Validating Formal Hardware Specifications Through Generated Behavioral Models

Robert Kunzelmann*†, Zeyad Tahoun*‡, Vinod Bangalore Ganesh*§,
Maximilian Berger*†, Emil Baerens*†, Wolfgang Ecker*†
*Infineon Technologies AG, Neubiberg, Germany
†Technical University of Munich, Munich, Germany
‡Politecnico di Torino, Turin, Italy
§Technical University of Dresden, Dresden, Germany
{robertniklas.kunzelmann, emil.baerens, wolfgang.ecker}@infineon.com,
{tahoonzeyad, vinodbangaloreganesh}@gmail.com, maximilian_alexander.berger@tum.de

***Abstract*—Developing large-scale integrated circuits starts with specifying the desired system behavior. While specifications must be precise and dependable for both design and verification purposes, traditional specifications rely on natural language documents and their human interpretation. This introduces two critical weaknesses: First, interpretation can be challenging due to vagueness and inherent ambiguity. Second, validating that a natural language specification expresses the intended behavior is hardly possible with deterministic methods. To tackle these challenges, we use a formal specification format, called the Universal Specification Format (USF), with unambiguous syntax and semantics. USF applies to the specification of general digital hardware and automatically generates formal properties for design verification. Still, it must be ensured that the formal specification—and thus the generated properties—correctly express the desired system behavior. In this paper, we present a novel code generator for behavioral simulation models to execute USF specifications and validate them against use cases. Moreover, we introduce and integrate runtime checks into the simulations that automatically detect inconsistencies and gaps in the specification. This methodology has been applied to industrial-strength hardware components and their formal specifications, demonstrating the effectiveness and industry-readiness of our behavioral simulation models and the automated runtime checks. We finally show that USF enables reusable code generators for both simulation-based specification validation and formal design verification.**

***Index Terms*—Behavioral modeling, code generation, design verification, formal specification, specification validation.**

I. Introduction

The development of modern integrated systems necessitates precise and thorough specifications. Design and verification engineers can only work efficiently if they can rely on the requirements and constraints they must fulfill, which implies that those requirements must be unambiguously stated. Unfortunately, traditional text-based specification approaches do not match these needs [1–5]. Design specifications often express the system behavior in natural language. As a consequence, incorrect or incomplete specifications account for 40–60% of the root causes of functional flaws in design projects [1, 2].

This work was partly funded by the German Federal Ministry of Research, Technology and Space (BMFTR) within the project Scale4Edge under contract number 16ME0122K.

The problems with many informal specifications are twofold. The first challenge lies in writing complete and consistent specifications [3]. Particularly, validating that an informal natural language specification matches the intended behavior without the ability to execute and test the specification can only be achieved by laborious and error-prone manual review. A second issue arises from the human interpretation of written specification documents. Their increasing technical complexity and the inherent ambiguity of natural language make them prone to misinterpretation [4, 5]. Consequently, deriving hardware implementations and verification references from informal specifications with inherent uncertainty poses challenges to the reliability of the entire development process.

Addressing the first issue of interpreting specifications, we use a formal specification format (Sec. II), which eliminates the ambiguity of natural language and introduces a precise syntax and semantics. This work uses the Universal Specification Format (USF), which describes general digital hardware from an operational perspective as a set of discrete hardware functions [6]. Additionally, we perform time-annotation to express the system's cycle-accurate temporal behavior (Sec. II-C) [7]. This formalization makes the resulting specifications machine-readable, enabling algorithmic information retrieval and transformation. Accordingly, USF is used to generate formal properties through automated code generation [8]. As verification references, these generated properties mathematically prove that a design implementation matches the specification.

While formalization through USF makes interpreting a hardware specification unambiguous, creating the formal models themselves is still prone to human error. Therefore, this work introduces a simulation-based approach to validate the correctness, completeness, and consistency of a USF specification (Sec. III). We consider a formal specification to be the most abstract and holistic formalization of the system's behavior. Generally, there exists no other formal reference to statically check against. Thus, we employ use case simulations to validate the specification dynamically in specific scenarios where the intended system behavior is explicitly known. We present a code generator for timed behavioral models to dynamically execute and analyze the specification. The generated models

978-3-9826741-1-7/DATE26/© 2026 EDAA

are cycle-accurate to match the abstraction level of the USF-generated formal properties used in design verification. Moreover, we integrate general checks into the behavioral models that analyze the simulations at runtime, automatically detecting inconsistencies and gaps in the specification (Sec. III-B).

