A Case Study on Formal Equivalence Verification Between a C/C++ Model and Its RTL Design

A Case Study on Formal Equivalence Verification Between a C/C++ Model and Its RTL Design

Gaetano Raia[1], Gianluca Rigano[2], David Vincenzoni[2(✉)], and Maurizio Martina[1]

[1] Politecnico di Torino, Torino (TO), Italy
gaetanomaria.raia@studenti.polito.it, maurizio.martina@polito.it
[2] STMicroelectronics, Agrate Brianza (MB), Italy
{gianluca.rigano,david.vincenzoni}@st.com

Abstract. In the field of communication system products, most datapath Digital Signal Processing algorithms are initially developed at a high-level in MATLAB® or C/C++. Subsequently, design engineers use these models as a reference for implementing Register Transfer Level designs. The conventional approach to verify their equivalence involves extensive Universal Verification Methodology dynamic simulations, which can last for months and require significant verification efforts. However, some elusive errors might still occur because it is infeasible to explore all input combinations with this method. On the other hand, Formal Equivalence Verification aims to verify that a Register Transfer Level design is functionally equivalent to the reference high-level C/C++ model across all possible legal states. With recent advancements in formal solver technology, Formal Equivalence Verification provides a distinct benefit by using mathematical methods to ensure that the Register Transfer Level (timed) matches the original high-level C/C++ model (untimed). This drastically reduces the verification time and ensures the exhaustive coverage of the design state space. This paper presents an in-depth exploration of complex Finite State Machine with datapath verification, specifically focusing on Multiplier-Accumulator, Tone Generator, and Automatic Gain Control, by employing the formal equivalence methodology. Although these signal processing blocks were previously verified throughout Universal Verification Methodology dynamic simulations, Formal Equivalence Verification was able to identify hard-to-find bugs in just a few weeks by utilizing the new workflow, thereby streamlining the verification process.

Keywords: Formal Equivalence Verification · Jasper™ C2RTL App · C/C++ model · RTL design · Formal datapath verification

1 Introduction

Integrated circuits have become a cornerstone in both commercial and industrial domains. As the demand for more sophisticated electronic devices has surged,

A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 373–389, 2025.
https://doi.org/10.1007/978-3-031-71177-0_23

the intricacy of these devices has considerably increased. This has necessitated continuous evolution in design methodologies and verification processes to meet the advancing technological requirements, as the cost of finding and solving bugs has grown exponentially throughout the design process [14,16].

Verification is a critical process designed to confirm that a Device Under Verification (DUV) maintains its intended behavior throughout its implementation. In the domain of System-on-a-Chip, a variety of verification technologies has been established. These technologies are essential for ensuring the functionality of these devices, which are complex designs integrating multiple disciplines. While various categories of design flaws contribute to integrated circuits re-spins, functional flaws remain the leading cause of bugs [18].

Additionally, the median percentage of total integrated circuit project time dedicated to functional verification is approximately between 50% and 60% [17]. This figure can vary depending on the design; projects that utilize existing pre-verified Intellectual Property (IP) may require less verification time, whereas those with newly developed IP could require more. In general, test planning and testbench development are the areas in which verification engineers spend the most of their time, respectively 47% and 21% [17].

Over the last decade, functional verification has been mainly conducted through the use of Universal Verification Methodology (UVM) testbench environments. A testbench serves as a verification framework that administers a predefined set of input patterns, also referred to as stimuli, to the Register Transfer Level (RTL) design. The primary function of a testbench is to facilitate the observation of whether the DUV yields the correct outputs, compared to a reference model, in reaction to these stimuli, as shown in Fig. 1.

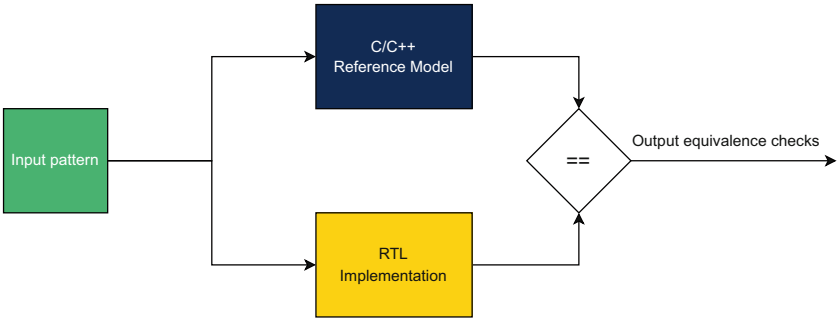

Fig. 1. Conceptual representation of functional equivalence verification between a C/C++ reference model and an RTL implementation

However, specifying millions of test vectors for exhaustive verification becomes impractical in simulation-based approaches due to the exponential increase in scenarios with the number of input bits: for instance, a 32×32 bit

integer multiplier would have 2^{64} total input combinations, at one million combinations checked per second, resulting into hundreds of thousands of processor years. Random simulations are a practical alternative, providing a statistical overview of compliance with specifications rather than exhaustive verification. Yet, purely random inputs can miss corner cases or produce unrealistic scenarios. Constrained-random simulations address this by guiding random input generation within defined parameters, improving coverage but still not guaranteeing full design space exploration.

