2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
OCTOBER 26-27, 2021

Formal Property Verification of the Digital Section of an Ultra-Low Current Digitizer ASIC

Katharina Ceesay-Seitz, Verifiation Engineer, CERN, Geneva, Switzerland
(*katharina.ceesay-seitz@cern.ch*)
Sarath Kundumattathil Mohanan, Design Engineer, CERN, Geneva, Switzerland
(*sarath.mohanan@cern.ch*)
Hamza Boukabache, Project Manager, CERN, Geneva, Switzerland (*hamza.boukabache@cern.ch*)
Daniel Perrin, Section Leader, CERN, Geneva, Switzerland (*daniel.perrin@cern.ch*)

***Abstract*—This paper details our experience with Formal Property Verification (FPV) of the digital section of a mixed-signal Application Specific Integrated Circuit (ASIC) for ultra-low current measurements. The ASIC was developed as a prototype front-end for the future version of the CERN RadiatiOn Monitoring Electronics (CROME), which is a safety-critical system. The main functionality could be formally proven even though the design contained several counters. A large number of faults could be discovered and removed. The paper aims to demonstrate FPV with SystemVerilog Assertions on a concrete example to give the reader an idea whether and how FPV can be applied to similar designs.**

Keywords—Formal Property Verification (FPV); SystemVerilog Assertions (SVA); ASIC verification; digital electronics

I. INTRODUCTION

The usage of formal verification for digital electronics has grown over the past 10 years. Currently around 40% of ASIC projects and around 20% of FPGA projects are using the technology [1]. The majority of the verification teams is still not using it because it is often not clear whether the Return on Investment (ROI) will be sufficient. For verification teams which are new to formal verification it is hard to determine whether a design is suitable for formal verification and to estimate the effort. The paper aims to support this decision by sharing our experience, the challenges we faced, the techniques we used and to report on the effort, obtained results and insights.

The CERN (European Organization for Nuclear Research) Radiation Protection Group is responsible for monitoring the levels of ionizing radiation on all CERN sites and for ensuring the radiological safety of the persons working there or living in the area. This includes monitoring of the experimental service areas close to the physics experiments as well as environmental monitoring at the boundaries of the CERN sites. The new CERN RadiatiOn Monitoring Electronics (CROME) is developed as a flexible and modular system that will gradually replace all existing monitoring systems in the different zones. Therefore, the system must be able to measure a wide range of mixed-field radiation. Currently it uses a commercially available circuit for current measurements from femto- to nanoampere [2]. Prototypes for a new ASIC based front end have been designed in order to further improve the sensitivity. The Design Under Verification (DUV) of this paper is the digital section of a mixed-signal version of this ASIC. It is a current digitizer that is capable of measuring current from femto- to microampere [3].

Figure 1 summarizes the architecture of the design. It consists of two measurement channels, where Channel 1 performs the main measurement while Channel 2 measures leakage currents which are used to compensate the results from Channel 1. Both channels consist of analogue measuring circuits that pass comparator outputs to the digital section of the chip. These outputs are synchronized on the digital side and used by several kinds of pulse

