification runtime while keeping its soundness.

Also, the UCEGAR flow could be coupled with the assumption extraction. More precisely, after the design abstraction of the UCEGAR flow, multiple counter-examples could be created and mined for information. On the one hand, this would provide more insight on the relevant signals of the intermediary counter-examples. On the other hand, it would lead to more precise assumptions than a cut-point or a stuck-at – which are the ones currently proposed by UCEGAR.

In addition to this unification of all our methods, there are several directions in which to improve them individually. *SYNOPSYS CONFIDENTIAL*

The long-term goal is to extend our methodology to many critical functional verification steps in the VLSI flow. In particular, our UCEGAR flow appears to be very effective when the logic that is relevant to the assertion is small, but its boundaries cannot be easily identified. For instance, the same UCEGAR could be reused to functionally verify false or multi-cycle paths. On the way, this would lead us to improve the UCEGAR heuristics based on the type of assertion.

Multiple parts of our assumption extraction can also be improved. First, in generating the counter examples, we could consider not only stuttering states, but also a stuttering

sequence of states (even though the added computing cost will need to be evaluated). Another extension concerns the set of assumption templates. Industrial libraries of properties are much richer (for instance the ones described in OVL[3]). A systematic analysis of such libraries, and prioritization of the assumptions to be applied after mining is yet to be performed. Also, in view of obtaining fast results on large designs, a systematic parallelization of our algorithms should provide large performance improvements. Finally, the property mining from counter-examples could be investigated in the scope of RTL debugging.

APPENDIX A

Table of Notations

General	
$\mathbb{B}$	Boolean set
$\mathbb{N}$	Naturals set
Netlist model	
$\mathcal{D}$	Design graph
$\mathcal{A}$	Set of all signals
$\mathcal{E}$	Set of all edges
Type	Type function
$\mathcal{T}$	Set of signal types
α	Signal
$\mathcal{T}_{\mathsf{zero}}$	Constant zero signal type
$\mathcal{T}_{\mathsf{one}}$	Constant one signal type
$\mathcal{T}_{\mathsf{in}}$	Primary input signal type
$\mathcal{T}_{\mathsf{out}}$	Primary output signal type
$\mathcal{T}_{\mathsf{not}}$	Inverter signal type
$\mathcal{T}_{\mathsf{and}}$	AND-gate signal type
$\mathcal{T}_{\mathsf{or}}$	OR-gate signal type
$\mathcal{T}_{\mathsf{seq}}$	Sequential signal type
$\mathcal{A}_*$	Set of signals of type $\mathcal{T}_*$
$\mathcal{A}_{\mathsf{flop}}$	Set of flop output signals
$\mathcal{A}_{\mathsf{flop},*}$	Set of flop *-input signals
$\mathcal{A}_{\mathsf{latch}}$	Set of latch output signals
$\mathcal{A}_{\mathsf{latch},*}$	Set of latch *-input signals
$\mathcal{A}_{\mathsf{mux}}$	Set of mux output signals
$\mathcal{A}_{\mathsf{mux},*}$	Set of mux *-input signals
$\mathsf{getClockOf}$	Function from sequential output to CLK input
Pred	Predecessor function in $\mathcal{D}$
Succ	Successor function in $\mathcal{D}$
$\pi_{\alpha..\alpha'}$	Structural path from α to α'
COI_α	Cone-of-influence of α
Functional model	
$\mathcal{M}$	Moore machine
Σ_s	State alphabet

$\Sigma_{s,0}$	Initial states
Σ_l	Input alphabet
Σ_g	Output alphabet
δ	Transition function
λ	Output function
s	State
l	Input alphabet letter
g	Output alphabet letter
Σ	Global alphabet
c	Configuration of the machine (letter of the global alphabet)
ω	Execution (reachable word of the global alphabet)
Ω	Set of executions
Val_ω	Valuation function in ω
Val	Generalized valuation function

Formal Verification

χ	Formal assumption
$\mathcal{X}$	Set of assumptions
χ_{user}	User-defined assumptions
χ_{all}	Set of assumptions
χ_{add}	Set of assumptions
ϕ	Formal property (mostly used for assertions)
$\mathcal{P}$	Set of all safety properties upon $\mathcal{A}$
α_ϕ	Output signal of the monitor for property ϕ
$\mathcal{M}\|\chi \vDash \phi$	Model $\mathcal{M}$ satisfies ϕ when constrained by χ

Algorithms

check	Formal check / Model checking
signals	Signals (and their whole bus) considered in an execution
size	Size of a set
push/enqueue	Adds an item
pop/dequeue	Removes and returns the top item
top	Returns the top item
Queue	Queue data structure
Stack	Stack data structure
Visited	Set of visited signals in a BFS/DFS traversal
res	Result of the algorithm

Clock Network Analysis

$\mathcal{A}_{\mathsf{data},\phi}$	Set of data signals for property ϕ
$\mathcal{A}_{\mathsf{clk}}$	Set of clock signals
$\mathcal{A}_{\mathsf{clk,prim}}$	Set of primary clock signals

$\mathcal{A}_{\mathsf{clk,deriv}}$	Set of derived clock signals
$\mathcal{A}_{\mathsf{rst}}$	Set of reset signals
$\mathcal{A}_{\mathsf{rst,prim}}$	Set of primary reset signals
$\mathcal{A}_{\mathsf{conf}}$	Set of configuration signals
m	Mode (vector of configuration values)
M	Set of modes
Dom	Returns the set of primary clocks reaching a signal
Blocking Condition	
$\mathsf{disable}$	Creation of a blocking condition
Lit_{ω}	Litteral function
χ_{ω}	Blocking condition
Justification	
$J_{\omega,\phi}$	Justification function
$\widehat{\omega}$	Justified execution
$\mathcal{J}_{\omega}$	Set of justified signals
UCEGAR	
$\mathcal{D}^{\#}$	Abstracted design
$\omega^{\#}$	Abstract counter-example
Cut	Set of cut-points
Ref	Set of signals to refine
Extraction of Assumptions	
$\mathrm{T_C}$	Assumption template function
$\mathrm{T_M}$	Property template function
$\mathcal{T}_{\mathcal{C}}$	Set of assumption templates
$\mathcal{T}_{\mathcal{M}}$	Set of property templates
$\prec$	Strict order for instantiated templates
Impl	Set of implications of instantiated templates
μ	Metric of assumptions relevance

