Xprova: Formal Verification Tool with Built-in Metastability Modeling

Ghaith Tarawneh and Andrey Mokhov
School of Electrical and Electronic Engineering, Newcastle University
Newcastle upon Tyne NE1 7RU, United Kingdom
{ghaith.tarawneh, andrey.mokhov}@ncl.ac.uk

***Abstract*—This paper presents Xprova, an open-source formal verification tool for multi-clock designs. Xprova is a model checker that can discover property violations caused by the incorrect implementation of clock domain crossing circuits. Unlike existing clock domain crossing verification tools, Xprova does not rely on structural or functional analysis to detect deviations from standard design practices. Instead, it transforms the input circuit to model the onset and propagation of metastability digitally, then conducts a state space exploration to search for property violations. This approach is intrinsically capable of identifying several well-known clock domain crossing problems including missing synchronizers, path reconvergence issues and glitches. It also improves debuggability by generating counter-example waveforms showing the onset and mechanics of metastability-induced design failures. We discuss the features, underlying methodology and implementation of the tool then present use cases to compare it to commercial alternatives.**

I. Introduction

Most modern digital systems contain tens to hundreds of voltage and frequency domains, some massively-parallel architectures even thousands. Transferring signals between these domains is known as Clock Domain Crossing (CDC) and is a well-recognized thorny design area. This is because several assumptions that are often taken for granted in synchronous logic do not apply at clock domain boundaries. Most notably, asynchronous signal transitions cannot be constrained relative to their receiving clocks. Consequently, these transitions can violate the setup-hold time constraints of recipient flip-flops and drive them into metastable states. The metastable flip-flops can then latch non-deterministic values and propagate timing violations further, to their own destinations, resulting in various types of failures. To avoid such caveats, clock boundary logic must be carefully designed. In practice, this involves implementing a stereotypical design pattern that is correct by construction, such as synchronization.

Designing correct clock boundary logic is difficult because conventional EDA tools have limited support for modeling timing violations, metastability and their effects in digital simulation. Digital tools often model flip-flops as idealized storage elements with discrete binary values, an abstraction that does not capture the anomalous behavior of metastable flip-flops and its consequences [1]. Multi-clock designs may therefore show no issues when simulated and verified using conventional tools, but still fail once implemented in silicon.

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

This paper presents Xprova,[1] a model checker that attempts to address clock boundary verification at a fundamental level. Similar to other model checkers, Xprova explores the state space of an input design to either verify or disprove functional assertions that describe the design's intended behavior. However, it can also model and simulate the onset, propagation and effects of metastable states in synchronous circuits, using the methodology presented in [2]. It can therefore discover property violations caused by the incorrect handling of metastability and other timing hazards at clock domain boundaries. This approach is fundamentally different from the structural and functional analysis techniques used by other CDC verification tools. In comparison, Xprova offers the following advantages:

(1) *Zero configuration*: it is agnostic to how signals are transferred between clock domains and does not require information about what synchronization scheme, handshaking mechanism or other design patterns are used;

(2) *Fewer false positives*: it reports fewer false positives because it does not rely on structural and functional rules-of-thumb that have plenty of exceptions in practice;

(3) *Generality*: it can verify non-stereotypical designs, including those that violate standard practices [3];

(4) *Debuggability*: when a property violation is discovered, it generates a counter-example showing the issue in simulation waveforms, making it easier for the designer to understand and debug the root cause;

(5) *No built-in knowledge*: it is intrinsically capable of recognizing several well-known clock boundary issues including missing synchronizers, path reconvergence, data incoherency and crossover path glitches, without built-in assumptions about what constitutes a correct design.

Xprova is an open-source tool, developed in Java and available under the MIT license. It supports the Verilog 2001 standard and a property language similar to SystemVerilog Assertions (SVA). This paper presents the tool and compares it with commercial alternatives. Section II provides necessary background on existing CDC verification approaches while Section III discusses the features and implementation details of the tool. Section IV concludes by presenting two use cases.