Moreover, this approach is resource-intensive due to the number of components involved and is likely prone to subtle errors during the construction of the verification environment. Ultimately, the UVM requires considerable effort to verify its correctness prior to initiating the verification process. This results in prolonged verification cycles, which may prove sub-optimal for projects facing strict time-to-market constraints or operating with limited resources.

As electronic designs have become more complex and the time allocated for design cycles has decreased, the industry has developed a suite of verification methodologies and Electronic Design Automation (EDA) tools to address these challenges. This paper introduces an innovative verification flow that leverages Formal Equivalence Verification (FEV) to check that RTL designs match C/C++ models. This approach, based on mathematical properties, ensures exhaustive coverage and a significant reduction in verification time when compared to traditional UVM dynamic simulations. The comprehensive version of this paper, which includes in-depth discussions of the validated designs and the methodologies employed, is available for reference [15].

2 Formal Equivalence Checking in C-vs-RTL Scenarios

Functional verification is an essential step in the design process, aimed at confirming that the implementation reflects the design intent. The reconvergence model [2] suggests that the purpose of verification consists in ensuring that the result of some transformation, such as RTL coding, is as expected. This can be accomplished through a secondary path reconverging with the primary design path at a shared origin, namely the specification model (see Fig. 2).

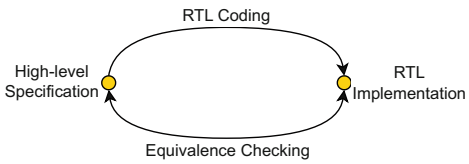

Fig. 2. Recovergence model of functional verification through equivalence checking

Formal equivalence checking employs mathematical reasoning to confirm that an *implementation* adheres to a *specification*. Formal verification leverages a language with precisely defined syntax and semantics to encapsulate the system's intended behavior, utilizing the IEEE standard for SystemVerilog Assertion (SVA) [8]. Through mathematical proofs, formal verification ensures the correctness of the Device Under Verification regardless of the input values, as it implicitly consider any legal case in the design state space. Two models are considered equivalent if, upon exhaustive analysis of all possible cases, the formal verification tool has not identified any discrepancies - commonly referred to as counterexamples - that would negate the equivalence.

To elucidate the mechanisms employed in today's equivalence-checking tools, it is instructive to consider a common computational equivalence model known as *miter*. This model effectively acts as a product machine that combines two Finite State Machine (FSM) designs by aligning each corresponding pair of primary inputs and connecting each pair of outputs to an XOR gate, as shown in Fig. 3. Establishing equivalence between two machines, denoted as $M_{\text{spec}}(X)$ for the specification machine and $M_{\text{impl}}(X)$ for the implementation machine, necessitates the demonstration that for any given input sequence $X = (x_1, x_2, \ldots, x_n)$, the outputs of the product machine consistently yield a zero value. Equivalence is thus confirmed by proving the nonsatisfiability of Eq. 1 across all possible inputs X, where $\oplus$ denotes the XOR gate operation.

$$M_{spec}(X) \oplus M_{imp}(X) \tag{1}$$

While there are various methods to address this challenge, such as Binary Decision Diagrams (BDDs) and Satisfiability (SAT) algorithms [14,16], they are beyond the scope of this paper.

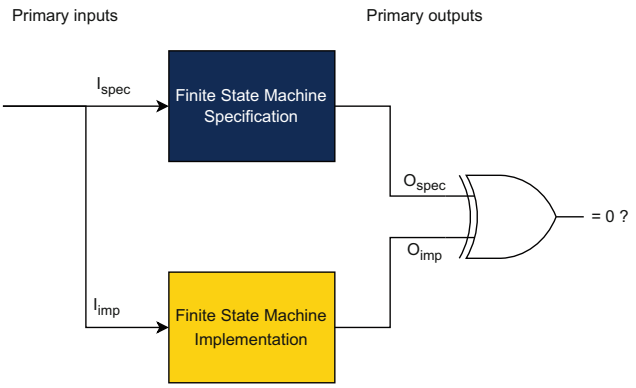

Fig. 3. Miter model of two FSM designs to verify through formal equivalence

2.1 Jasper™ C2RTL App

The advent of a novel category of formal engines embedded in Cadence® Jasper™ C2RTL App, specifically optimized for evaluating RTL datapath implementations against their C/C++ algorithmic specifications, has marked a significant leap in verification performance. These specialized engines are now capable of delivering performance that is up to 100 times more efficient than that of traditional general-purpose formal engines [10]. This innovation represents a substantial breakthrough for semiconductor companies, which frequently depend on robust, standardized EDA tools to manage the complexities inherent to design processes.