We demonstrate the effectiveness of behavioral simulations in specification validation by applying USF and the presented code generator to a set of industrial hardware components (Sec. IV). Particularly, we list case studies where our automated runtime checks effectively identified gaps and inconsistencies in specifications. Additionally, we use the validated specification of each component to generate formal properties and verify their respective hardware implementations, demonstrating the versatility of USF. Finally, we discuss the implications and applicability of our behavioral simulations (Sec. IV-C) and compare them against related works (Sec. V).

II. Specification Format and Model Semantics

This section introduces the USF specification format, which serves as the input for generating behavioral models. We exemplify the format and provide a mathematical formalization of its semantics, establishing the foundation for the later transformation into behavioral simulation models (cf. Sec. III).

A. USF Specification and Traces

USF models generalize digital hardware systems by their interface and abstract system state, as well as a set of discrete hardware functions accessing the state and interface [6]. The concept of expressing hardware by a discrete function set has already been introduced as a modeling technique, most notably in the context of Instruction Set Architectures (ISAs) of processors, but also in Instruction-Level Abstraction (ILA) [9], the Bluespec hardware description language [10], and temporal logic assertions for verification [11].

Similar to ISAs, USF expresses functions by their trigger conditions and executed data flow, i.e., the arithmetic and logic expressions performed on the system state by a function when selected. In addition, functions can be time-annotated into cycle-accurate models, adopting the notation of *traces* from [7]. A trace is a transition system closely related to abstract state machines [12]. The states of a trace always align with the steps of a discrete time grid, typically the system clock. Trace states are abstract constructs that potentially cover a sub-space of concrete system states. Transitions between states model conditional data propagation through the system and can span multiple cycles, unlike sequential logic. Such time-annotated USF specifications enable the straightforward generation of temporal logic properties for verification [8].

An example of a USF model is given in Fig. 1 as a partial specification of a First In First Out (FIFO) buffer. Besides the interfaces, state variables, and untimed function expressions, it also includes the partial traces of the *pop* and *push* functions. As highlighted in purple, the transitions and expressions have been time-annotated, indicating that updating the primary output during a *pop* takes zero cycles while updating the internal state during a *push* takes one cycle.

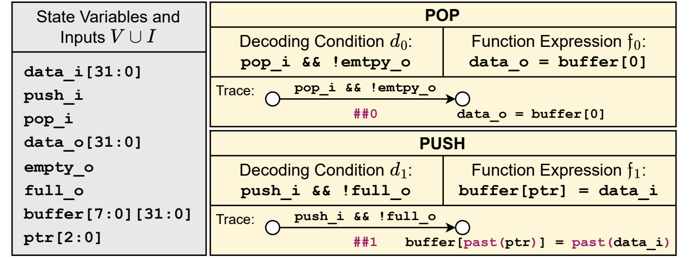

Fig. 1. Excerpt from an example USF model of a linear FIFO exemplifying the formal specification of state variables and inputs, partial *pop* and *push* functions, and their cycle-accurate expression via traces.

B. Formalization of Untimed Function Models

As previously mentioned, USF specifies hardware from a function-based perspective. In contrast to conventional Finite State Machine (FSM) models of sequential logic, which explicitly define the state space Q^V, USF defines the state variables V_{state} and V_{output} that hold the system state and primary output assignments. Rather than the transitions between single states, USF specifies functions $\mathcal{F}$ which access the system state and primary inputs I. To perform a function $\mathfrak{f} \in \mathcal{F}$, a set of decoding conditions D is specified, where each condition $d \in D$ triggers one or multiple functions.

Formally, we define a USF specification as the quintuple $\mathcal{U} := \langle V, q_0, I, \mathcal{F}, D \rangle$:

- V: Finite, non-empty set of state variables including the primary outputs with $V = V_{\text{state}} \cup V_{\text{output}}$; the finite set of reachable system states is denoted by Q^V
- $q_0 \in Q^V$: Initial system state
- I: Finite set of primary inputs with the input alphabet Σ^I
- $\mathcal{F}$: Finite set of hardware functions with $\mathfrak{f} \in \mathcal{F},\ \mathfrak{f} : (\Sigma^I \times Q^V) \mapsto Q^V$
- D: Finite set of decoding conditions, each selecting potentially multiple hardware functions with $d \in D,\ d : (\Sigma^I \times Q^V) \mapsto 2^{\mathcal{F}}$

C. Trace-Based Time-Annotation

Based on the definition of USF, there is a *trigger-reaction* relationship between a decoding d and its corresponding function $\mathfrak{f}$. While this relationship is untimed in USF, design tasks like assertion-based verification and behavioral modeling require the expression of temporal relations between events.