[1] Available on https://github.com/xprova

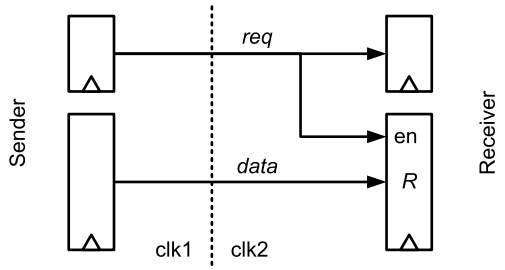


Figure 1. Missing synchronizer: an unsynchronized crossover signal `req` is used to enable a data register R.

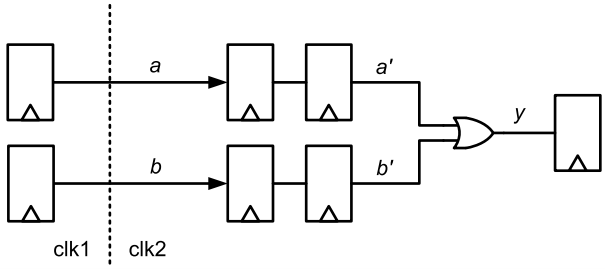


Figure 2. Path reconvergence: simultaneous changes in a and b (at clk1) may be copied to a' and b' in different cycles (of clk2).

II. Background

The growth in the number of on-chip clock domains during the past decade lead to the development of specialized EDA tools to help designers avoid metastability and other clock boundary design caveats. Today, many such tools exist including Questa CDC by Mentor Graphics, SpyGlass CDC by Synopsys and Conformal CDC by Cadence. These tools differ in features but share a common underlying scheme; they check designs against lists of pre-programmed rules and design patterns and issue warnings when deviations from standard practices are detected. This section reviews this approach with the aim of establishing a baseline for comparison with metastability simulation in Xprova.

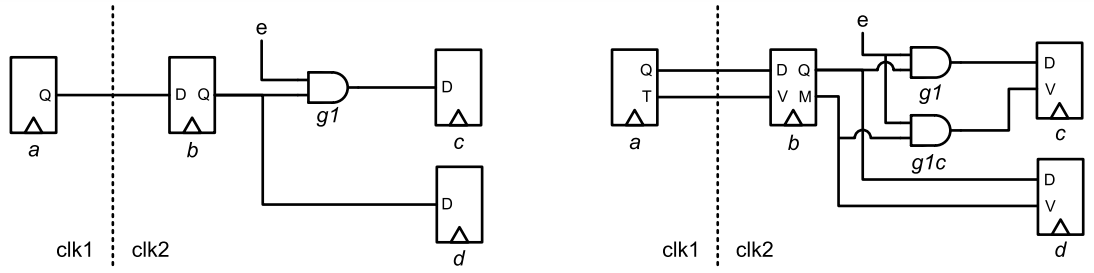
A. Structural Analysis

Clock boundary issues can often be identified by analyzing the circuit structure alone. Consider, for example, the circuit in Figure 1. Here, a sender sets up a value on `data` and asserts `req` to initiate a handshake. On the receiver's side, `req` is used as an enable signal to latch `data` in a local register R. This is a well-recognized problematic implementation; the unbounded arrival time of `req` means that different bits of R may become metastable if `req` arrives too close to the clock edge. When this happens, some bits of R may succeed to capture the respective `data` bits while others retain their old values, effectively corrupting the stored value. This can be prevented by piping `req` through a flip-flop chain (i.e. a synchronizer) before using it to latch `data`. In principle, this and similar cases can be recognized automatically by an EDA tool that analyzes the structure of the circuit and looks for crossover signal buses that are not multiplexed by an accompanying pipelined (control) signal.

In practice, it is difficult to tell from circuit structure alone if the data is being synchronized correctly, or if the lack of synchronization is a problem in the first place. The circuit in Figure 1, for example, can function correctly without a synchronizer if `data` and `req` were part of a quasi-stable configuration word that is initialized during system startup and remains static at runtime. In fact, the design may still be correct even if `data` bits were to change at runtime, so long as `data` changes by a single-bit and the output of R is subsequently synchronized (a common design pattern in asynchronous FIFOs with Gray-coded read and write pointers). Structural analysis tools prefer to err on the safe side and therefore rely on conservative rules of thumb to make sure no issues go undetected. As a result, more correct circuits are suspected and get marked for the attention of the designer as containing potential issues. The toll on verification time can be hefty if one considers that applying structural analysis to a commercial SoC can generate 100s of thousands of clock boundary warnings, 90% of which are false positives due to tool misconfiguration and non-standard design cases [4].