APPENDIX B

Reset Propagation Algorithm

SYNOPSYS CONFIDENTIAL

APPENDIX C

Simple Clock Tree Description in Verilog

```
module test(input clk1, clk2, clk3, SEL, EN, in, RST, [1:0] CFG,
            output reg out);
  reg[1:0] divCounter;
  always @(posedge clk2 or posedge RST)
  begin
    if (RST)
      divCounter <= 0;
    else if (divCounter==CFG)
      divCounter <= 0;
    else
      divCounter <= divCounter + 1;
  end
  reg clkdiv;
  always @(posedge clk2)
  begin
    if (divCounter == CFG) begin
      clkdiv <= ~clkdiv;
    end
  end

  assign clksel = SEL ? clkdiv : clk1;

  reg enLatch;
  always @(clksel)
  begin
    if (~clksel)
      enLatch <= EN;
  end
  assign clkgate = clksel & enLatch;

  always @(posedge clkgate or posedge RST)
  begin
    if (RST)
      out <= 0;
    else
      out <= in;
  end
endmodule
```

APPENDIX D

Additional CNA Results on the Industrial Design

Derived: CLKCPU		#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	#17
Source		CLK1	CLK1	CLK1	CLK1	CLK1	CLK1	CLK1	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2
Frequency		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Configuration signals	GE	-	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	O2	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SY	-	-	-	0	0	0	0	0	0	0	1	-	-	-	-	-	-
	CG	-	-	-	0	0	0	0	-	-	-	-	1	1	1	1	-	-
	EG	-	-	-	0	0	0	0	1	1	1	1	-	-	-	-	1	1
	O6	-	-	-	0	0	0	0	-	1	-	-	1	1	-	-	1	-
	O7	-	-	-	0	0	0	0	-	-	1	1	-	-	1	-	-	1
	O39	0	-	-	0	0	0	0	0	0	0	0	-	0	0	0	0	-
	O40	0	-	-	0	0	0	0	-	-	-	-	-	-	-	-	-	-
	FT	-	-	-	-	-	1	-	0	-	-	-	-	-	-	0	-	-
	S2	-	-	-	0	0	-	-	1	-	-	-	-	-	-	1	-	-
	O32	-	1	-	0	-	-	1	0	0	0	0	0	0	0	0	0	0
	O38	0	-	1	-	1	-	-	0	0	0	0	0	0	0	0	0	0
	O3	0	0	0	-	-	-	-	0	0	0	0	1	0	-	-	-	1
	O117	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O1	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O30	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0	0
	S16	-	-	-	-	-	-	-	1	1	1	1	1	1	1	1	1	1
	S0	-	-	-	-	-	-	-	1	-	-	1	1	-	-	1	-	-
	EM	-	-	-	-	-	-	-	-	1	1	-	-	1	1	-	1	1

Derived: CLKCPU		#18	#19	#20	#21	#22	#23	#24	#25	#26	#27	#28	#29	#30	#31	#32
Source		CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2	CLK2
Frequency		1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2
Configuration signals	GE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	O2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SY	0	0	-	-	-	-	-	-	-	-	-	-	-	0	0
	CG	1	1	1	1	1	1	1	1	-	-	-	-	-	-	-
	EG	-	-	-	-	-	-	-	-	1	1	1	1	1	1	1
	O6	1	-	1	1	1	-	-	-	1	1	1	-	-	-	-
	O7	-	1	-	-	-	-	-	-	-	-	-	-	1	-	-
	O39	0	0	0	-	-	-	0	0	0	-	-	-	-	0	0
	O40	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	FT	-	-	-	-	-	0	0	0	-	-	-	0	-	0	0
	S2	-	-	-	-	-	1	1	1	-	-	-	1	-	1	1
	O32	-	-	-	-	-	1	-	-	-	-	-	-	-	1	1
	O38	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	O3	0	0	-	1	1	1	-	-	-	1	1	1	1	0	0
	O117	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	S16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	S0	1	1	-	1	-	-	1	-	-	-	1	-	-	-	1
	EM	-	-	1	-	1	1	-	1	1	1	-	1	1	1	-

APPENDIX E

Implications Between Assumption Templates

Given the templates of Table 5.1, the following table shows all potential implications. For brevity, we consider α_1, α_2, α_3 to be literals instead of signals. With α_1 being the literal for signal α, it can be read as either α or $!\alpha$.

χ	χ'
always α_1	**always** $\neg\alpha_1 \rightarrow \alpha_2$
always α_2	**always** $\alpha_1 \rightarrow \alpha_2$
always α_1	**always** $\neg\alpha_1 \rightarrow$ **next** α_2
always α_2	**always** $\alpha_1 \rightarrow$ **next** α_2
always α_1	**always** $(\neg\alpha_1 \,\&\&\, \alpha_2) \rightarrow \alpha_3$
always α_2	**always** $(\alpha_1 \,\&\&\, \neg\alpha_2) \rightarrow \alpha_3$
always α_3	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow \alpha_3$
always α_1	**always** $(\neg\alpha_1 \,\&\&\, \alpha_2) \rightarrow$ **next** α_3
always α_2	**always** $(\alpha_1 \,\&\&\, \neg\alpha_2) \rightarrow$ **next** α_3
always α_3	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow$ **next** α_3
always α_1	**always** $(\neg\alpha_1 \,\&\&$ **next** $\alpha_2) \rightarrow$ **next** α_3
always α_2	**always** $(\alpha_1 \,\&\&$ **next** $\neg\alpha_2) \rightarrow$ **next** α_3
always α_3	**always** $(\alpha_1 \,\&\&$ **next** $\alpha_2) \rightarrow$ **next** α_3
always $\alpha_1 \rightarrow \alpha_2$	**always** $(\alpha_1 \,\&\&\, \neg\alpha_2) \rightarrow \alpha_3$
always $\alpha_1 \rightarrow \alpha_3$	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow \alpha_3$
always $\alpha_2 \rightarrow \alpha_3$	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow \alpha_3$
always $\alpha_1 \rightarrow \alpha_2$	**always** $(\alpha_1 \,\&\&\, \neg\alpha_2) \rightarrow$ **next** α_3
always $\alpha_2 \rightarrow \alpha_3$	**always** $(\alpha_1 \,\&\&$ **next** $\alpha_2) \rightarrow$ **next** α_3
always $\alpha_1 \rightarrow$ **next** α_3	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow$ **next** α_3
always $\alpha_2 \rightarrow$ **next** α_3	**always** $(\alpha_1 \,\&\&\, \alpha_2) \rightarrow$ **next** α_3
always $\alpha_1 \rightarrow$ **next** α_3	**always** $(\alpha_1 \,\&\&$ **next** $\alpha_2) \rightarrow$ **next** α_3
always $\alpha_1 \rightarrow$ **next** α_2	**always** $(\alpha_1 \,\&\&$ **next** $\neg\alpha_2) \rightarrow$ **next** α_3

Bibliography

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *ICALP*, pages 1–17, 1989.