Bridging the gap between abstract function expressions and the requirement for cycle-accurate temporal logic, we use time-annotation to manually specify the temporal evaluation and execution of each decoding condition d and function $\mathfrak{f}$. Here, we apply the notation of traces, which are abstract transition systems [7, 12]. A trace comprises a set of temporal states S that model the ordering of actions Λ^l performed on the system state during the execution of $\mathfrak{f}$. Transitions model the conditions and time steps between actions. Thus, for each USF functionality $\langle d, \mathfrak{f} \rangle$, where d triggers $\mathfrak{f}$, a trace $\mathcal{T}$ models the cycle-accurate Refinement (Ref) of decoding and execution:

$$\mathcal{T} \models \text{Ref}(\langle d, \mathfrak{f} \rangle) \tag{1}$$

In (1), a trace $\mathcal{T}$ is a quintuple $\mathcal{T} := \langle S, s_0, S_e, \tau^l, \Lambda^l \rangle$:

- S: Finite, non-empty set of temporal states; different from the system states Q^V (cf. Sec. II-B)
- $s_0 \in S$: Initial trace state
- $S_e \subseteq S$: Finite, possibly empty set of end states
- τ^l: Transition function of l time steps; includes the initial condition to start the trace
  $\tau^l \models \mathrm{Ref}(d)$ and $\tau^l : (S \times Q^V \times \Sigma^I) \mapsto S$
- Λ^l: Set of trace actions of l time steps with
  $\Lambda^l \models \mathrm{Ref}(\mathfrak{f})$ and $\lambda^l \in \Lambda^l,\ \lambda^l : (S \times Q^V \times \Sigma^I) \mapsto Q^V$

It is important to note that the temporal states—or trace states—S are abstract concepts used for modeling. They do not have a unique encoding stored in an internal register. Instead, they represent potentially overlapping sub-spaces of the system state space Q^V where actions are performed. In USF, a hardware function $\mathfrak{f}$ expresses the untimed bit-accurate update of the system state. The trace actions Λ^l refine $\mathfrak{f}$ and express which part of the system state is updated at which time point. The length $l \in \mathbb{N}_0$ of an action $\lambda^l \in \Lambda^l$ denotes the time window of past values, which is accessed to compute the system state update. Transitions between trace states are modeled by τ^l and can take an arbitrary number of time steps to express multi-cycle and immediate operations.

III. Behavioral Model Generation

The following section introduces the generation of behavioral models from time-annotated USF and trace models. These behavioral models are used to evaluate and validate the formal specifications by use case simulation. Moreover, this section presents a general runtime checker that tracks the write accesses to each internal signal in each cycle and automatically detects gaps or inconsistencies in the specification.

A. Mapping Traces to Software Semantics

To execute USF and traces, we transform the corresponding static model into a discrete and fixed-increment time progression simulation. The simulation consists of two parts: First, the use of callable trace functions[1], each modeling the execution of a single trace. And second, a simulator kernel scheduling the parallel execution of these trace functions. As all traces in the specification express cycle-accurate transition systems, we first translate each trace into a callable trace function that models the trace's transitions, and a trace object that stores the current state of the trace. The kernel loops over each simulated cycle and updates each trace object by applying its respective trace function. Here, we dynamically activate and deactivate trace objects when traces start and finish, respectively.