B. Functional Analysis

Taking the functional properties of the design into account can help resolve ambiguities concerning whether certain clock boundary circuit structures pose an actual problem or not. Tools use functional analysis to detect "path reconvergence" issues for instance, a scenario illustrated in Figure 2. Here, two bits a and b are synchronized independently and used to compute $y = a' + b'$, (where a' and b' are the synchronized copies). As is the case with missing synchronizers, this too is a non-standard design pattern with possibly problematic consequences. Although a and b are both synchronized, synchronizers have non-deterministic latencies and so simultaneous changes in a and b may be copied to a' and b' in different receiver clock cycles, an issue referred to as *data incoherence*. Has this been taken into consideration when designing the circuit or did the designer incorrectly assume that $a' = b'$? Again, most tool prefer to err on the safe side and therefore issue a warning if combinational paths from a' and b' to a single destination flip-flop can be sensitized at the same time (a condition that is satisfied in our example). The intuition behind this assumption is that path sensitization indicates that both operands are "in use" at the same time and so one cannot rule out the possibility that their incoherence may cause a problem. This of course is another conservative generalization with its own exceptions. Just because a and b are used simultaneously does not imply that the design will necessarily fail. For example, y itself may be logically masked in all the cases where a and b transition simultaneously. In general, it is difficult to tell whether data incoherence represents a true issue without considering the design's specification. Functional analysis can therefore aid in resolving certain ambiguities but does not eliminate them completely.



(A) Original Circuit

(B) Transformed Circuit

Figure 3. An example of the circuit transformation applied by Xprova. First, flip-flops are replaced with model cells that communicate timing violation information using additional pins (`T`, `V` and `M`). Copies of combinational paths are then inserted to model the propagation of timing violations between flip-flops and logical masking by local (stable) signals.

III. Features and Implementation

This section presents Xprova and discusses some of its features and implementation details. We compare the methodology used by the tool against structural and functional verification in Section IV, using concrete circuit examples.

A. Overview

Xprova is a Verilog model checker that can prove/disprove assertions expressed in an SVA-like property language. It has a command line interface and can be used either interactively or by calling script files. The following script demonstrates the basic steps in a typical verification run:

```
# load user cell library
library load gates90nm.v

# load metastable flip-flop models (internal library)
library load lib/xprova.v

# load design
read -b examples/fifo/fifo.v

# define assumptions and assertions
assume full |=> ~write
assume empty |=> ~read
assert ~(empty & full)
assert write |=> ~empty
assert read |=> ~full

# attempt prove
prove
```

Similar to other model checkers, assumptions specify regions of the state space that are of interest while assertions express how the design is intended to behave. The tool performs an exhaustive state space exploration in the search for states where all assumptions hold but an assertion does not. If no such states are found, the tool indicates that the assertion was proven. Otherwise, it prints a counter-example showing a sequence of inputs that advances the design from reset to a state where the assertion is violated.

Xprova's core functionality is provided by a circuit transformation, described next, that augments the input design before passing it to the internal model checker. The transformation enables the circuit to simulate metastability effects digitally.

B. Modeling Metastability

The simulation of metastability effects is performed automatically and in a manner that is transparent to the user. The tool recognizes multi-clock designs and modifies them by:

(1) replacing flip-flops with model cells that can simulate setup/hold time violations, non-deterministic outputs and clk-to-q delay violations, and
(2) adding additional combinational circuit elements to model the transfer of metastability between flip-flops.

We discuss this transformation using the circuit in Figure 3 as a working example while referring readers to [2] for a thorough discussion on the methodology. The purpose of the transformation is to enable flip-flops to communicate information about timing violations and thus model the onset, propagation and effects of metastability. This is done using the additional signals:

(1) `T`: output transition,
(2) `V`: setup-hold time violation, and
(3) `M`: metastable output.