[2] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. Focs: Automatic generation of simulation checkers from formal specifications. In *Computer Aided Verification*, CAV '00, pages 538–542, London, UK, UK, 2000. Springer-Verlag.

[3] Accellera. *Open Verification Library (OVL)*, Accessed April 2014. `http://accellera.org/activities/working-groups/ovl`.

[4] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *ASP-DAC*, pages 19–24, 2006.

[5] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *Design Automation Conference (DAC)*, pages 218–223, 2004.

[6] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[7] L. Benini and G. d. Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[8] M. Bertasi, G. Di Guglielmo, and G. Pravadelli. Automatic generation of compact formal properties for effective error detection. In *CODES+ISSS*, pages 1–10. IEEE, 2013.

[9] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160 – 177, 2002. FMICS.

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 193–207, 1999.

[11] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design, Automation and Test in Europe*, DATE '07, pages 1188–1193, San Jose, CA, USA, 2007. EDA Consortium.

[12] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3 – 16, 2007. Compiler Optimization meets Compiler Verification.

[13] M. Boule and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *International Conference on Computer Design*, pages 221–228, Oct 2005.

[14] R. K. Brayton and A. Mishchenko. ABC: an academic industrial-strength verification tool. In *CAV*, pages 24–40, July 2010.

[15] F. Burns, D. Sokolov, and A. Yakovlev. GALS synthesis and verification for xMAS models. In *DATE*, 2015.

[16] D. Bustan, D. Fisman, and J. Havlicek. Automata construction for psl. Technical report, Freescale Semiconductor, Inc, 2005.

[17] T. Chaney and C. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[18] K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *CONCUR*, pages 147–161, 2008.

[19] S. Chaturvedi. Static analysis of asynchronous clock domain crossings. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1122–1125, March 2012.

[20] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

[21] E. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *ACM*, 1991.

[22] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.

[23] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, Jun 1989.

[24] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[25] C. E. Cummings. Clock domain crossing design & verification techniques using systemverilog. In *SNUG Boston, MA*, 2008.

[26] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *DATE*, pages 67–72, March 2015.

[27] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134, 2011.

[28] E. G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–692, May 2001.

[29] T. J. Gabara, G. J. Cyr, and C. E. Stroud. Metastability of CMOS master/slave flip-flops. In *IEEE Custom Integrated Circuits Conference*, pages 29.4/1–29.4/6, May 1991.

[30] S. V. Gheorghita and R. Grigore. Constructing checkers from PSL properties. *Control Systems and Computer Science*, 2:757–762, 2005.

[31] R. Ginosar. Fourteen ways to fool your synchronizer. In *Asynchronous Circuits and Systems*, pages 89–96, May 2003.

[32] R. Ginosar. Metastability and synchronizers: A tutorial. *Design Test of Computers, IEEE*, 28(5):23–35, Sept 2011.

[33] S. Hagihara, Y. Kitamura, M. Shimakawa, and N. Yonezaki. Extracting environmental constraints to make reactive system specifications realizable. In *APSEC*, pages 61–68, 2009.

[34] Haifa-IBM-Laboratories. *IBM Generalized Buffer*, Accessed May, 2017. `http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/tutorial3/`.

[35] S. Hertz, D. Sheridan, and S. Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Trans. on CAD*, 32(6):952–965, 2013.

[36] IEEE. IEEE standard for verilog hardware description language. *IEEE Std 1364-2005*, pages 1–560, 2006.

[37] IEEE. IEEE standard vhdl language reference manual. *IEEE Std 1076-2008*, pages 1–620, Jan 2009.

[38] IEEE. IEEE standard for property specification language (PSL). *IEEE Std 1850-2010)*, pages 1–182, April 2010.

[39] IEEE. IEEE standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2012*, pages 1–1315, Feb 2013.

[40] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. *Anzu: A Tool for Property Synthesis*, pages 258–262. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[41] T. Kapschitz and R. Ginosar. Formal verification of synchronizers. In *Correct Hardware Design and Verification Methods*, pages 359–362, 2005.

[42] N. Karimi and K. Chakrabarty. Detection, diagnosis, and recovery from clock-domain crossing failures in multiclock SOCs. *Computer-Aided Design of Integrated Circuits and Systems*, 32(9):1395–1408, Sept 2013.

[43] M. Kebaili, J.-C. Brignone, and K. Morin-Allory. Clock domain crossing formal verification: a meta-model. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 136–141, Oct 2016.

[44] M. Kebaili, G. Plassan, J.-C. Brignone, and J.-P. Binois. Conclusive formal verification of clock domain crossings using spyglass-cdc. In *SNUG France*, June 2016. Best paper award from the technical committee.

[45] B. Keng. *Advances in Debug Automation for a Modern Verification Environment*. PhD thesis, University of Toronto, 2013.

[46] B. Keng, E. Qin, A. Veneris, and B. Le. Automated debugging of missing assumptions. In *Asia-Pacific DAC*, pages 732–737. IEEE Computer Society, 2014.

[47] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, Nov 2001.

[48] R. P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach*. Princeton University Press, 1994.

[49] C. Kwok, V. Gupta, and T. Ly. Using assertion-based verification to verify clock domain crossing signals. In *Design and Verification Conference*, pages 654–659, 2003.

[50] Leda CDC Documentation. Clock domain crossing. Online, Sept 2017. `https://filebox.ece.vt.edu/~athanas/4514/ledadoc/html/pol_cdc.html`.

[51] C. Leong, P. Machado, and et. al. Built-in clock domain crossing (CDC) test and diagnosis in GALS systems. In *Proc. DDECS 2010*, pages 72–77, April 2010.

[52] B. Li and C.-K. Kwok. Automatic formal verification of clock domain crossing signals. In *ASP-DAC*, pages 654–659, Jan 2009.

[53] W. Li, L. Dworkin, and S. A. Seshia. Mining assumptions for synthesis. In *MEMOCODE*, pages 43–50, 2011.

[54] M. Litterick. Pragmatic simulation-based verification of clock domain crossing signals and jitter using SystemVerilog Assertions. In *DVCON*, 2006.

[55] M. Litterick. Full flow clock domain crossing - from source to si. In *DVCON*, 2016.