The integration of early-design formal verification checks into the design cycle can dramatically enhance the efficiency and effectiveness of the verification efforts. Nevertheless, it is essential to demonstrate that C/C++ models accurately capture the design intent, as these models often serve as the starting point for computational block development due to their abstraction capabilities, simulation speed, verification efficiency, and standard usage in the semiconductor industry. High-level C/C++ models can be easily verified at system level compared to RTL, to understand if they fulfill with their specifications: if so, they become the *golden reference* for the related RTL implementation. Implementing a redundancy layer enhances verification reliability by pinpointing whether inconsistencies stem from RTL coding mistakes or inaccuracies in translating design intent into C/C++, thereby preserving design integrity.

For the sake of clarity and focus, this paper does not delve into the specifics of formal engines, as their intricate details fall outside the scope of the current discussion (refer to [3,4] for any insight). The emphasis here is on the broader implications of these advanced tools and their impact on the semiconductor industry's verification practices, rather than on the technical nuances of the engines themselves.

3 The Verification Flow

In formal verification, intended behaviors are encapsulated as *properties*, which represent collections of logical and temporal relationships among subordinate Boolean and sequential expressions, usually written in SVA language. Over the past decades, verification engineers have been compelled to develop extensive sets of properties to capture all conceivable behaviors for verification. This approach has been both time-consuming and prone to risk, as the potential for overlooking certain properties could lead to incomplete verification and undetected design errors. The pivotal advantage of FEV in C-vs-RTL scenarios lies in the automatic comparison of the two models facilitated by the automatic generation of *assertions*. Concisely, an assertion is a declarative statement that specifies a property which must always hold. This automation streamlines the verification process, significantly reducing the manual effort and the associated risk of human error in property specification. Within this context, the formal

tool possesses the capability to generate mathematical properties checking that both the designs produce identical outputs under the same input conditions.

Although the verification methodology enhances autonomy in property generation and checking, it is not fully independent and continues to necessitate human guidance for configuring the verification environment, delineating the state space of the design, and addressing convergence issues that arise from state-space explosion in intricate digital circuits.

Despite these challenges, Jasper^{TM} C2RTL App is able to handle a large variety of datapath algorithms, such as unit arithmetic operations, high-level image processing algorithms, and encryption/decryption models [13]: in addition, it handles pipelines, feedback loops, floating-point and more [10]. The following list describes the innovative verification flow to apply FEV in verifying digital circuits, without the need to develop verification components and test vectors:

1. **C/C++ Model Compilation**: the tool adheres to the latest ANSI C++ standards and integrates with prevalent math libraries [10].
2. **RTL Compilation**: the tool supports SystemVerilog RTL implementations [10]. For non-SystemVerilog Hardware Description Language (HDL), equivalence checking tools ensure consistency with the original design [5,11].
3. **I/O Port Mapping**: verification engineers map input and output ports between the specification and implementation.
4. **Clock and Reset Definition**: the clock signal is identified in the RTL, and reset signal polarity is specified to detect the reset state.
5. **Input Assumptions**: engineers define input signal dynamics and protocols, acting as constraints to exclude illegal behaviors and prevent spurious counterexamples.
6. **Formal Engine Configuration**: while engineers can select optimizations for datapath-specific issues, leveraging the tool's machine learning-based configuration may yield optimal results [13], especially at the beginning.
7. **Coverage Property Specification**: engineers outline coverage properties to evaluate the DUV in targeted scenarios. These are employed by the user to prove the existence of at least one legal case fulfilling a specific condition, thereby facilitating the identification of the most concise path satisfying it.
8. **Proof Execution**: the tool checks for discrepancies between models using automatically generated and manually written assertions.

For the sake of clarity, Table 1 outlines all possible outcomes when proving an assertion. If the verification runs extensively without finding bugs, the verification user may decide to conclude the process, especially under tight design cycle deadlines. Generally, if a proof runs for more than 24 h without a result, it may be necessary to rewrite or decompose the proof or to try different engine modes [4]. This paper endeavors to delineate the challenges associated with managing sophisticated real-world digital circuits developed by STMicroelectronics and to outline effective strategies for ensuring convergence. To this end, it is advisable to construct a comprehensive verification strategy that establishes objectives, stages the complexity, and identifies coverage points.

Table 1. Possible proof status of formal verification

Proof status	Description
Unprocessed	The property is excluded from the proof target, even if declared
Undetermined	Neither full pass nor counterexample found over the run time
Counterexample	Property violated in at least one legal case
Proven	Property exhaustively proven in all the legal cases
Cover	The intended behavior can occur at least once
Unreachable	The intended behavior is never possible

4 Reconstruction of FSM-Like Datapath Behavior

In Digital Signal Processing (DSP) applications, numerous digital circuits exhibit behavior similar to FSM, where the next state is determined by the current state. This characteristic is straightforward to replicate in RTL designs due to the presence of memory elements such as registers that can hold state information. However, in C/C++ models, which are inherently untimed, managing state transitions to drive the next state can be challenging. In this section, the FEV process on checking the functional equivalence of a Multiplier-Accumulator (MAC) is described in detail.