The behavior of the transformed circuit can be explained as follows. When the value stored in a changes, its `T` output is asserted to indicate an output transition. On the first subsequent edge of clk2, b's `V` input is asserted, indicating that b's setup-hold time conditions have been violated. b then asserts it's `M` output to indicate that it is metastable and "propagates" its metastable state to d, with whom it shares an `M` to `V` connection.

The connection between b and c is slightly more complex due to the presence of combinational logic (gate $g1$). Here, we wish to model the logical masking of b's metastable state by the (stable) signal e. We do this by first creating a copy of $g1$ ($g1c$) then passing to it an unknown bit (**`X`**) as b's active state. If the output of $g1c$ is **`X`** then the combinational path from b to c is sensitized and metastability can propagate between the two. On the other hand, if the output of $g1c$ is either 0 or 1 then the metastable output of b has been masked. Since the signals `T`, `V` and `M` are used to test logical masking between flip-flops, they all, in fact, use an "active-**`X`**" encoding where **`X`** represents an active state while $0/1$ represent an inactive state.

In reality, of course, metastable states do not rise and propagate between vulnerable flip-flops on *each* cycle. To model

the non-deterministic properties of metastability, each model flip-flop has two internal signals $r1$ and $r2$ that determine (1) whether an asserted `V` causes metastability and (2) the value of `Q` if the flip-flop becomes metastable. Xprova extracts all $r1$ and $r2$ signals as top-level inputs to the design and includes them in its state space exploration model. It can therefore enumerate all possible sequences of metastability onset and propagation for a given design. This approach exploits the power of formal verification to discover property violations resulting from otherwise-obscure chains of metastable events.

C. Property Language

Xprova's property language will feel familiar to users of SystemVerilog Assertions (SVA) and Property Synthesis Language (PSL). The following script shows some supported operators and built-in functions:

```
# basic logical operators:
assert w = (x & y) | ~(z ^ y)

# relational operators:
assert (data == v1) & (data != v2)
assert (data >  v1) & (data <  v2)
assert (data >= v1) & (data <= v2)

# implication:
assert x |-> y
assert x |=> y
# (x implies y on the same/following cycle)

# signal level changes:
assert $rose(x) | $fell(y) | $stable(z) | $changed(w)

# signal level history:
assert $always(x) | $never(y)

# bit reduction:
assert $any(x) | $all(y)

# past and future value referencing:
assert (x == @2 y) & (x == #3 z)
# (x equals y two cycles before, and z three cycles after)

# sequences:
assert x ## y ## z
# (x followed by y and z, each on a consecutive cycle)
```

Xprova can also verify liveness assertions, expressed using the `$eventually` built-in function:

```
# liveness assertion:

assert $eventually(trigger, expr)

# (once trigger is true, expr must become true, eventually)
```

D. Verification Flow

The verification flow using Xprova is illustrated in Figure 4. The tool first augments the input design by substituting flip-flops with model cells and inserting combinational path duplicates, as described in Section III-B. Property circuits are then synthesized and included in the netlist. Xprova subsequently generates a model of the circuit (in either C++ or Java, based on user's choice), consisting of a cycle-based simulator and a state space exploration function. The model is compiled, executed and, upon termination, prints the status of each assertion, either pass (proven) or fail (disproven). For each disproven assertion, Xprova generates a counter-example waveform in VCD format. Counter-examples are always the shortest possible for safety (i.e. non-liveness) assertions, and include the `T`, `V` and `M` signals of the implicated flip-flops to help users trace the cause and mechanics of the violation.