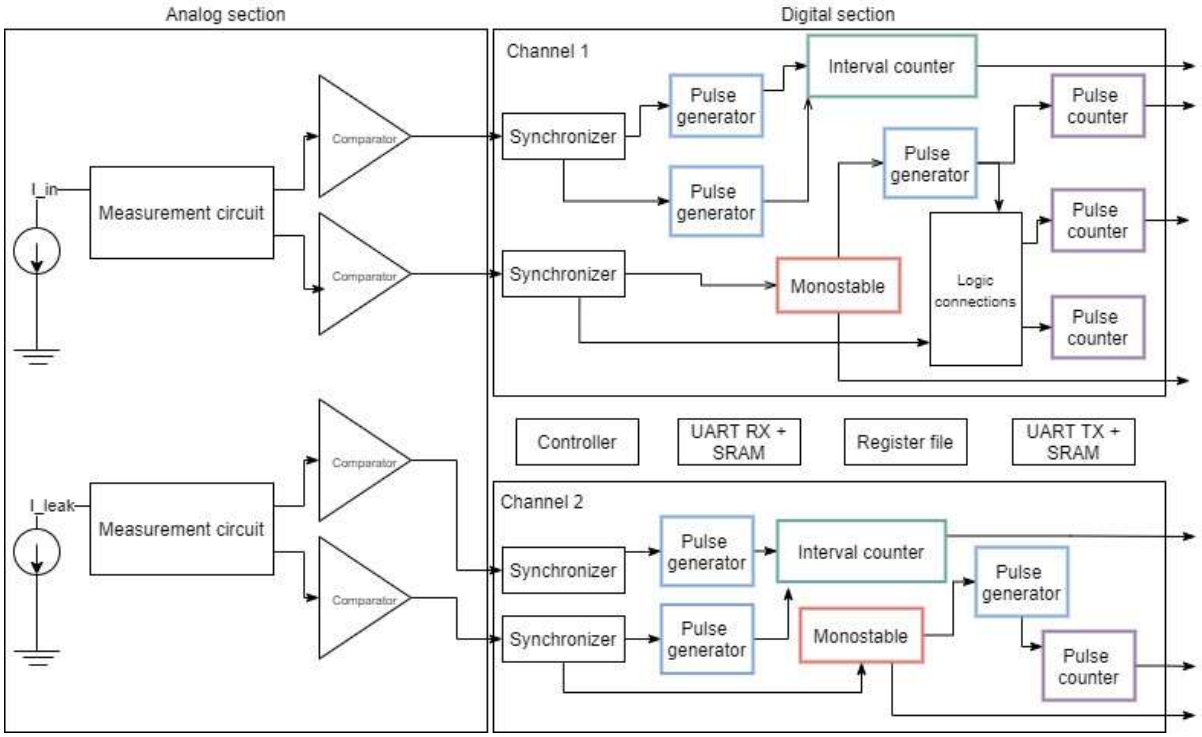

Figure 1. System Architecture of the Mixed-Signal Design [3]

and interval counters to measure the current. The resulting counts are stored in an SRAM and can be read out via an UART. The final current value is determined by the application that reads from the UART. Several parameters of the chip can be configured during operation via the UART as well.

II. Related Work

Formal verification is often used for protocol verification, Finite State Machines (FSM) and designs with short sequential depth. Counters and memories are typically abstracted away in order to avoid state space explosion [4]. In our case we had to verify the counters themselves, abstractions could not always be used. UART controllers FSMs are a typical candidate for formal verification [5]. Verifying that data passes through a design as expected is harder. Solutions with symbolic data tracking values have been presented in [10]. Very often formal verification is used for parts of the design or for bug hunting [4, 5]. Recently, formal coverage metrics gained attraction for sign-off of System-on-Chip (SoC) ASICs with formal verification [6] or FPGA verification [8].

III. Methodology

The verification of the current digitizer was based on our functional verification methodology for safety-critical FPGA designs that we developed for the verification of CROME [8]. Formal Property Verification (FPV) with SystemVerilog Assertions was the main technique used. The present prototype will not yet be used in a safety-critical system, therefore it was not mandatory to use a functional safety standard compliant verification process. Nevertheless we followed our methodology, because it is very useful for planning, progress tracking and communication. In particular the frequent and early reviews of specification, verification code and documents lead to the discovery of several faults that could be removed without debugging efforts. This is a great benefit as debugging is where most of the verification time is spent [1].