The MAC unit finds extensive application across various DSP fields, including but not limited to audio and speech processing, image and video compression, telecommunications, radar and sonar systems, as well as biomedical signal processing. Specifically, it enables rapid computation for tasks such as filtering, convolution, and Fourier transforms in embedded systems. The main purpose of MAC is to repetitively add the product between two input signals to previously obtained intermediate results of the same nature. More precisely, such a functionality can be modeled under a mathematical point of view, according to Eq. 2. Given $a(k)$, $b(k)$ as input signals sampled at k-th cycle, the multiplication operation produces an intermediate result which is added to the sum of those computed during the $k-1$ previous cycles.

$$result = \sum_{k=0}^{N} a(k) \cdot b(k) \tag{2}$$

4.1 Specifications

The Device Under Verification is a Floating-Point MAC compliant to the IEEE-754 Standard for Floating-Point Arithmetic [7], patented by STMicroelectronics [19], whose interface and computational block diagrams are shown in Fig. 4. The block accepts three floating-point operands (namely *fp_a*, *fp_b*, *fp_c*) and a starting condition triggering a new operation to execute, that is indicated by *fp_opcode_i* (refer to List. 1.1 and List. 1.2 in [15] for C and RTL pseudo-codes).

Operational stability requires that, once the execution phase begins, the input signal *fp_op_start* remains inactive, while the input operands and the opcode signal retain their values until the completion of the computational phase. The block supports both straight and recursive arithmetical operations, involving multiplication, addition, and subtraction. On the output side, two distinct floating-point results (*fp_m* and *fp_z*) are provided, respectively containing the sampled outcomes produced by the multiplier and the adder/subtractor blocks.

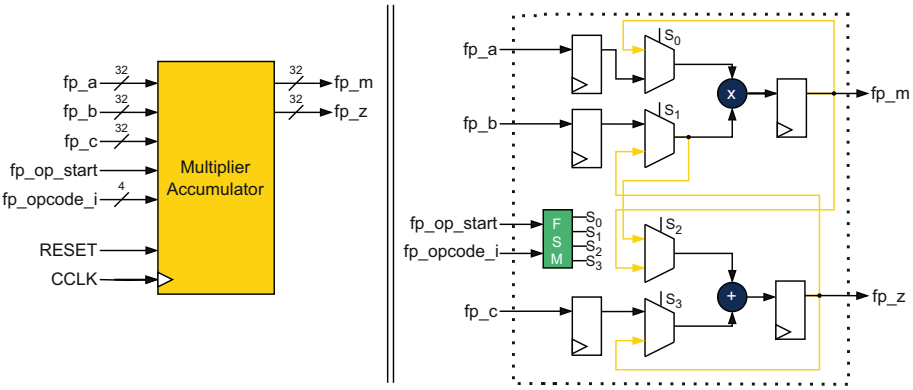

Fig. 4. Interface and computational block diagrams of the MAC

4.2 Verification Strategy

The verification aimed to affirm the RTL implementation's equivalence with its C model. The circuit comprised a pipelined floating-point multiplier and a combinatorial floating-point adder from a third-party IP, complicating direct formal verification. Focus thus shifted to verifying the control logic and feedback mechanism, using fixed-point behavioral models to represent the floating-point units analytically. Recursive operations in the RTL allow reusing previous outputs as operands for current computations, challenging to replicate in untimed C/C++ models due to their instantaneous computation on the same formal analysis cycle. While appropriate latency was easily introduced in model comparisons to properly handle the gap in pipelined designs, the formal tool lacked features to enable the C/C++ model to retain past values.

The most promising strategy consisted in extending the interface of the C/C++ model (see List. 1.1 in [15]), in a way that output values could be brought back as operands for recursive operations. Within this framework, SVA assumptions were essential to ensure the feedback continuity, forcing that *fp_m_in* corresponds to the prior cycle's *fp_m_out*, and similarly for *fp_z_in* and *fp_z_out*, as delineated in List. 1.3 in [15].

Further strategies to achieve complete convergence in useful time consisted in scaling down the bit-width of the arithmetical operands, from 32-bit to 8-bit

and 16-bit. In fact, design scaling simplifies the state space and thus accelerates convergence in formal verification by diminishing the computational complexity and the number of potential states to explore. Moreover, a case-splitting strategy, guided by opcode values, was implemented to isolate and address the most challenging cases for the verification tool, uncovering bottlenecks tied to specific algorithmic attributes. Consequently, manual assertions were crafted to refine the scope of auto-generated properties, as reported in List. 1.4 in [15].