[56] A. Lozano and J. L. Balcázar. The complexity of graph problems for succinctly represented graphs. In *Graph-Theoretic Concepts in Computer Science*, pages 277–286, 1989.

[57] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[58] P. C. McGeer and R. K. Brayton. Efficient algorithms for computing the longest viable path in a combinational network. In *ACM/IEEE Design Automation Conference*, pages 561–567, June 1989.

[59] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[60] Mentor Graphics. Questa CDC. Online, Jan 2017.

[61] A. Mishchenko, N. Een, and R. Brayton. A toolbox for counter-example analysis and optimization. In *IWLS*, 2013.

[62] K. Morin-Allory and D. Borrione. A proof of correctness for the construction of property monitors. In *Tenth IEEE International High-Level Design Validation and Test Workshop, 2005.*, pages 237–244, Nov 2005.

[63] K. Morin-Allory, F. N. Javaheri, and D. Borrione. Efficient and correct by construction assertion-based synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(12):2890–2901, Dec 2015.

[64] Oracle. OpenSPARC T2. Online, Nov 2017. http://www.oracle.com/technetwork/systems/opensparc/opensparc-t2-page-1446157.html.

[65] G. Plassan, K. Morin-Allory, and D. Borrione. Extraction of missing formal assumptions in under-constrained designs. In *ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '17, pages 94–103. ACM, 2017.

[66] G. Plassan, K. Morin-Allory, and D. Borrione. Mining missing assumptions from counter-examples. In *ACM Transactions on Embedded Computing Systems*. ACM, 2018. Invited extended version of MEMOCODE'17. Submitted.

[67] G. Plassan, H.-J. Peter, K. Morin-Allory, F. Rahim, S. Sarwary, and D. Borrione. Conclusively verifying clock-domain crossings in very large hardware designs. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Sept 2016.

[68] G. Plassan, H.-J. Peter, K. Morin-Allory, S. Sarwary, and D. Borrione. *Improving the Efficiency of Formal Verification: The Case of Clock-Domain Crossings*, pages 108–129. Springer International Publishing, 2017.

[69] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Oct 1977.

[70] A. Pnueli. Logics and models of concurrent systems. In K. R. Apt, editor, *In Transition from Global to Modular Temporal Reasoning About Programs*, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[71] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[72] R. K. Ranjan, C. Coelho, and S. Skalberg. Beyond verification: Leveraging formal for debugging. In *DAC*, pages 648–651, July 2009.

[73] Real Intent. Meridian CDC. Online, Jan 2017. http://www.realintent.com/real-intent-products/meridian-cdc/.

[74] S. Sarwary, H.-J. Peter, G. Plassan, B. Chakrabarti, and M. Movahed. Formal clock network analysis visualization, verification and generation, 2017. Provisional Patent Application.

[75] S. Sarwary and S. Verma. Critical clock-domain-crossing bugs. *Electronics Design, Strategy, News*, April 2008.

[76] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, Sep 1994.

[77] Synopsys. Spyglass CDC. Online, Jan 2017. https://www.synopsys.com/verification/static-and-formal-verification.html.

[78] G. Tarawneh, A. Mokhov, and A. Yakovlev. Formal verification of clock domain crossing using gate-level models of metastable flip-flops. In *DATE*, pages 1060–1065, March 2016.

[79] University of Washington. *The Daikon Invariant Detector*, Accessed May, 2017. http://plse.cs.washington.edu/daikon/.

[80] M. Y. Vardi. *Alternating automata: Unifying truth and validity checking for temporal logics*, pages 191–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[81] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the Symposium on Logic in Computer Science*, pages 332–344, 1986.

[82] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *DATE*, pages 626–629, 2010.

[83] L. Xiu. *From frequency to time-average-frequency: a paradigm shift in the design of electronic system*. IEEE Press series on microelectronic systems. Wiley-IEEE Press, Hoboken, NJ, 2015.

APPENDIX F

Résumé en Français

Introduction

Les systèmes numériques sont désormais présents massivement dans notre vie quotidienne. Pour tout produit, le moindre problème fonctionnel peut causer des conséquences désastreuses sur le plan financier ou humain. Un système a ainsi besoin d'être validé sur de multiples critères afin de détecter ses problèmes aussi tôt que possible. En particulier, et contrairement aux logiciels, les circuits électroniques doivent être validés sur des modèles virtuels, avant leur fabrication.

Le flot de conception d'un circuit microélectronique se déroule sur plusieurs niveaux d'abstraction. A chaque étape, une technique de vérification est utilisée afin de détecter et corriger les potentielles erreurs au plus tôt. A partir d'un langage de description de matériel tel que le Verilog ou le VHDL, la simulation de traces d'exécutions est une technique très utilisée, bien que ne garantissant pas l'absence de bogue dans le circuit. Lorsque le circuit est utilisé dans un contexte où la sécurité est critique, et donc qu'aucun bogue ne doit subsister, des méthodes formelles sont utilisées. Celles-ci modélisent le circuit en équations mathématiques qui décrivent entièrement son comportement, et permettent ainsi de garantir l'absence de bogue. Toutefois, la complexité de cette vérification formelle oblige les ingénieurs de vérification à ne l'appliquer que sur des tailles de circuits modestes.

Toutefois, certains circuits très complexes sont aussi très utilisés, notamment dans les objets connectés. Cette complexité vient en particulier de certaines optimisations de performance et de consommation, qui consistent à adaptent la fréquence de l'horloge aux tâches en cours. Plusieurs domaines d'horloges sont alors utilisés : des horloges rapides pour les tâches demandant de la performance, et des horloges plus lentes pour les tâches non critiques. Cela génère des milliers d'interconnexions entre les différentes horloges, aussi appelées *traversées de domaines d'horloges* (ou CDC).

Typiquement, un CDC se manifeste par un circuit numérique entre deux éléments séquentiels recevant des horloges déphasées. De nombreux problèmes sont soulevés par les CDCs [25, 75] (métastabilité, incohérence des bus, corruption des données, valeurs transitoires, ...), et les concepteurs des circuits doivent implémenter des structures particulières afin d'éviter tout bogue [25, 31] (flop cascadées, protocole poignée de main, codage de Gray, ...). Les ingénieurs de vérification doivent alors vérifier que tous les problèmes possibles aient été adressés.