A mistake in a property would affect the verification result of a large part of the functionality, because one property verifies thousands of scenarios. In simulation one would write many test cases for the same purpose and a mistake in one of them would have less severe consequences. It will most likely be found due to inconsistencies with the results of other test cases. A wrong assertion can easier go unnoticed. Therefore it is important to use quality assurance techniques such as: reviews, manual or automated mutations of the design or verification code,

expression of the same verification requirements redundantly in several properties or addition of cover properties (see section IV.E). Counter examples of failing block level properties were independently recreated in top level simulation by the designer in order to differentiate between DUV bugs, misinterpretations of requirements or faults in the verification code. This aided the designer in understanding the scenario and it validated whether abstractions were underconstraining the design. These test benches were then used by the designer to remove DUV faults. Section IV.C lists several examples of corner cases which were believed to be faults in the DUV. It turned out that they either had not been considered by the designer or they were missing in the specification. A precise and detailed specification could have avoided the debugging and correction efforts in these cases.

IV. EXPERIENCE

This section presents common use cases of formal verification based on verification of the current digitizer.

A. *Short sequential depth of the property*

Relationships between signals that span a sequential depth of a few clock cycles are very good candidates for simple properties. A common example is the relationship of outputs with an enable signal.

Example 1: The requirement "All outputs should be low if an enable signal is low" can be expressed with a simple implication, e.g.: `assert property (enable == 0 |=> (output1 == 0 && output2 == 0)).` This was proven within seconds and it lead to the discovery of some bugs mentioned in section IV.C.

Example 2: The requirement was: "For all 2^{32} possibilities of the target value and any combination with other input signals, any time the counter equals the target value, the design must generate a pulse." Instead of writing many test cases for different target values and input combinations, the requirement can be verified with the following property: `assert property (counter == targetValue |=> $rose(pulse)).` It could be proven exhaustively within some seconds.

B. *Chain of events*

Verification requirements and therefore properties can often be expressed as a chain of events as depicted in Figure 2. Some scenario on the inputs must imply an outcome which itself must imply another outcome and so on. The outcomes are sometimes expected to occur only in the case of the specified scenario and sometimes also in further cases. That's why the 'iff' operator cannot always be used.

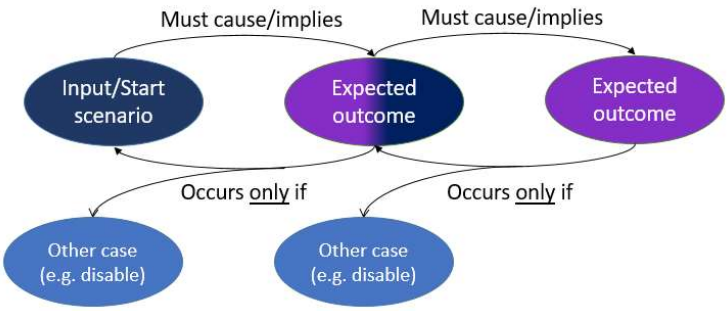

Figure 2 Chain of events

Figure 3 demonstrates such a chain for the interval counters. The **interval counter** is specified to count the number of clock cycles between a start and an end pulse if an enable signal is high. Counting happens internally and the counter output shall be loaded with a new value after a valid end pulse. Many corner cases had to be considered: The counter output is also updated after the internal counter reaches its maximum value even if there was no valid end pulse. A second start pulse arriving after counting has started shall be ignored, but after an end pulse counting shall continue and a second end pulse shall produce a further count output. A start and end pulse arriving at the same time shall set the counter output to 1 and counting shall only restart with the next start pulse. Table 1 lists the chain of requirements and corresponding properties. The first scenario is modelled as a signal assignment. 'end_pulse', 'start_pulse' and 'enable' signals are inputs, 'interval_count' and 'count_valid' are

outputs. Signals ending in 'Tb' are modelled in synthesizable SystemVerilog code. Note that P2 is needed for the corner case where the internal counter reached the same value a second time when a valid end pulse arrives.