To alleviate the verification load and ensure comprehensive equivalence, assertions were reformulated. This entailed incorporating the triggering condition and confirming the initial congruence of output signals from the preceding cycle. Stability of the summation outputs was verified over the calculation span of six cycles, whereas the multiplication output from the RTL was validated to not changing within the five-cycle interval post-triggering, concurrently maintaining equivalence with the C model in the subsequent cycle (see List 1.5 in [15]).

4.3 Results

Table 2 encapsulates the verification methodology for the MAC block. Despite some instances of *undetermined* proof results arose, we ultimately established complete equivalence across all cases by advancing through the verification sequence, confirming the functional correctness of the control part of the MAC block. Table 3 reports the run-times obtained at the last verification stage by bit-width value, illustrating the substantial influence of design scaling on formal tool performance. Additionally, Table 4 provides run-times from an earlier verification phase, demonstrating how case-splitting helps identify the most challenging scenarios for proof, namely recursive operations.

Table 2. Staging complexity in the verification of the MAC

Stage	Description
1	Verification of the control logic with fixed-point behavioral operators
2	Reconstruction of the feedback mechanism using SVA assumptions
3	Scaling down the bit-width to reduce the complexity of the formal problem
4	Application of case-splitting technique and manually written properties
5	Reformulating the manually written assertions to lighten the verification load

Table 3. Final run-times of the MAC verification considering all the opcodes

Opcode	8-bit	16-bit	32-bit
$[0x00, 0x0C]$	17.27 s	58.66 s	99.32 s

Table 4. Run-times at stage 4 of the MAC verification sequence (* stands for *undetermined* proof result, while apex symbol indicates the value at the previous cycle)

Opcode	Equation	8-bit	16-bit	32-bit
$0x00$	$fp\_z = fp\_b + fp\_c$	0.32 s	0.50 s	0.59 s
$0x01$	$fp\_z = fp\_b - fp\_c$	0.35 s	0.64 s	1.42 s
$0x02$	$fp\_z = fp\_z' + fp\_c$	0.34 s	1.44 s	1.34 s
$0x03$	$fp\_z = fp\_z' - fp\_c$	0.33 s	1.39 s	1.88 s
$0x04$	$fp\_z = fp\_m' + fp\_c$	$\approx$ 12 h *	$\approx$ 12 h *	$\approx$ 12 h *
$0x05$	$fp\_z = fp\_m' - fp\_c$	2.85 s	4.69 s	17.81 s
$0x06$	$fp\_m = fp\_a \cdot fp\_b$	7.54 s	39.57 s	131.97 s
$0x07$	$fp\_m = fp\_m' \cdot fp\_b$	$\approx$ 12 h *	$\approx$ 12 h *	$\approx$ 12 h *
$0x08$	$fp\_m = fp\_z' \cdot fp\_a$	2.62 s	21.35 s	0.88 s
$0x09$	$fp\_z = fp\_a \cdot fp\_b + fp\_c$	10.62 s	91.52 s	175.57 s
$0x0A$	$fp\_z = fp\_a \cdot fp\_b - fp\_c$	10.04 s	76.93 s	186.11 s
$0x0B$	$fp\_z = fp\_z' + fp\_a \cdot fp\_b$	13.4 s	63.97 s	530.02 s
$0x0C$	$fp\_z = fp\_z' \cdot fp\_a + fp\_c$	11.56 s	65.89 s	211.05 s

5 Decomposition of a Complex Cone of Influence

Utilizing the Jasper^{TM} C2RTL App facilitates equivalence checking to ascertain the functional correctness of an RTL design without necessitating manual property specification. Nevertheless, the automatic generation of end-to-end properties aimed at confirming output signal consistency under equivalent inputs can engender an intricate Cone Of Influence (COI). The COI, pivotal in formal verification, circumscribes the relevant RTL logic impacting a given property, enabling the exclusion of non-influential logic (refer to Fig. 5).

A case study on an STMicroelectronics-designed pipelined, frequency-tunable, and programmable-gain tone generator was undertaken to investigate methods for decomposing the COI in the context of challenging automatically generated properties. The tone generator is crucial for calibrating audio DSP systems, developing signal processing algorithms, and testing telecommunications networks.