As described by (1) in Sec. II-C, a USF model is manually time-annotated by a set of traces $\mathbf{T}$:

$$\mathbf{T} = \{\mathcal{T}_i \mid 0 \leq i < n\} \tag{2}$$

The first step to map a specification from the USF format to executable software semantics is to transform each trace $\mathcal{T}_i$ into a sequential logic-like FSM $\mathcal{M}_i$. $\mathcal{M}$ uses the same formalism as a trace $\mathcal{T}$ from Sec. II-C with the distinction that each transition consumes exactly one cycle. Thus, we automatically flatten multi-cycle transitions by inserting intermediate states and resolve zero delay transitions by merging their source and sink states. Further, the state space Q is augmented with a state variable to give each trace state $s \in S$ a unique encoding.

$$\mathbf{T} \mapsto \mathbf{M} = \{\mathcal{M}_i \mid 0 \leq i < n\} \tag{3}$$

For the behavioral simulation, each FSM $\mathcal{M}_i$ is represented by, first, a trace object $o_{i,j}$ storing the state of $\mathcal{M}_i$ at cycle $0 \leq j < m$ and, second, a callable trace function $f_i \models \tau_i$:

$$\begin{gathered} \mathbf{M} \mapsto O_{\text{static}} = \{o_i \mid 0 \leq i < n\} \\ \mathbf{M} \mapsto F = \{f_i \mid 0 \leq i < n\} \end{gathered} \tag{4}$$

such that:

$$\begin{gathered} o_{i,0} \models s_{i,0} \\ o_{i,j+1} = f_i(o_{i,j}) \\ o_{i,j} \models s_{i,e} \in S_{i,e} \implies \nexists f_i(o_{i,j}) \end{gathered} \tag{5}$$

In (5), $o_{i,0}$ is the initial value of a trace object modeling the initial state $s_{i,0}$ of its respective trace $\mathcal{T}_i$. The trace object is discretely updated to its next value by applying f_i. Only if the trace reached its end, i.e., $o_{i,j}$ holds a value associated with the end states $S_{i,e}$ of $\mathcal{T}_i$, there exists no next state and f_i does not yield a new result.

Due to the definition of traces from Sec. II-C, a trace $\mathcal{T}_i$ can be inactive, active, or even active multiple times simultaneously—up to $p_i \in \mathbb{N}^+$ to model parallelism—at any cycle j. Thus, we denote the set of currently active trace objects O_j:

$$O_j = \left\{ o_{i,j,k_i} \;\middle|\; \forall i : \left(\nexists o_{i,j,k_i} \bigvee \{o_{i,j,k_i} \mid 0 \leq k_i < p_i\} \right) \right\} \tag{6}$$

where k_i in (6) denotes the index of trace objects of type i.

For structuring, we can split O_j into:

- $O_{\text{new},j}$: Newly instantiated trace objects
- $O_{\text{active},j}$: Previously instantiated trace objects
- $O_{\text{done},j} \subseteq O_{\text{new},j} \cup O_{\text{active},j}$: Previously and newly instantiated objects that have reached their final state

such that:

$$\begin{gathered} O_j = O_{\text{new},j} \cup O_{\text{active},j} \\ O_{\text{active},0} = \emptyset \\ O_{\text{active},j+1} = O_j \setminus O_{\text{done},j} \\ o_{i,j} \models s_{i,e} \in S_{i,e} \iff o_{i,j} \in O_{\text{done},j} \end{gathered} \tag{7}$$

Equations (7) state that at the start of the simulation, when $j = 0$, there are no active traces, and the only activity comes from the input stimuli and newly instantiated traces. During simulation, the next set of active traces includes all currently active traces, excluding those that reached their respective end states, as their trace objects are deactivated in the next cycle.

Based on (5) and (7), the discrete and fixed-increment time progression kernel is constructed as a three-dimensional loop iterator over cycle j, trace type i, and trace instances k_i, as

[1]USF *hardware functions* specify data flow. In contrast, *trace functions* in behavioral simulations are callable software methods updating trace objects.

Algorithm 1. Cycle-based simulator kernel looping over all active trace objects to update their modeled states and activate new trace objects if possible.