Les principaux fournisseurs d'outils fournissent des outils d'analyse des CDC : Synopsys SpyGlass CDC [77], Mentor Questa CDC [60], Real Intent Meridian CDC [73],.... Une analyse structurelle du RTL leur permet de détecter les synchroniseurs, puis des

propriétés formelles correspondant aux protocoles sont générées [41, 43, 49, 54], e.g. en PSL [38] ou SVA [39]. Ces propriétés peuvent alors être vérifiées par simulation ou par les méthodes formelles. Toutefois, étant donné la nature rare des phénomènes propres aux CDC, la vérification formelle est souvent préférée.

Le principal problème de la vérification de modèles utilisant les méthodes formelles est qu'elle nécessite des temps trop élevés pour prouver les propriétés des systèmes multi-horloges complexes. L'apport d'un expert est alors nécessaire afin de préciser les modèles et d'ajuster la configuration du moteur formel, pour enfin atteindre un résultat conclusif : une preuve, ou un échec des propriétés. Au lieu d'améliorer les moteurs formels, cette thèse prend l'approche de simplifier le modèle, tout en répondant aux problématiques suivantes :

- Configurer le modèle du système dans un mode réaliste ;
- Générer des hypothèses sur les protocoles modélisant l'environnement ;
- Contrer l'explosion de l'espace des états ;
- Analyser les contre-exemples.

Vérification formelle des modèles de circuits

Les langages de description de matériel sont représentés par une sémantique structurelle et une sémantique fonctionnelle. Afin de modéliser cette dualité, nous utilisons deux graphes : un graphe orienté dont les noeuds sont étiquetés (modèle structurel) et une machine de Moore (modèle fonctionnel).

Le modèle structurel est créé par la synthèse logique, qui transforme le RTL en graphe de composants parmi : entrée primaire, sortie primaire, constante à zéro, constante à un, porte ET, porte OU, et opérateur séquentiel. Un exemple de modèle structurel est présent sur la Figure F.1, où chaque type de noeud est représenté par un symbole différent.

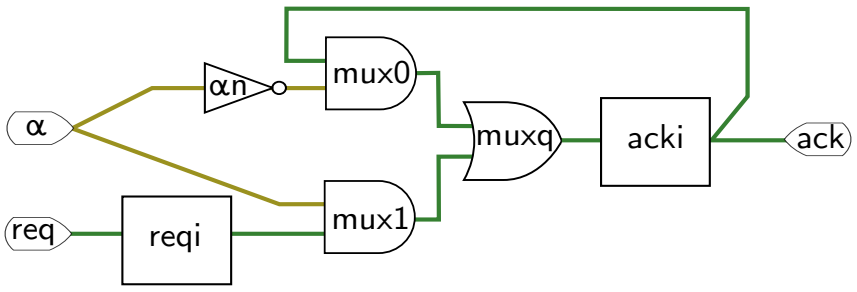


Figure F.1: Example de graphe structurel pour un protocole de poignée de main

Le modèle est formellement défini par $\mathcal{D} = \langle \mathcal{A}, \mathcal{E}, \mathsf{Type} \rangle$, où:

- $\mathcal{A}$ est l'ensemble des noeuds (aussi appelés signaux) ;
- $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$ est l'ensemble des arcs ;

- $\mathsf{Type} : \mathcal{A} \rightarrow \mathcal{T}$ est la fonction d'étiquetage,
  avec $\mathcal{T} = \{\mathcal{T}_{\mathsf{in}}, \mathcal{T}_{\mathsf{out}}, \mathcal{T}_{\mathsf{zero}}, \mathcal{T}_{\mathsf{one}}, \mathcal{T}_{\mathsf{not}}, \mathcal{T}_{\mathsf{and}}, \mathcal{T}_{\mathsf{or}}, \mathcal{T}_{\mathsf{seq}}\}$.

Afin de construire le modèle fonctionnel $\mathcal{M}$ correspondant à $\mathcal{D}$, chaque signal est associé à une valeur binaire. La sémantique $\mathcal{M}$ est décrite par une machine déterministe à états finis, où chaque état correspond à une combinaison des valeurs des opérateurs séquentiels. Un état transite vers un autre après avoir évalué la logique combinatoire, et avoir mis à jour les valeurs individuelles de tous les éléments séquentiels. Cette machine de Moore modélise le circuit par $\mathcal{M} = \langle \Sigma_s, s_0, \Sigma_l, \Sigma_g, \delta, \lambda \rangle$, où:

- $\Sigma_s = \mathbb{B}^{|\mathcal{A}_{\mathsf{seq}}|}$ est l'ensemble des états ;
- $\Sigma_{\mathrm{s},0} \subseteq \Sigma_s$ est l'ensemble des états initiaux ;
- $\Sigma_l = \mathbb{B}^{|\mathcal{A}_{\mathsf{in}}|}$ est l'alphabet d'entrée ;
- $\Sigma_g = \mathbb{B}^{|\mathcal{A}_{\mathsf{out}}|}$ est l'alphabet de sortie ;
- $\delta : \Sigma_s \times \Sigma_l \rightarrow \Sigma_s$ est une fonction de transition d'état ;
- $\lambda : \Sigma_s \rightarrow \Sigma_g$ est une fonction de sortie.

La relation entre le modèle structurel et le modèle fonctionnel est dépendant de la fonction de valuation $\mathrm{Val}_\omega : \mathcal{A} \times \mathbb{N} \longrightarrow \mathbb{B}$, qui indique la valeur d'un signal à un certain instant discret. Le type d'un opérateur définit la relation entre sa valeur et les valeurs de ses entrées. A noter que le seul non-déterminisme de Val_ω réside dans les valeurs des opérateurs du type $\mathcal{T}_{\mathsf{in}}$ et du type $\mathcal{T}_{\mathsf{seq}}$ au temps 0.

La vérification formelle d'un modèle est basée sur des propriétés formelles définissant un comportement correct du circuit. Nous utiliserons le formalisme de la logique temporelle linéaire (LTL), dont les langages PSL et SVA sont des sous-ensembles. Une propriété LTL est construite sur le modèle $\mathcal{M}$ en considérant tous les signaux de $\mathcal{A}$ comme des propositions atomiques. En particulier, nous nous intéressons au sous-ensemble $\mathcal{P}$ des propriétés LTL de sûreté [47, 57].

Une propriété qui vérifie le comportement du modèle est appelée une assertion. A l'inverse, une hypothèse (ou contrainte) est une propriété spécifiant l'environnement du système. Lorsque la propriété ϕ est satisfaite sous l'hypothèse χ dans le modèle $\mathcal{M}$, on note :

$$\mathcal{M}|\chi \vDash \phi$$

Un vérificateur de modèle a pour but d'indiquer si une propriété est bien satisfaite, en explorant le graphe fonctionnel $\mathcal{M}$. Si une propriété n'est pas satisfaite, elle est dite en échec, et s'accompagne alors d'un contre-exemple donnant une trace d'exécution du modèle depuis un état initial et jusqu'à l'échec de la propriété.