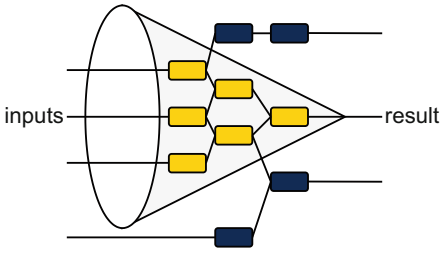

Fig. 5. Conceptual representation of the Cone Of Influence

5.1 Specifications

The tone generator accepts inputs such as the tone setup choice *mode_i* (either *single* or *double* tone), phase steps $\Delta\Phi$*_single* and $\Delta\Phi$*_double*, and programmable gains *gain_single* and *gain_double*, as shown in Fig. 6. It then generates outputs that consist of either one or two tones in the in-phase (I) and quadrature (Q) components, *Y_I* and *Y_Q* respectively. The phase step sets the incremental change in phase between successive samples, thereby setting the frequency of the tone(s), while gain controls the amplitude scaling applied to the output signal (see C and RTL pseudo-codes in List. 1.6 and List. 1.7 in [15]).

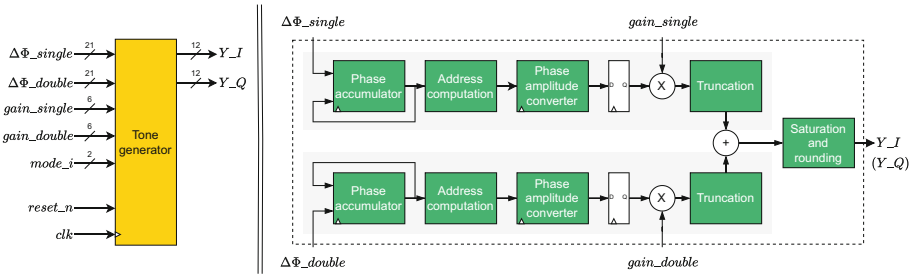

Fig. 6. Interface and computational block diagrams of the tone generator

The protocol governing input signals mandates static values within their defined legal ranges throughout execution:

- *mode_i* signal assumes values within the set {0, 1, 2}, where 0 denotes the idle state, 1 corresponds to single tone mode, and 2 to double tone mode.
- *gain_single* and *gain_double* signals are constrained to $[0, 63]$ and $[0, 31]$.
- $\Delta\Phi$*_single* and $\Delta\Phi$*_double* signals are restricted to the range $[0, (2^{21} - 1)]$.

5.2 Verification Strategy

The verification's primary objective was to establish the complete equivalence of the RTL implementation of the tone generator with its corresponding C/C++ model. Central to the block's functionality is a phase accumulator unit designed to iteratively compute the subsequent phase value, $(\Phi(t + 1))$, from the current phase, $(\Phi(t))$, and the input phase increment, $(\Delta\Phi)$, as described by the equation $(\Phi(t + 1) = \Phi(t) + \Delta\Phi)$. To manage the inherent feedback within the phase accumulator, a verification strategy analogous to that delineated in Sect. 4 was employed. This strategy proved ineffective, except for large phase step values, primarily due to the extensive bit-width and the vast array of potential cases which could not be exhaustively verified.

Given the design's intrinsic architecture, reducing parallelism was not feasible without altering the models, a course of action avoided due to the potential for

introducing errors. Consequently, an alternative strategy was adopted, which involved overconstraining the current phase value node in both the RTL and C/C++ models to accept any value within its legal range, independent of the phase step. The overconstraint ensured that the subsequent phase value would correspond to the overconstrained phase node's value from the previous cycle, thereby emulating the phase accumulator's functionality (see List. 1.8 in [15]). Overconstraining the internal phase node had not compromised the functionality of the circuit, since it affected both downstream and upstream logic.

Despite promising, overconstraining the internal phase node did not result as a completely satisfactory strategy because of the long time required to achieve the full proof. Because this was mainly due to the high complexity of the computational load, a more powerful technique, consisting in inserting extra assertions by leveraging intermediate equivalent points between the C/C++ and RTL models, was employed. While it may appear that adding more assertions could increase the workload for the verification tool, proven assertions at the intermediate key points can actually aid the formal tool in verifying more complex automatically generated end-to-end properties (see List. 1.9 [15]). In this case, intermediate equivalent points were placed at data processing stages (refer to Fig. 6), such as:

- Sample values coming out from the phase amplitude converter.
- Sample values scaled by the input gain and then truncated.

A more sophisticated and efficient verification strategy involved explicitly instructing the formal verification tool to utilize proven assertions at intermediate equivalent key points within the design. By doing so, these assertions act as simple blocks within a more complex chain of end-to-end properties. As the verification tool progresses through the smaller properties, it utilizes the *proven assertions* as *helper assumptions* for subsequent assertions in the verification chain. An end-to-end property is considered *proven* if all its helper assumptions are also *proven*. Consequently, the *assume-guarantee* method was employed as the terminal verification technique to expedite the attainment of a comprehensive proof. This approach mitigated the complexity of the global Cone Of Influence by partitioning challenging monolithic assertions into discrete, tractable formal verification sub-problems, each with a correspondingly narrowed COI.