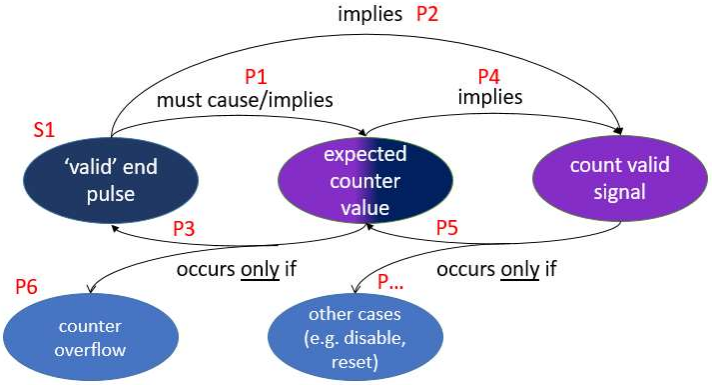

Figure 3 Chain of events - Example 1

Table 1 Chain of Events

Label	Verification requirement	Verification code
S1	An **end pulse** is valid if it coincides with or was preceded by at least one start pulse and the enable signal had been high since the first start pulse (tracked by 'countingActiveTb').	`// Signal assignment` `assign validEndPulse = (end_pulse && (countingActiveTb || (start_pulse && enable)))`
P1	**A valid end pulse** must generate the **expected counter output.**	`assert property`[1]`( validEndPulse |=> interval_count == intervalCntTb);`
P2	When there is a **valid end pulse, the count valid signal shall be asserted**.	`(validEndPulse |=> count_valid);`
P3	Verify that every time **the counter output is changed to a new value different from 0 and the maximum value,** it is preceded by a **valid end pulse.**	`(($changed(interval_count) && interval_count != 0 && interval_count != (2**`Bit_Width`)-1) |-> $past(validEndPulse));`
P4	Verify that if the **counter value is updated to something else than 0**, in the same clock cycle the **count valid signal is high.**	`(($changed(interval_count) && interval_count != 0) |-> (count_valid));`
P5	The count_valid shall rise **only** if a valid end pulse was received one cycle earlier or the interval counter equals its maximum value.	`(count_valid |-> $past(validEndPulse)) || interval_count == (2**`Bit_Width`)-1));`
P6	Verify that if the interval counter is set to the maximum value without being preceded by an end pulse, the test bench signal 'overflowCheckCounterTb' equals the maximum value as well.	`($changed(interval_count) && interval_count == (2**`Bit_Width`)-1 && $past(end_pulse) == 0) |-> overflowCheckCounterTb == (2**`Bit_Width`)-1);`

The chain can be continued by verifying that the 'count_valid' signal is a pulse if it was generated due to an end pulse (not due to an overflow). The interval counter properties can be used as input assumptions for a connected block or within proofs of functionality one level higher in the block hierarchy. This chain can be continued until a top-level output of the design. All these properties were proven exhaustively for all combinations of inputs and the full bitwidth of 40. Several bugs were found. They are part of the list in the next section.

[1] "assert property" is omitted in the following rows for better readability.

C. *Corner case bugs*

Formal verification is extremely useful for bug hunting and discovering hidden corner cases [6]. This section lists some examples of faults that were found in the current digitizer. Several of them were related to an ambiguous specification. A more precise specification could have avoided several iterations between design and verification engineers. The genuine bugs concerned mainly corner cases such as signal changes that happen simultaneously or in close proximity with changes of other signals. Many bugs were related to overflows of internal counters.

Specification ambiguities:

- It was not clearly specified until which event an enable signal had to be high. There is a legal case in which the DUV can generate an output even though the enable signal is already low.
- A corner case where the start and end pulses of the interval counter arrive in the same cycle overwrite the counter value with 1. Furthermore, counting is required to continue after an end pulse, but not in this case, as it already means that the read-out current has the maximum value.
- Very often the cycle accurate behavior of the design was not specified. Delays or signal lengths were not specified. This caused many spuriously failing assertions.

Genuine bugs – related to overflows:

- A FIFO's 'empty' signal rose when the internal data counter overflowed even though the FIFO was actually completely full.
- Another overflow mechanism did not work when the overflow happened exactly one clock cycle before one of the input signals rose.
- The overflow output of the interval counter was initially a fault but then kept as functionality, because it is useful to indicate a very low current. The fault was found when the bitwidth of the counter was reduced to 8, which caused a previously inconclusive property to fail. The functionality was proven with the properties P5 and P6 of Table 1. P5 was proven in a few seconds, while P6 needed several hours for the full bitwidth.