Atteindre une configuration complète du circuit

Pour tout flot de vérification, la première étape est de contraindre le système dans un contexte réaliste. Dans le cas des circuits multi-horloges, un élément crucial est la confi-

guration des modes des horloges. Un simple exemple d'arbre d'horloge est donné sur la Figure F.2. On retrouve dans un arbre d'horloge classique quatre types d'opérateurs :

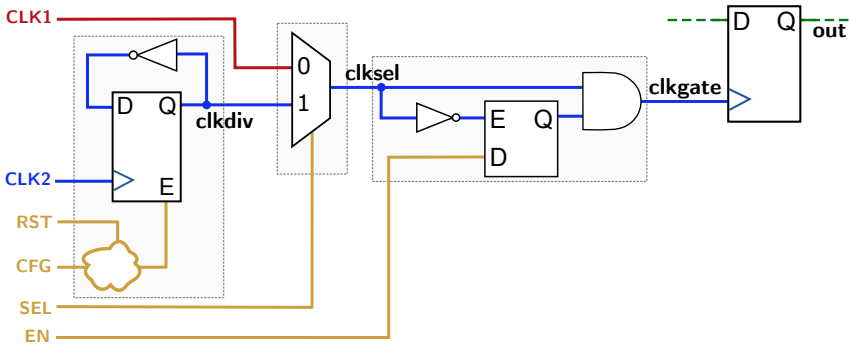


Figure F.2: Arbre d'horloge typique

sélection, combinaison, blocage, transformation. Ces opérateurs peuvent également être contrôlés par des signaux de configuration. Les ingénieurs de vérification ont alors la lourde tâche de connaître ces signaux ainsi que leur influence sur les horloges.

Afin de générer formellement cette connaissance, l'arbre d'horloge est tout d'abord structurellement identifié, et séparé en deux catégories : chemins d'horloge et signaux de configuration. Pour identifier ces différents composants, les horloges primaires sont propagées en avant sur le modèle structurel $\mathcal{D}$ jusqu'à atteindre les mémoires du système. Ensuite, les signaux en frontière de ces chemins d'horloge sont propagés en arrière jusqu'aux registres de configuration, ou aux entrées primaires.

Un signal est considéré comme ayant un comportement d'horloge s'il passe de 0 à 1 infiniment souvent. Une propriété de vivacité peut alors être écrite pour vérifier ce comportement. De plus, dans un mode de fonctionnement statique, les signaux de configuration sont constants. Une hypothèse formelle peut alors également être écrite pour décrire cette contrainte.

Afin de générer un mode dans lequel l'horloge a un comportement correct, la propriété de vivacité est inversée. Un vérificateur de modèle cherchera alors à prouver qu'il n'existe aucun mode sous lequel un signal donné a un comportement d'horloge. Si cette preuve est en échec, un contre-exemple est obtenu indiquant un mode correct de l'horloge. En ajoutant désormais une contrainte empêchant les signaux de configurations de prendre les mêmes valeurs que dans ce mode, le vérificateur de modèle génèrera un nouveau mode. Ce procédé peut être répété jusqu'à obtenir une preuve, et donc une description fonctionnelle de tous les modes de l'arbre d'horloge.

Ces modes peuvent alors être utilisés par les ingénieurs comme une spécification de leur arbre d'horloge. Un mode peut aussi être sélectionné afin de vérifier d'autres propriétés fonctionnelles du système dans un contexte réaliste. Comme le montre la Figure F.3, l'ingénieur est également aidé d'une version schématique de l'arbre d'horloge.

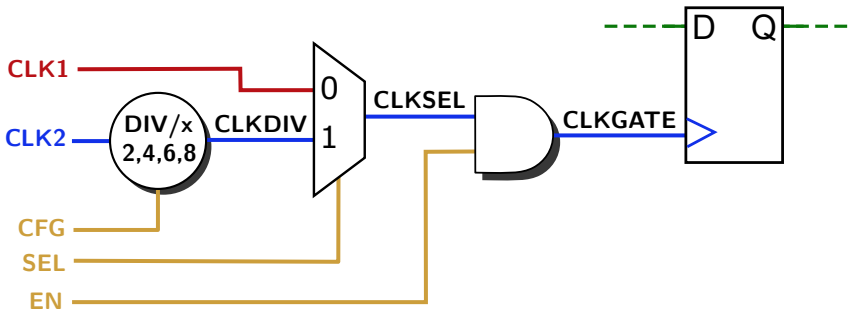


Figure F.3: View simplifiée de l'arbre d'horloge typique

Eviter une explosion de l'espace des états

L'expérience montre que la vérification de modèle voit ses limites sur les circuits industriels, trop complexes, ce qui entraîne une explosion de l'espace des états de $\mathcal{M}$. Il est en effet possible qu'aucun résultat ne soit obtenu par l'algorithme de vérification de propriété, même après plusieurs jours. Les ingénieurs de vérification cherchent alors à manuellement réduire la taille du système analysé, jusqu'à obtenir un résultat conclusif. Cependant, cette méthode demande beaucoup de temps ainsi qu'une très bonne connaissance du circuit, ce qui n'est pas réaliste dans un contexte industriel.

Elle peut toutefois être automatisée en utilisant des abstractions localisées du circuit, ce qui sur-approxime son fonctionnement et réduit l'espace des états. Une preuve de la propriété sera alors valide pour le circuit non-abstrait, alors qu'un échec génèrera un contre-exemple. La technique de raffinement d'abstraction guidée par contre-exemple utilise ce principe.

Tout d'abord, des heuristiques définissent une frontière d'abstraction minimale autour de la propriété. La trace d'exécution est analysée afin d'en déduire les signaux ayant participé à l'échec de la propriété. L'ingénieur de vérification est également consulté afin de catégoriser ces signaux et de potentiellement définir leur comportement en suivant des motifs préétablis. La frontière de l'abstraction du circuit est alors étendue sur ces signaux, ce qui précise leur comportement. Lorsqu'aucun signal à la frontière de l'abstraction n'a causé l'échec, ou lorsque l'ingénieur décide d'arrêter les itérations, l'algorithme lui génère un contre-exemple local. Cela permet de procéder à un débogage plus poussé, et de corriger le circuit ou bien de corriger les contraintes.