5.3 Results

The application of FEV techniques successfully confirmed the functional equivalence between the C/C++ model and the RTL implementations. For the sake of clarity, proof convergence was achieved by following the verification sequence reported in Table 5. Table 6 reports the infeasibility of feedback reconstruction using SVA assumptions, highlighting the formal tool's difficulty with diminishing phase step values. Table 7 summarizes the run-times by verification stage and working mode, proving the advantage of determining the equivalence at intermediate points to aid the formal tool in achieving convergence.

Table 5. Staging complexity in the verification of the tone generator

Stage	Description
1	Reconstructing the feedback of the phase accumulator
2	Overconstraining the current phase node using SVA assumptions
3	Proving the equivalence at intermediate points inserting extra assertions
4	Applying the assume-guarantee to leverage equivalence at intermediate points

Table 6. Run-times at stage 1 of the tone generator verification sequence

Input phase step	Proof status	Run-time
2^{20}	Proven	17.6 s
2^{19}	Proven	62.8 s
2^{18}	Proven	29 min
2^{17}	Proven	$\approx$ 12 h
2^{16}	Undetermined	$\approx$ 24 h

Table 7. Run-times at different stages of the tone generator verification sequence

Single tone mode	Stage 2	Stage 3	Stage 4
Run-times	221 min	132 min	26 min
Proof status	Proven	Proven	Proven
Time reduction		40%	80%
Double tone mode	Stage 2	Stage 3	Stage 4
Run-times	48 h	111 min	33 min
Proof status	Undetermined	Proven	Proven
Time reduction		N/A	70%

6 Proving the Equivalence with a MATLAB®-derived C Code

Significant algorithmic differences between high-level C/C++ models and RTL designs present notable challenges in proving functional equivalence using FEV techniques. This is especially the case for C code derived from MATLAB® where complexity can increase due to several factors, such as the variations in data types and bit-width choices. Despite casting procedures are supported by the formal tool, it is highly recommended to minimize type discrepancies between RTL and C/C++ representations. This approach was employed in the verification of an Automatic Gain Control (AGC) design.

6.1 Specifications

The main purpose of an AGC circuit, within a receiver in a communication system, is to maintain a constant output amplitude level of a signal despite variations in the amplitude of input signal, as represented in Fig. 7. The device under analysis is an AGC (targeting IEEE 802.15.4g protocol [6]) designed by STMicroelectronics, governed by an FSM with datapath mechanism. Due to confidentiality constraints, detailed information about the specific block cannot be disclosed in this publication. The AGC accepts inputs from a Received Signal Strength Indicator (RSSI) block through *rssi_result* signal, providing an output gain value to a Programmable Gain Amplifier (PGA) using *gain* signal, as shown in Fig. 7. Configuration of the AGC is achieved by setting the *mode* signal, initializing the gain with *start_gain*, and adjusting the gain using *gain_step*.

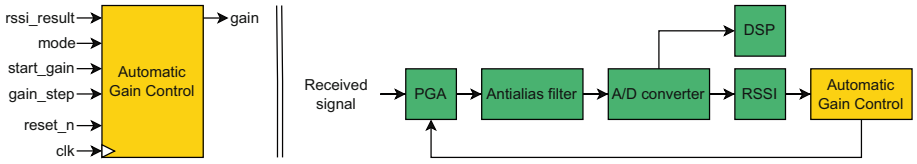

Fig. 7. System level representation of the Automatic Gain Control

6.2 Verification Strategy

Significant algorithmic differences between the two models precluded to leverage intermediate equivalent key points to aid the formal tool. Consequently, beyond feedback reconstruction technique described in Sect. 4 to emulate the FSM behavior, further modifications were implemented. To alleviate verification overhead, all *double* data types in the C/C++ model - automatically converted from MATLAB® codes using MATLAB Coder [12] - were converted to *int* data types to align with the RTL design specifications. This conversion necessitated the creation of new functions within the C/C++ code, which are listed in Tab. 10 in [15]. Additionally, to prevent state explosion issues commonly associated with counters, their maximum count values were deliberately constrained to zero to avoid multiple accumulation of samples. This cap was not overly restrictive, as the accumulation underwent separate verification from the gain computation under specific input scenarios, namely the main target of the verification process.

6.3 Results

The verification of this case was particularly challenging due to algorithmic divergences between the high-level and low-level models and the design's complex control logic. Table 8 outlines the verification sequence utilized for the AGC block, which ultimately resulted in the run-times presented in Table 9. Despite the full equivalence was not proven in all cases, the application of FEV techniques

resulted valuable by quickly identifying two mismatches between the C/C++ model and the RTL design, corresponding to subtle overflow cases that were not discovered during previous UVM dynamic simulations.