Uninitialized registers:

Formal verification can be run with synthesis or with simulation semantics. An 'X' in simulation means that the signal value is "unknown". This is also the case for gate level simulation. In synthesis semantics it means "don't care". In silicon an 'X' could turn into a 0 or a 1 [11]. The following example demonstrates what can happen.

```
if signal_in = '1' then
  other_signal <= '1';
else
  other_signal <= '0';
end if;
```

With simulation semantics as well as in simulation the else branch will be taken if 'signal_in' is unknown. In synthesis semantics (and also in silicon) the signal could have any value (also 1) and therefore the first branch might be taken in an unexpected case. In the current digitizer this scenario happened when some internal signals were not reset and could therefore be 'X'. A counter example for a failing property showed a case where the data that would have been sent out via the UART would have been wrong. There are also automated formal verification tools available on the market that check the propagation of 'X'.

D. *Proving counters*

Counters are often removed from a DUV and replaced with an abstract model. This is not an option if the counter and its reset logic need to be proven. This section presents three approaches that were used in such cases.

1) Counting with two clock cycles property

The current digitizer contains a 32-bit counter whose size can't be reduced without impacting the logic. It is expected to count up to a target value provided as input. Once the target is reached, the counter must reset. There are also other cases in which the counter is expected to reset (e.g. an enable signal going low). Since the property will be proven for all combinations of all input values, these other cases must be stated in the property as well ('otherCondition' in the example), otherwise the proof would fail and these scenarios would be shown as counter example. If there are unspecified corner cases, then such kind of properties will reveal them.

The following assertion proves that the counter counts up each clock cycle in all cases except the ones specified in the last three 'OR' branches. The counter is compared to its own value one clock cycle earlier. At any time this property needs to evaluate the counting over only two clock cycles. Because it must be true in any two clock cycles, the property proves the full counting exhaustively for the full range of 2^{32} different target values. It was proven within seconds.

```
aExpectedCounting: assert property(
  counter == $past(counter) + 1     ||      // Normal counting
  $past(counter) == 0 ||                    // Case 1: counting not started
  $past(counter) == $past(targetValue) ||   // Case 2: counter should reset
  otherCondition                            // Case 3: counter should reset
));
```

This assertion only proves that the counter is increased in the normal counting mode. It does not prove what happens in cases 1 – 3. Additional properties have to be written in the same manner as explained in section IV.B. For example, case 2 can be proven with the following property: `(counter == targetValue |=> counter == 0)` which states that any time the counter equals the target value in the next clock cycle it must be 0. This was proven within seconds as well.

On the other hand the following property remains inconclusive for several days if the input signals are allowed to change any time.

```
aNoUnexpectedCntRst: assert property(
  counter == 0
  |-> ($past(counter) == 0 ||
       $past(counter) == $past(targetValue)
       || otherCondition)
);
```

By overconstraining the 'targetValue' to be stable and the 'otherCondition' to be always false, the property is proven. An exhaustive proof can be obtained within minutes by adding the following assertion as invariant[9]:

```
assert property (!(gate_counter > gate_length_sig));
```

The following property combines both requirements. It verifies that at any time the counter either counts up or one of cases 1 – 3 is true and the counter value is 0. This property was proven within seconds.

```
aExpectedCountingNoUnexptedRst: assert property(
  counter == $past(counter) + 1 ||          // Normal counting
  ( ($past(counter) == 0 ||                 // Case 1: counting not started
  $past(counter) == $past(targetValue) ||   // Case 2: counter should reset
  otherCondition)                           // Case 3: counter should reset
  && counter == 0 )
);
```

A cover property stating that the counter reaches a specific value of several thousands in decimal was only covered after several hours. Another cover property that stated that the counter reaches the same value and also equals the configurable target value took several days. All properties of this section have a sequential depth of 1 and consist of around 10 gates and 0 flops. There is no structural difference between **`aNoUnexpectedCntRst`**, the cover properties and the other assertions. While it is known that properties with a large sequential depth suffer from state-space explosion, it cannot be concluded that properties with a short sequential depth can be always proven within a reasonable time.