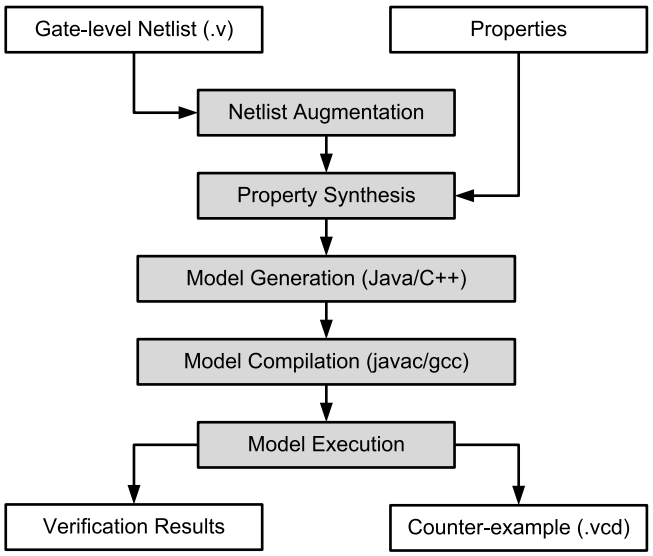


Figure 4. Typical verification flow using Xprova (white boxes are inputs/outputs and gray boxes are processing steps).

Xprova can export the augmented netlist as a Verilog file, giving users the freedom to verify it using external formal verification tools. This option is provided because the metastability-modeling feature provided by the tool is not tied to exhaustive state space exploration as a formal verification technique. Users may therefore export their augmented designs and verify them using bounded-model checkers, theorem provers or other formal techniques. External verification tools must support three-valued logic, however, since unknown bits are used by the augmented circuit to model the propagation of timing violations between flip-flops.

In addition to external formal verification tools, Xprova integrates with several free and open-source EDA tools. It can invoke `yosys` [5] behind the scenes to synthesize behavioral designs prior to verification, optionally during the loading step (using the switch `-b` of the `read` command).[2] Counter-examples can be opened automatically in `gtkwave` [6] and various netlist representations can be exported as graphs in DOT format.

[2]Many clock boundary issues exist at the netlist abstraction level and can be verified only once the behavioral design has been synthesized (Section IV-B presents an example of an issue introduced during synthesis). Xprova supports loading behavioral designs but users should be aware that, for behavioral designs, verification results will apply to the netlists synthesized by yosys during the loading step (which can be exported using the `write` command).

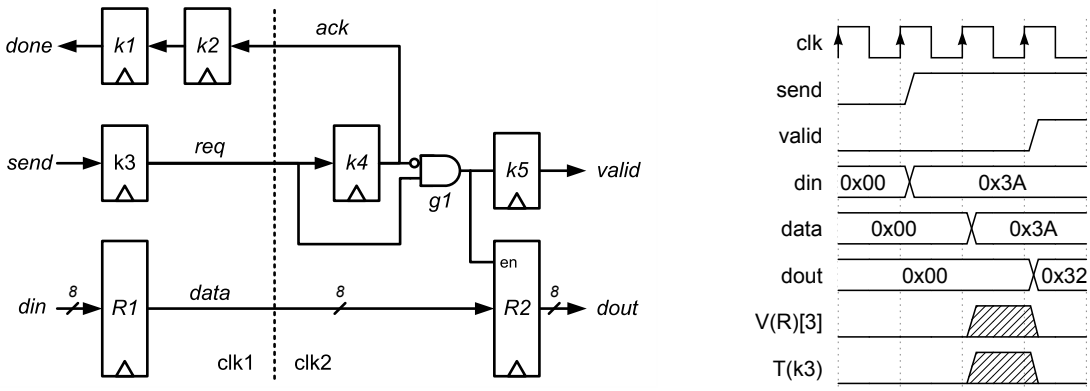


Figure 5. (Left) A circuit to transfer data items between clock domains, missing a synchronizer on the receiver side. (Right) Counter-example generated by Xprova, showing an assertion violation caused by the missing synchronizer (mismatch between sent and received data items).

IV. Examples

This section presents examples of multi-clock circuits in which errors have been made when designing clock interface logic. We demonstrate the use of Xprova by attempting to prove certain assertions describing each design and debugging the resulting counter-examples.

A. Example 1: Missing Synchronizer

In the first example (Figure 5), a circuit is provided to transfer data items across a clock domain boundary. At the interface level, the circuit is intended to operate as follows: (1) the sender sets up `din` then asserts `send`, (2) `valid` is then asserted when `dout` contains a new data item and (3) `send` must be kept high during the transfer, until `done` goes low.