L'exercice de ce flot (visible sur la Figure F.4) dans un circuit industriel montre que 100% des propriétés de synchroniseurs CDC peuvent être prouvées, alors qu'un flot de vérification sans abstraction ne prouve que 63% des propriétés. Cela montre l'efficacité des abstractions dans le cas des protocoles très localisés sur le circuit.

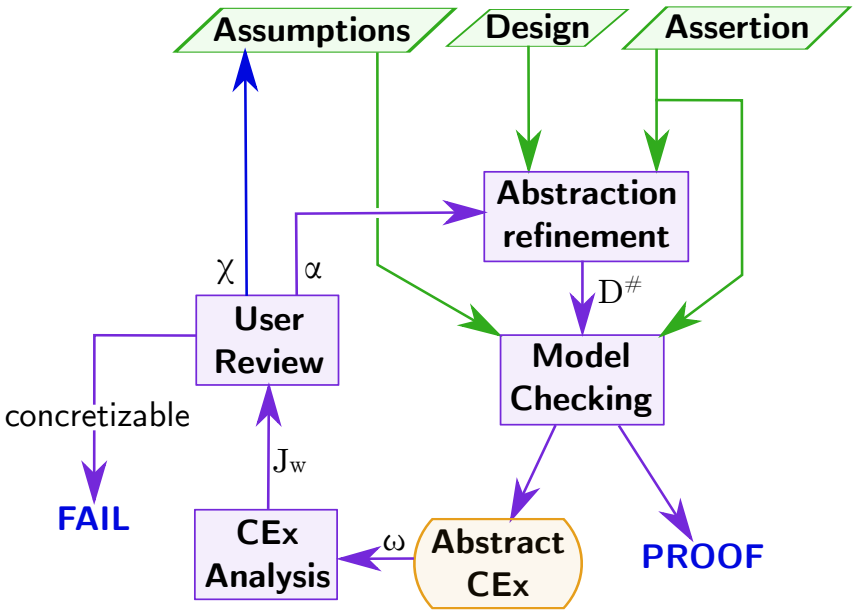


Figure F.4: Flot d'abstraction guidé par l'utilisateur

Corriger les circuits sous-contraints

Même après avoir résolu le problème d'explosion de l'espace des états, le travail de l'ingénieur n'est pas terminé. La vérification formelle explorant tous les états, un ingénieur se retrouve souvent face à de nombreux contre-exemples qui ne sont pas causés par un bogue mais par un comportement irréaliste du système. Des hypothèses formelles doivent alors être appliquées pour contraindre le système dans un comportement réaliste. Dans la plupart des cas, l'ajout de contrainte supplémentaire suffit à obtenir un résultat réaliste, mais cela demande une expertise et un temps considérable à l'ingénieur.

Afin d'augmenter la quantité d'informations disponibles sur l'échec d'une propriété, plusieurs contre-exemples vont être générés pour une même propriété. A chaque fois qu'un contre-exemple est obtenu, on cherche à en trouver un autre en empêchant le circuit de prendre les valeurs déjà obtenues dans les précédents contre-exemples. Une contrainte est ainsi construite en analysant quels signaux ont participé à l'échec de la propriété, et quels états de l'exécution bégaient. Cela permet d'obtenir des contraintes suffisamment fortes pour générer des contre-exemples très différents les uns des autres.

Les valeurs des signaux sont alors extraites des contre-exemples afin d'en déduire des motifs de comportements. Une correspondance est réalisée entre ces motifs et des modèles trouvés dans les spécifications formelles des protocoles courants Typiquement, si deux signaux montent en même temps dans tous les contre-exemples, il y a de fortes chances pour qu'ils révèlent la raison de l'échec de la propriété.

Ces motifs intéressants sont alors transformés en contraintes formelles permettant d'éviter le comportement des signaux correspondant. De multiples heuristiques sont utilisées pour filtrer ces contraintes, et ne garder que celles ayant le plus de chance de mener

à un comportement réaliste.

Le flot global de la génération de contre-exemple et de l'extraction de contrainte est visible sur la Figure F.5. Sur un circuit industriel, cette technique a réussi à générer de manière totalement automatique une contrainte manquante sur un protocole de type AMBA, et ce dans un temps raisonnable.