2) **Counting with local variables**

The interval counter has been described in section IV.B. The two clock cycles approach could not be used for this counter because there are many corner case scenarios that can influence the counter value. One solution was the usage of local variables within properties as demonstrated in the following simplified example. The signals cond1 – cond7 are set in the test bench based on combinations of input signals that reflect all the corner cases.

```
property pCounting();
  reg[`Bit_Width - 1:0] lCnt; // local counter
  (((cond1, lCnt = 1) or (cond2 or cond3, lCnt = 0))[*0:$]
    ##1
    ((cond4, lCnt++) or (cond5, lCnt = lCnt))[*0:$]
    ##1
    ((cond6, lCnt = lCnt) or (cond7, lCnt = 1)))
  |=> (lCnt == duvCnt);
endproperty
```

A local variable was used for counting the clock cycles between the start and end pulses. As the two pulses can have a distance of 2^{32} clock cycles, the properties could span this amount of sequential clock cycles and they became very complex and large (e.g. 200 gates, 30 flops). Usually it is recommended to write shorter properties. Nevertheless, these properties were proven in less than one hour for the full bitwidth of 40. Several bugs and unspecified points related to the corner case scenarios were found.

A drawback of this approach is that they were very difficult to debug and it was easy to make mistakes in the verification code. Generally, when an antecedent does never match, the tool reports that the property is "vacuous". The problem with "OR" statements in the antecedent is the matching of one branch of the " OR " can mask the fact that another branch can never occur. Cover properties and code coverage as described in section IV.E were used to ensure that the properties with the local variables covered all specified scenarios.

3) **Reference models**

Large parts of the current digitizer were verified with reference models. The functionality was modelled in synthesizable SystemVerilog inside the formal testbench based on the requirements. A simple property stated that the outputs of both the DUV and the reference model must always be equal. If this property is proven, the design functionality is proven exhaustively for any sequence of inputs, infinite sequential depth and for all possible combinations of input values. The block-level model code and properties were reused on (sub-)system-level. Input assumptions on block level were turned into assertions on (sub-)system-level. Additional modelling code was used for modelling the (sub-)system functionalities, which e.g. added delays or inverted or combined signals.

The pulse generator, input pulse counter and monostable pulse generator which can output a pulse of configurable length, were verified this way. The sub-systems Channel 1 and 2 were both proven with this approach. The interval counter and monostable pulse generator were modelled but the method did not scale for the full bitwidth. The counter width had to be reduced to 16 bits and 4 bits respectively in order to be able to get a proof within some hours. While the local variable approach was faster in terms of runtime, the reference model approach was faster in terms of development time and the code was more readable and therefore easier to debug. It will also be easier to adapt it for a future version of the design. The method was also successful in proving the expected value of an empty signal of the FIFO mentioned in section V. It depended on an internal counter that was counting up or down respectively when data was written to or read out from the FIFO.

E. Coverage

Unless requirements are written in a formal specification language and the reference model is derived from them, it cannot not be considered as golden model. Requirements can be misinterpreted and human mistakes can be made. There is a risk that the same bug is introduced in both models, which would not be revealed when comparing the outputs of the two models. The risk can be lowered if the verification engineers have no knowledge about the design, which is often required by safety standards, but it cannot be completely removed [8]. Cover properties that reflect the requirements were used to further reduce this risk. These are properties for which the formal tool produces a signal assignment and a wave form if they are coverable. Otherwise the tool reports that they are uncoverable. They can be used to sanity check the verification code. Wrong assumptions or faulty auxiliary code can overconstrain the design in a way that the tool cannot chose all legal input signal value combinations [4]. In some cases, properties were covered even though they were expected to be uncoverable. This revealed mistakes

in the auxiliary code of the formal module and in the assumptions. They were also used to improve the meaningfulness of requirements-based functional coverage.