Table 8. Staging complexity in the verification of the AGC

Stage	Description
1	Reconstructing the feedback mechanism to emulate the FSM behavior
2	Converting the data types of the C/C++ model from *double* to *int*
3	Disabling counters to avoid multiple accumulation of samples

Table 9. Run-times at the final stage of the AGC verification sequence

Mode	Description	Proof result	Run-time
Mode_1	Fixed-gain configuration	Proven	0.34 s
Mode_2	Gain can only decrease	Proven	121.74 s
Mode_3	Gain can only decrease until a certain value	Proven	754.10 s
Mode_4	Gain can both increase and decrease	Undetermined	$\approx$ 48 h
Mode_5	Gain is determined by non-trivial RSSI conditions	Undetermined	$\approx$ 48 h

7 Conclusion

This paper presented a case study on applying various FEV techniques within the context of verifying the functional equivalence between three-real world high-level C/C++ models and RTL designs using the Jasper^{TM} C2RTL App. This innovative approach enables exhaustive exploration of the design state space, potentially revealing bugs that traditional verification methods might miss. Moreover, by utilizing high-performance formal engines, the verification time for typical DSP components has been significantly reduced - from months to just a few weeks per case study -compared to UVM dynamic simulations, specifically:

- UVM environment setup traditionally requires six weeks, whereas C2RTL preparation, including port mapping, adaptation of C/C++ models, and verification plan formulation, is completed within one week.
- Test development in UVM extends over five weeks, in contrast to the two weeks needed for incorporating appropriate constraints in C2RTL. This entails specifying legal input signal values and protocols, methodically exploring the design state space, and applying effective verification techniques to ensure convergence.

– Debugging in UVM, which involves analyzing dynamic simulation waveforms, typically spans two weeks. Conversely, C2RTL reduces this to a matter of days, benefiting from the provision of succinct counterexample waveforms and facilitated root cause analysis.

Customizing the formal tool to accommodate the specific characteristics of each DUV proved to be a non-trivial task. There is no replacement for the verification user's knowledge of the expected behavior and the selection of appropriate techniques to assist the tool in handling FSM-like behaviors, large COI and significant algorithmic difference between high-level C/C++ models and the RTL designs. In conclusion, this paper has demonstrated that the strategic application of FEV techniques, facilitated by the Jasper^{TM} C2RTL App, significantly enhances the efficiency and effectiveness of DSP component verification.

References

1. Albin, K.: Oracle labs: the cost of SoC bugs. In: Design and Verification Conference and Exhibition, U.S. (2016)
2. Bergeron, J.: Writing Testbenches using SystemVerilog . 1st edn. Springer, (2006)
3. Cadence Design System: Jasper C to RTL Equivalence Checking App User Guide (2023)
4. Cadence Design System: Jasper Engine Selection Guide (2023)
5. Formality Equivalence Checking. https://www.synopsys.com/glossary/what-is-equivalence-checking.html. Accessed 15 June 2024
6. IEEE: IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 3: Physical Layer (PHY) Specifications for Low-Data-Rate, Wireless, Smart Metering Utility Networks, pp. 1–252, IEEE Std 802.15.4g-2012 (2012)
7. IEEE-754: Standard for Floating-Point Arithmetic. IEEE Std 754-2008, pp. 1–58. IEEE (2008)
8. IEEE Computer Society and IEEE Standards Association Corporate Advisory Group: IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800^{TM}-2017). IEEE, New York (2017)
9. Jasper C Apps. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-c-formal-verification.html. Accessed 13 June 2024
10. Jasper C Apps. https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/jasperc2rtl. Accessed 14 June 2024
11. Jasper SEC App. https://www.cadence.com/zh_TW/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-sequential-equivalence-checking-app.html. Accessed 15 June 2024
12. MATLAB Coder. https://it.mathworks.com/products/matlab-coder.html. Accessed 19 Jun 2024
13. Mittal, V., Roy, S., Singhal A.: Embracing datapath verification with Jasper C2RTL App. In: Design and Verification Conference, India (2022)
14. Perry, D.L., Foster, H.: Applied Formal Verification: For Digital Circuit Design. 1st ed. McGraw Hill LLC (2005)

15. Raia, G., Vincenzoni, D., Rigano, G., Martina, M.: A Case Study on Formal Equivalence Verification between a C/C++ Model and its RTL Design: A Long Companion Version. Zenodo (2024). https://doi.org/10.5281/zenodo.12591803
16. Seligman, E., Schubert, T., Kirankumar, M.: Formal Verification: An Essential Toolkit for Modern VLSI Design, 1st edn. Morgan Kaufmann Publishers Inc, San Francisco (2015)
17. The 2022 Wilson Research Group Functional Verification Study (Part 8). https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/. Accessed 13 June 2024
18. The 2022 Wilson Research Group Functional Verification Study (Part 12). https://blogs.sw.siemens.com/verificationhorizons/2023/01/09/part-12-the-2020-wilson-research-group-functional-verification-study-2/. Accessed 13 June 2024
19. Vincenzoni, D., Raffaelli, S.: Circuit for performing a multiply-and-accumulate operation. (10089078, 3299952,10437558) (2016)