1 **foreach** $j \in [0; m[$ **do** ▷ cycle-by-cycle progression
2 $O_{\text{next}} \leftarrow \emptyset$ ▷ update existing traces
3 **foreach** $i \in [0; n[$ **do**
4 **foreach** $o_{i,j,k_i} \in O_{\text{active}}$ **do**
5 $o_{i,j+1,k_i} \leftarrow f_i(o_{i,j,k_i})$ ▷ no effect if $\nexists o_{i,j+1,k_i}$
6 $O_{\text{next}} \leftarrow O_{\text{next}} \cup \{o_{i,j+1,k_i}\}$
7 **end**
8 **end**
9 $O_{\text{new}} \leftarrow \emptyset$ ▷ instantiate new traces
10 **repeat**
11 **foreach** $i \in [0; n[$ **do**
12 **if** activationConditionTrace(i) **then**
13 $k_i \leftarrow k_i + 1$
14 $o_{i,j,k_i} \leftarrow$ new()
15 $o_{i,j+1,k_i} \leftarrow f_i(o_{i,j,k_i})$ ▷ no effect if $\nexists o_{i,j+1,k_i}$
16 $O_{\text{new}} \leftarrow O_{\text{new}} \cup \{o_{i,j+1,k_i}\}$
17 **end**
18 **end**
▷ repeat until stabilized and no new traces are triggered
19 **until** unchanged(O_{new})
20 $O_{\text{active}} \leftarrow O_{\text{next}} \cup O_{\text{new}}$ ▷ active traces for next cycle
21 **end**

shown in Algorithm 1. First, the next cycle values for all active trace objects are determined by applying their respective trace function f_i (lines 2–8). Next, new trace objects are activated based on the current system state and input stimuli. Algorithm 1 abstracts the decision of whether a new trace object of type i is activated by evaluating *activationConditionTrace(i)*, which is the starting condition of a trace included in τ^l (cf. Sec. II-C). If a new trace object is activated, its trace function is called since the initial state $s_{i,0}$ can have an immediate effect on the global system state (lines 14–15). Consequently, we repeatedly loop over the trace activation until the activation condition evaluates to false for all trace types i (lines 10–19). At this point, there are no further updates to the system state, and the kernel progresses to the next cycle $j + 1$.

As the target simulation language, we use non-synthesizable SystemVerilog for its intrinsic support for hardware semantics and time units. Trace objects and functions are implemented in classes that are dynamically instantiated. The simulation kernel runs in a `forever` loop to perform immediate updates to the system state, rather than a sequential process.

B. Trace Output Merging and Conflict Resolution

While Algorithm 1 focuses on the scheduling and updating of traces, each active trace also computes an update to the shared system state by executing $\lambda^l \in \Lambda^l$ corresponding to its current trace state (cf. Sec. II-C). The parallel execution of discrete traces requires that all proposed updates to the system state are collected and checked for consistency before committing to the state update for the next cycle progression. This check of multiple update proposals from all active traces is performed automatically by a general software function integrated into our behavioral simulations. At runtime, it collects all values from all traces driving a shared signal.

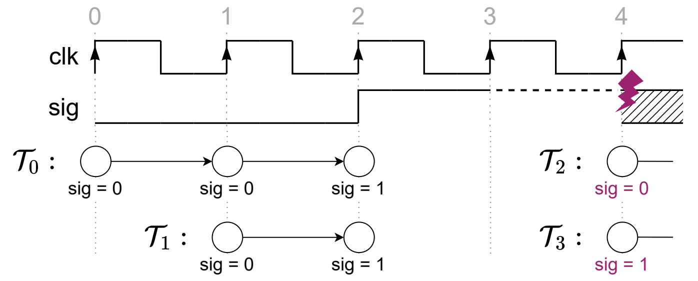

Fig. 2. Execution example of multiple traces covering four signal driver scenarios: single driver at cycle 0, conflict-free multiple drivers at cycles 1–2, no driver at cycle 3, and conflicting multiple drivers at cycle 4.

This simulation runtime checker can detect four scenarios for each signal, as exemplified in Fig. 2:

- **Single Driver (Cycle 0)**: A single trace drives the signal. The signal is updated accordingly.
- **Conflict-Free Multiple Drivers (Cycles 1–2)**: Multiple traces drive the signal with the same value. The signal is updated accordingly.
- **No Driver (Cycle 3)**: No trace drives the signal, resulting in an invalid state. The signal retains its previous value, while a warning indicates this gap in the specification.
- **Conflicting Multiple Drivers (Cycle 4)**: Multiple traces drive the signal with different values, leading to a conflict. The simulation is halted with an error message, reporting the inconsistency in the specification.

For each signal, we store the value and validity of this signal at the current cycle. A value is marked as *invalid* if it is not driven by any trace (as exemplified at cycle 3 in Fig. 2). Invalid signals are not a critical failure, meaning the simulation continues. Still, they indicate a gap in the specification, meaning there is a cycle where the behavior of at least one signal is undefined, similar to the *Determination Test* for the completeness of formal properties [13, Sec. 5.3.6].