In our methodology we assign at least one assertion to each requirement [8]. With the reference model approach the same assertion verifies most requirements. Therefore, we also assigned cover properties to the requirements and only considered a requirement as functionally covered if the cover property was covered as well. Structural coverage was calculated by the formal verification tool. For many blocks it could only be calculated for smaller bitwidths. Only if VHDL generics or (System)Verilog parameters were used consistently inside the DUV, it can be concluded that 100% structural coverage would also be achievable with a larger bitwidth. In that case it can be concluded that the design does not contain any further untested functionality, which is very important for safety [12] or security [13], and that no part of the functionality depends on hard coded larger values, as this would have been detected by uncovered branches or untoggled bits. It cannot be concluded whether a small value is by mistake hard coded or an internal signal uses a smaller bitwidth than it should. A code review should be performed additionally if 100% functional coverage could not be reached with the full bitwidth (e.g. due to inconclusive properties).

V. INTERACTION WITH THE UART

The blocks interacting with the UART could not be fully formally verified. On the TX side, the SRAM is accessed in a circular FIFO manner with the addresses calculated internally. By accessing the internal address signals, it could be proven that data which is written to an address is read out again if the same address is set as read address later. But it would have been more interesting to prove the wrapper logic that generates the addresses and write and read enable signals. The bottleneck was the handshake with a customized UART module which triggers the reading. 15 assumptions were needed to model it. They were proven on top level. The UART is operating with a four times slower clock and the wrapper logic depends on a fixed number of clock cycles of this clock and on an unchangeable number of input pulses. The handshake needs 92 clock cycles of the fast clock to read out one byte at the fastest possible baud rate. This means that reading out one set of eight 32-bit data words needs 2944 clock cycles. The formal tool needed several hours to generate counter examples, which made debugging very difficult.

On the top-level we created some test cases constructed with assumptions and assertions in order to evaluate whether any results can be obtained. Some counters were abstracted by inserting cut points and generating dependent signal values or pulses with assumptions. The internal UART TX block was copied into a wrapper and connected to the top-level RX side. A proof that the monostable parameters can be sent through the UARTs and reach the block took several hours. On the RX side we were not able to get a result within a day. Data tracking similar to [10] was successful in some specifically constrained scenarios. For an exhaustive proof it would have been necessary to add invariants as described in [9, 10]. The correct generation of the FIFO empty signal could be exhaustively proven with a reference model. Several bugs related to the empty signal and the handshake were found.

VI. RESULTS

Table 2 Faults Per Method

		Fault lead to update of			
Method	**Total faults per method**	**Specification**	**DUV**	**Verification requirements**	**Verification code**
Review	**11**	4	5	7	2
Short sequences	**1**	-	-	1	1
Chain of events	**3**	2	3	1	1
Local variables	**6**	1	2	2	2
Reference model	**9**	3	5	6	6
AutoCheck	**2**	-	2	-	-
TOTAL	**30**	**10**	**15**	**17**	**12**

We spent roughly 3 person months on formal verification. About 30 hours were spent on the verification planning and review meetings. The formal verification tool Questa PropCheck by Siemens EDA was used. Around 40 assumptions were needed, 50 cover properties were written and 70 assertions were proven. Functional coverage was calculated similarly to [8], but cover properties were also taken into account when reference models were used. Structural coverage was computed by the formal tool. 100% functional and structural coverage was achieved for all the blocks inside Channel 1 and 2 and on subsystem level as well. In the case of the interval counter and monostable pulse generator the bitwidths of the counters were reduced by half. Remaining blocks were simulated and emulated. Table 2 shows how many faults were found with each method and where they triggered modifications. Several bugs were also found during verification planning, review of the specification and with the Questa AutoCheck static analyzer. The fabricated ASIC has been characterized and shown to meet its specification.