Internally, the circuit relies on a four-way handshake and uses a logic gate g_1 to enable the receiving register $R2$ when `req` goes high. This transfer mechanism would operate correctly if not for the absence of a synchronizer at the receiver's side (as shown in Figure 5, `req` is used directly to enable $R2$ without being piped through a synchronizer chain). We assume that this issue is not apparent at a first glance and proceed to verify the circuit by describing its behavior using the following Xprova properties:

```
# Assumption 1:
assume (req & !done) |-> send

# Assertion 1:
assert valid |-> ($when($rose(send), din) == dout)
```

where *Assumption 1* constrains the environment such that `send` remains high during a transfer and *Assertion 1* states that the sent and received data items are equal.[3] Attempting to prove this specification using Xprova results in a fail status assigned to *Assertion 1* and generates the counter-example shown in Figure 5.

At the moment it is unclear whether the violation is caused by a typical (synchronous) design issue or a clock boundary problem. However, a quick look at the counter-example makes it immediately obvious that it is the latter. When searching for property violations, Xprova examines metastability-free state space regions first. If no violations are found, the tool proceeds to explore other possibilities, always starting with the simplest scenarios of metastability onset and propagation. Therefore, the presence of a metastable event (a propagating sequence of timing violations) in the counter-example means that this event has explicitly caused the violation. In our counter-example, the `T` pin of $k3$ and `V` pin of $R2[3]$ are in an active state in cycle 3 and are therefore implicated in causing the violation (recall that these pins use active-**X** encoding). Examining the waveform, it appears that the data item `0x3A` was received as `0x32` and that the mismatch occurred when `data` was latched by $R2$ during cycle 3. Given that `V` pin of $R2[3]$ is active on the same cycle, it is now apparent that the setup/hold time conditions of flip-flop $R2[3]$ were violated, causing the fourth bit of the data item to be latched incorrectly (0 instead of 1) in cycle 3. The setup/hold time violation is caused by the transition of $k3$ during the same cycle and is therefore an indication that `req` is used, without synchronization, as a combinational input to $R2$.

Before moving on, it is worth comparing the verification flow above with structural/functional analysis. Structural analysis is very reliable at identifying missing synchronizers and would generate a warning when inspecting this circuit. However, it will leave it to the designer to understand the circuit and confirm whether the warning is indicative of an actual issue. As discussed in Section II, this process is costly in terms of verification time since the majority of structural warnings are actually false positives. Using Xprova, on the other hand, we were able to confirm the existence of an error with minimal investment in verification time: understanding the internal behavior of the design was not required until the presence of an error has been confirmed. The tool also generated the shortest sequence of inputs and metastable events to reproduce the error and has therefore made it easier to understand and debug the violation.

[3]To define what the sent item was, we use the `$when()` built-in function to sample `din` when `send` went high.

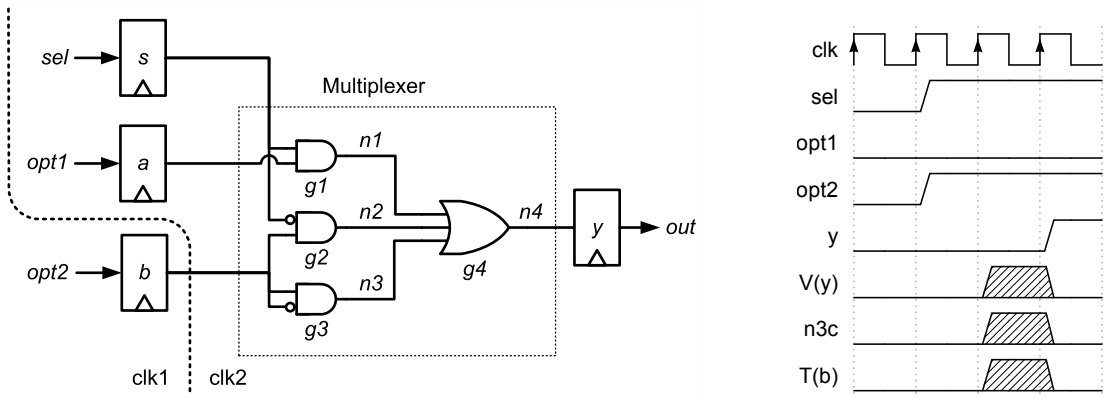


Figure 6. (Left) Circuit with a glitch hazard. (Right) counter-example generated by Xprova showing a property violation caused by the glitch.