C. Time-Annotation of Signal Values

Another challenge in simulating temporal logic is providing past signal values to execute $f \models \tau^l$ and Λ^l (cf. Sec. II-C). As exemplified in Fig. 1, traces precisely define which action is taken in which cycle, and from which past cycle signal values are required for the computation.

We perform a static analysis of the time-annotated USF model to determine the longest time window that will ever be accessed for each signal in $V \cup I$. Each signal for which past values are accessed is instantiated as an array of constant size, matching the accessed time window. The array elements store the value of the respective signal at consecutive cycles. Therefore, accessing a past signal value in a temporal expression is translated into accessing the respective array element. At the end of each simulated cycle, the arrays are updated to ensure the correct mapping between values and cycles.

IV. Application and Evaluation

This section presents the application of our specification format and generated behavioral simulation models. We demonstrate the effectiveness of behavioral models in detecting

TABLE I
EXPERIMENTAL RESULTS FOR THE SPECIFICATION AND SIMULATION, AS WELL AS DESIGN VERIFICATION AGAINST GENERATED PROPERTIES.

Component	USF Specification (.py)	Gen. Behavioral Model (.sv)	RTL Design (.sv)		Gen. Formal Properties (.sva)	
	Size	Avg. Sim. Time	Size[a]	Avg. Sim. Time	No. of Props.	Proof Time (min.:avg.:max.)
ALU	93 LoC	0.50 $\frac{\mu s}{\text{cycle}}$	100 FFs (6.4 kGE)	0.13 $\frac{\mu s}{\text{cycle}}$	11 (112 LoC)	0.2 s : 0.2 s : 0.2 s
SIMD Divider	483 LoC	0.43 $\frac{\mu s}{\text{cycle}}$	47 FFs (2.9 kGE)	0.50 $\frac{\mu s}{\text{cycle}}$	32 (+1)[b] (5749 LoC)	0.0 s : 31.2 s : 98.4 s
FIFO	123 LoC	0.50 $\frac{\mu s}{\text{cycle}}$	265 FFs (1.0 kGE)	0.18 $\frac{\mu s}{\text{cycle}}$	35 (399 LoC)	3.4 h : 3.4 h : 3.4 h[c]
MMIO	920 LoC	9.26 $\frac{\mu s}{\text{cycle}}$	58 FFs (4.2 kGE)	0.14 $\frac{\mu s}{\text{cycle}}$	171 (1650 LoC)	0.0 s : 0.2 s : 0.5 s
AHB Matrix	641 LoC	2.30 $\frac{\mu s}{\text{cycle}}$	183 FFs (8.0 kGE)	0.18 $\frac{\mu s}{\text{cycle}}$	207 (+12)[b] (4399 LoC)	0.0 s : 0.6 s : 1.7 s
Timer/Counter	2046 LoC	4.30 $\frac{\mu s}{\text{cycle}}$	117 FFs (1.9 kGE)	0.15 $\frac{\mu s}{\text{cycle}}$	190 (2509 LoC)	0.0 s : 0.6 s : 1.3 s
SPI Controller	1546 LoC	4.12 $\frac{\mu s}{\text{cycle}}$	90 FFs (2.2 kGE)	0.13 $\frac{\mu s}{\text{cycle}}$	129 (2987 LoC)	0.0 s : 20.0 s : 30.4 s

[a]Reported by the formal tool; not synthesis.
[b]Operational assertions generated from the USF specification with additional assumptions to constrain primary inputs.
[c]For independence from the RTL, the generated properties reimplement the FIFO state as constrained auxiliary variables, increasing the state space.

incomplete and inconsistent specifications and discuss their implications on simulation performance. From the same specification, we generate formal properties, showing the versatility of USF for specification validation and design verification.

A. Case Study

We apply our methodology to seven selected hardware components: a pipelined 32-bit Arithmetic Logic Unit (ALU), a 16-bit Single Instruction Multiple Data (SIMD) integer divider, an 8×32-bit FIFO buffer, and a 32-bit timer/counter peripheral with pulse-width modulation capabilities. The further three component specifications included noteworthy gaps and inconsistencies that were caught by the generated behavioral models and runtime checker (cf. Sec. III-B):

Memory-Mapped I/O (MMIO): A register file for runtime configuration of System-on-Chip peripherals. The registers can be accessed via software over the system bus or directly via the peripheral logic connected to the primary in- and outputs. The evaluated MMIO includes twenty register bit-fields and a 32-bit bus interface. The specification expresses the access policies and write prioritization.