VII. CONCLUSION

According to common perception the DUV of this work was mostly unsuitable for formal verification due to its many counters and serial interface. Nevertheless, many corner case faults could be found and many functionalities of the design could be proven. Examples of requirements and properties that could or could not be proven were presented. Counters that depended on less combinations of input signal values were easier to prove. More generalized properties seemed to be easier to prove than properties where a counter needed to reach a specific state (e.g. its maximum value). The structural size did not necessarily influence the proof result. Properties with the same small structure and sequential depth could be proven within seconds in some cases while others remained inconclusive for days. Some properties with much larger structure and sequential depth were proven within a reasonable time. The simplest approach of writing a reference model and comparing the outputs with a formal property was the most efficient in terms of development time. It revealed several bugs and most properties were proven within a short time. In some cases, it did not scale for large bitwidths of the counters. The verification process, which included a detailed specification and regular reviews, played an important role in finding faults.

REFERENCES

[1] H. Foster, Wilton Research Group and Mentor, A Siemens Business, “2020 Functional Verification Study”

[2] H. Boukabache, M. Pangallo, G. Ducos, N. Cardines, A. Bellotta, C. Toner, D. Perrin, D. Forkel-Wirth, “Towards a novel modular architecture for CERN radiation monitoring,” Radiat. Prot. Dosim. 173 (2017), pp.240-244, 2017

[3] S. Kundumattathil Mohanan, An Accurate Ultra-low Current Measurement ASIC for Ionization Chamber Readout, PhD thesis (to be published)

[4] E. Seligman, T. Schubert, M V A.K. Kumar, “Formal Verification: An Essential Toolkit for Modern VLSI Design“, Morgan Kaufmann, 2015, ISBN-13: 978-0128007273

[5] C. Krieg, M. Rathmair, F. Schupfer, “A Process for the Detection of Design-Level Hardware Trojans Using Verification Methods”, 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014, pp. 729-734, doi: 10.1109/HPCC.2014.112.

[6] A. Li, H. Chen, J.K. Yu, E.L. Teoh, I.P. Anand, “A Coverage-Driven Formal Methodology for Verification Sign-off”, Design and Verification Conference and Exhibition (DVCon) United States, 2019

[7] A. Gaur, G. Jain, R. Singh, “Metrics Driven Sign-off for SoC Specific Logic (SSL) Using Formal Techniques”, Design and Verification Conference and Exhibition (DVCon) United States, 2021

[8] Ceesay-Seitz K., Boukabache H., Perrin D. (2020) A Functional Verification Methodology for Highly Parametrizable, Continuously Operating Safety-Critical FPGA Designs: Applied to the CERN RadiatiOn Monitoring Electronics (CROME). In: Casimiro A., Ortmeier F., Bitsch F., Ferreira P. (eds) Computer Safety, Reliability, and Security. SAFECOMP 2020. Lecture Notes in Computer Science, vol 12234. Springer, Cham. https://doi.org/10.1007/978-3-030-54549-9_5

[9] A. Darbari, I. Singleton, “Industrial Strength Formal Using Abstractions”, In: CoRR (2016), https://arxiv.org/abs/1606.02347

[10] A. Darbari, “Smart Formal for Scalable Verification”, Design and Verification Conference and Exhibition (DVCon) United States, 2019

[11] S. Sutherland, I’m Still In Love With My X!, Design and Verification Conference and Exhibition (DVCon) United States, 2013

[12] European Committee for Electrotechnical Standardization, “Functional safety of electrical/electronic/programmable electronic safety-related systems,” May 2010

[13] X. Yhang, M. Tehranipoor, “Case Study: Detecting Hardware Trojans in Third-Party Digital IP Cores”, 2011 IEEE International Symposium on Hardware-Oriented Security and Trust, 2011, pp. 67-70, doi: 10.1109/HST.2011.5954998.