B. Example 2: Glitches

For the second example, we consider the circuit in Figure 6, an adaptation of a circuit published in [7]. The circuit is built around a simple multiplexer with an unusual implementation detail: the output of one of its logic gates ($g3$) represents an expression that evaluates to 0 in all cases $(b \wedge \neg b)$. Gate $g3$ is therefore redundant and can be removed while converting $g4$ into a two-input OR gate. Combinational logic "artifacts" such as these are sometimes created by synthesis engines as a result of applying certain optimizations. Apart from making the circuit more efficient (usually in ways that are obvious to the synthesis engine alone), synthesis artifacts do not change the Boolean expression represented by the circuit and are therefore harmless in synchronous logic. In this circuit, however, b belongs to a different clock domain and propagating it through different combinational paths can introduce a *glitch*. The glitch occurs when b transitions; if the paths from b to the inputs of $g3$ have different delays then a logic high state may appear temporarily on $n3$ and propagate to $n4$ just by the time it is sampled by y. Interestingly, this glitch may introduce an error even when the "relevant" circuit parts are synchronous. For example, when $s = 1$ it is intuitive to think of the multiplexer as blocking the asynchronous bit b and selecting the (stable) synchronous bit a. This, however, is not the case. Even when $s = 1$, transitions of b may cause temporary glitches at the input of y.

As before we will ignore our knowledge of the circuit's issues for the purpose of illustration.[4] We assume that the circuit is provided as a black box, alongside an intended specification that includes the property:

```
# Assertion 1:
assert sel |=> (out == @2 opt1)
```

where the operator `@2` samples the operand 2 cycles earlier. After running Xprova on this circuit, *Assertion 1* is given a fail status and the counter-example in Figure 6 is generated. The counter-example shows a case where sel was high but y did not copy $opt1$ after two cycles, a contradiction of *Assertion 1*. As with Example 1, the counter-example includes few metastability modeling nets in an active state and so we can tell that the violation is caused by a metastable event. Tracing back from out, the setup/hold time conditions of y were violated on cycle 3 ($\mathtt{V}(y)$ = `X`) and this was caused by a metastable state propagating through $n3$ (its duplicate net $n3c$ is `X`) and originating from a transition of b (since $\mathtt{T}(b)$ = `X`). This is sufficient to tell what is wrong with this circuit; there is a combinational path $b \rightarrow y$ that is permitting b's transitions to arrive at y's input when sel is high. The hazard can be avoided by changing the multiplexer implementation, or better yet by adding a synchronizer at b's output. These solutions can be verified by re-running Xprova after making the changes and checking that *Assertion 1* receives a pass status.

[4]The reader may notice that the circuit is also missing a synchronizer.

V. Conclusion

We presented Xprova, a formal verification tool capable of identifying property violations caused by clock domain boundary issues. The tool offers better observability and debuggability compared to commercial alternatives based on structural and functional analysis.

References

[1] I. W. Jones, S. Yang, and M. Greenstreet, "Synchronizer Behavior and Analysis," *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, May 2009.

[2] G. Tarawneh, A. Mokhov, and A. Yakovlev, "Formal Verification of Clock Domain Crossing using Gate-level Models of Metastable Flip-Flops," *Proceedings of the 2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[3] G. Tarawneh, M. Függer, and C. Lenzen, "Metastability Tolerant Computing," *23rd IEEE International Symposium on Asynchronous Circuits and Systems*, May 2017.

[4] Y. Lee, N. Kim, J. B. Kim, and B. Min, "Millions to thousands issues through knowledge based SoC CDC verification," *2012 International SoC Design Conference (ISOCC)*, Nov 2012.

[5] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[6] T. Bybell, "GtkWave electronic waveform viewer," 2010.

[7] Y. Peng, I. W. Jones, and M. R. Greenstreet, "Finding Glitches Using Formal Methods," *ASYNC 2016*, May 2016.