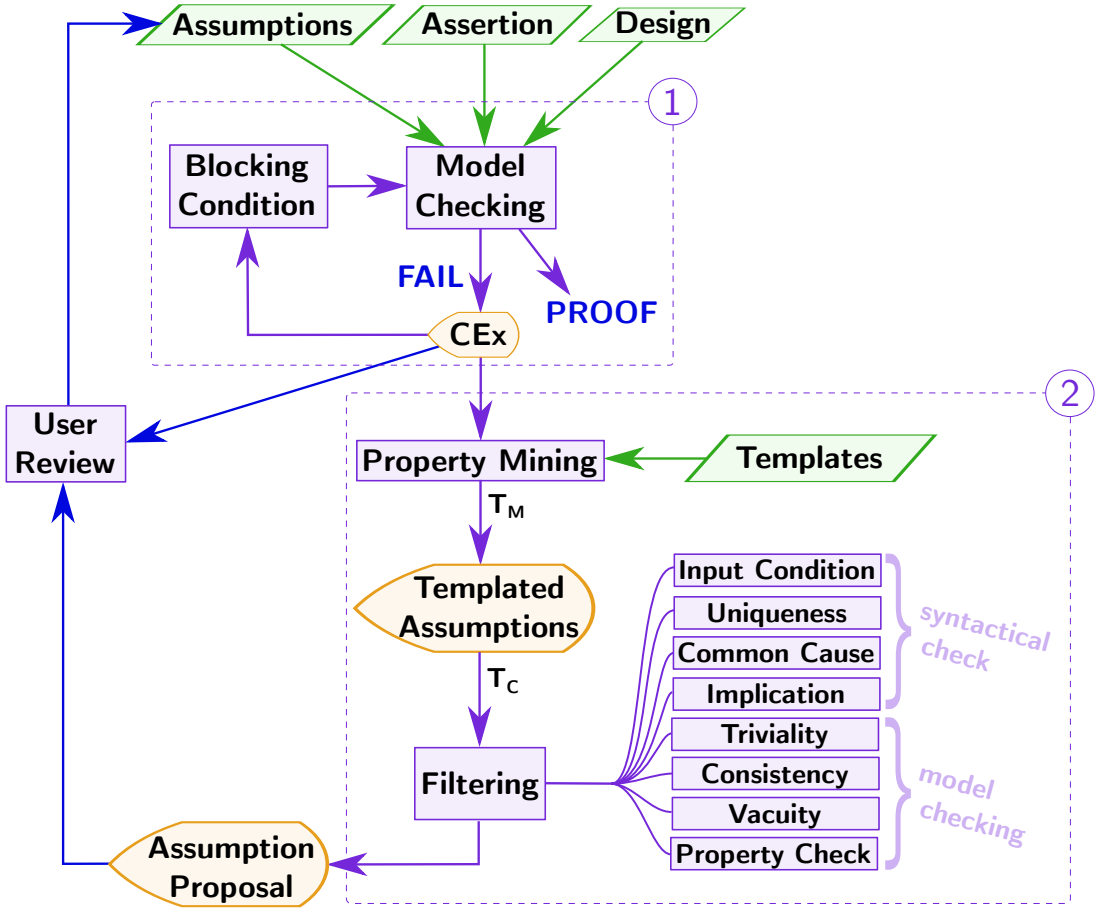


Figure F.5: Flot de génération des hypothèses formelles

Conclusion

La vérification formelle des propriétés des systèmes multi-horloges est un problème bien trop complexe pour être résolu par une vérification de modèle traditionnelle. Cette thèse alors propose plusieurs solutions pratiques, directement utilisables par des ingénieurs de vérification. Chaque problème soulevé dans le flot de vérification a été adressé par une technique distincte : la configuration des horloges, la vérification de propriétés, le débogage de spécifications incomplètes.

Chacune s'intégrant dans le flot de vérification habituel, il en résulte un flot amélioré (voir Figure F.6). La clé de ce flot est la combinaison de la rapidité de l'analyse structurelle

avec l'exhaustivité des méthodes formelles. Cette dualité est ainsi visible sur toutes les techniques développées, en utilisant les formalismes du graphe structurel et de la machine d'états. Ainsi, le flot obtenu permet d'atteindre des résultats formels conclusifs sur des circuits industriels multi-horloges, et ce tout en configurant formellement le circuit dans un mode réaliste.

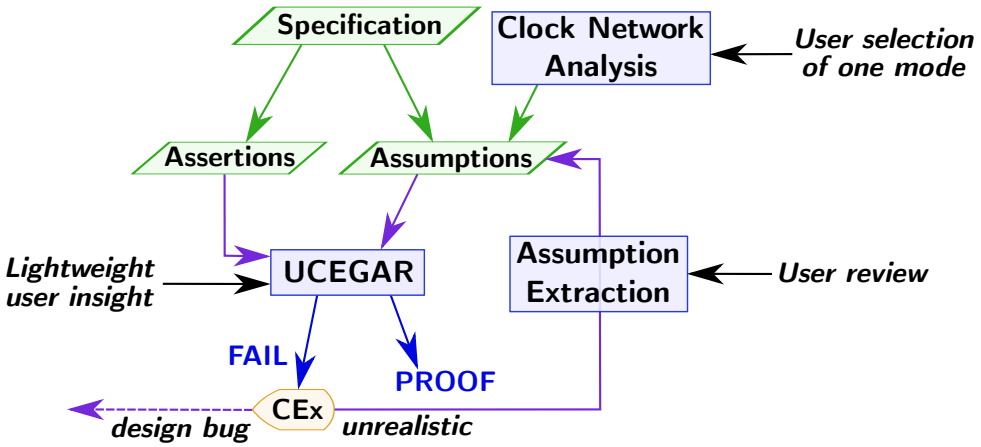


Figure F.6: Flot amélioré pour une vérification formelle conclusive

Conclusive Formal Verification of Clock-Domain Crossing Properties

Abstract — Modern hardware designs typically comprise tens of clocks to optimize consumption and performance to the ongoing tasks. With the increasing number of clock-domain crossings as well as the huge complexity of modern SoCs, formally proving the functional integrity of data propagation became a major challenge. Several issues arise: setting up the design in a realistic mode, writing protocol assumptions modeling the environment, facing state-space explosion, analyzing counter-examples, ...

The first contribution of this thesis aims at reaching a complete and realistic design setup. We use parametric liveness verification and a structural analysis of the design in order to identify behaviors of the clock and reset trees. The second contribution aims at avoiding state-space explosion, by combining localization abstractions of the design, and counter-example analysis. The key idea is to use counterexample-guided abstraction refinement as the algorithmic back-end, where the user influence the course of the algorithm based on relevant information extracted from intermediate abstract counterexamples. The third contribution aims at creating protocol assumptions for under-specified environments. First, multiple counter-examples are generated for an assertion, with different causes of failure. Then, information is mined from them and transformed into realistic protocol assumptions.

Overall, this thesis shows that a conclusive formal verification can be obtained by combining inexpensive structural analysis along with exhaustive model checking.

Keywords — formal methods, model checking, verification, static analysis, SoC, CDC.

Vérification Formelle Concluante des Propriétés des Systèmes Multi-Horloges

Résumé — Les circuits microélectroniques récents intègrent des dizaines d'horloges afin d'optimiser leur consommation et leur performance. Le nombre de traversées de domaines d'horloges (CDC) et la complexité des systèmes augmentant, garantir formellement l'intégrité d'une donnée devient un défi majeur. Plusieurs problèmes sont alors soulevés : configurer le système dans un mode réaliste, décrire l'environnement par des hypothèses sur les protocoles, gérer l'explosion de l'espace des états, analyser les contre-exemples, ...

La première contribution de cette thèse a pour but d'atteindre une configuration complète et réaliste du système. Nous utilisons de la vérification formelle paramétrique ainsi qu'une analyse de la structure du circuit afin de détecter automatiquement les composants des arbres d'horloge. La seconde contribution cherche à éviter l'explosion de l'espace des états en combinant des abstractions localisées du circuit avec une analyse de contre-examples. L'idée clé est d'utiliser la technologie de raffinement d'abstraction guidée par contre-exemple (CEGAR) où l'utilisateur influence la poursuite de l'algorithme en se basant sur des informations extraites des contre-exemples intermédiaires. La troisième contribution vise à créer des hypothèses pour des environnements sous-contraints. Tout d'abord, plusieurs contre-exemples sont générés pour une assertion, avec différentes raisons d'échec. Ensuite, des informations en sont extraites et transformées en hypothèses réalistes.

Au final, cette thèse montre qu'une vérification formelle concluante peut être obtenue en combinant la rapidité de l'analyse structurelle avec l'exhaustivité des méthodes formelles.

Mots-clés — méthodes formelles, vérification de modèle, analyse statique, SoC, CDC.

ISBN: 978-2-11-129240-6