While *read* and *write* traces were covered in an initial version of the specification, no explicit *hold* traces were specified to ensure that registers actually preserve their value. This gap was detected by the runtime checker presented in Sec. III-B, as the register values were marked as *invalid* except when explicitly driven during write accesses.

Advanced High-Performance Bus (AHB) Matrix: A fully connected crossbar with three manager and six subordinate interfaces [14]. The specified traces ensure compliance with the AHB protocol during the address and data phase, and express the crossbar's internal decoding and arbitration.

Here, the initial specification modeled the correct behavior of the *hready* output from the matrix to a manager when accessing a subordinate. However, the default value after reset was not specified, meaning the formalized specification did not fully comply with the AHB protocol. While potentially hard to detect in a static format, this issue was immediately detected during the dynamic execution of the behavioral model.

Serial Peripheral Interface (SPI) Controller: A peripheral communication device for serial data transmissions of 4-bit packets and single transmit and receive buffer registers [15]. The USF model covers the configuration modes of the SPI and specifies data transmissions by the sequential order of events on the transmission lines.

By generating and simulating the behavioral model, we identified a flaw in the specification that lacked a *Mutex*-like locking scheme, allowing two transmissions to be active simultaneously. This resulted in an access conflict on the single transmission line, which the runtime checker detected.

B. Simulation and Verification Performance

Table I lists the experimental results of applying USF to the aforementioned hardware components. For each, we show the size of the USF specification, written in Python. Besides automatically generating the behavioral models, as presented in this work, we also utilize USF to produce formal properties with an existing generator [8]. Further, Table I shows a Register-Transfer Level (RTL) implementation for each component. Lines of Code (LoC) are measured as logical source code with *cloc v2.04*. Simulations are run for 10^6 cycles with constrained random stimuli using *Verilator v5.040* and *GCC 10.3.0* with *O3* optimization. For formal verification, we use Cadence *Jasper FPV v2025.03*. All experiments are run on a 64-bit Linux machine with an AMD EPYC 74F3@3.2 GHz using eight threads and 32 GB of memory.

Comparing the average simulation times of the behavioral models in column three with corresponding RTL simulations in column five of Table I, shows a performance benefit for the RTL simulations. However, it is important to reemphasize that the behavioral models are generated from time-annotated specifications and are therefore cycle-accurate. Thus, they exhibit a similar level of detail as RTL simulations. This enables a highly accurate analysis of the specifications at the cost of a computational overhead. Another overhead comes from the runtime checker, which analyzes each signal at every cycle (cf. Sec. III-B). Still, since the measured simulation times are not prohibitively long, we consider the simulation performance acceptable. Sec. IV-C further discusses the trade-offs made in our simulation approach.

The last column presents the results of verifying the RTL designs against the generated properties. The proofs are generally efficient, with most properties being verified in under a second. Notably, the FIFO properties require a longer proof time as

they reimplement the system state with constrained auxiliary variables to ensure independence from the RTL design. This increases the state space, leading to a longer verification time.

C. Specification, Simulation, and Code Generation Benefits

This work presents an approach for early validation of hardware specifications by (i) use case simulation of generated behavioral models, and (ii) automated runtime checks in these simulations to detect gaps and inconsistencies. Therefore, we made two deliberate trade-offs in our simulation approach. First, we construct cycle-accurate behavioral models from time-annotated specifications, making the simulations highly accurate while naturally limiting performance. Second, we integrate a runtime checker to detect gaps and inconsistencies in the specifications. This checker analyzes all specified signals at every cycle, introducing a computational overhead while ensuring the automated detection of undefined signals and specification conflicts. Both choices prioritize accuracy and automated checking over performance. While applications in fast prototyping or system simulation would prioritize performance and apply different simulation approaches, our behavioral models specialize in specification validation, as successfully demonstrated in Sec. IV-A.

Further, the behavioral models are automatically generated from the specification with a reusable code generator. This generator is only specific to the USF format and, therefore, applies to all presented components, as well as future specifications. Likewise, we also construct the presented formal properties with a reusable code generator. The capability to write a single specification and use code generators to produce multiple design artifacts offers a significant reduction in effort and time compared to manually writing component-specific simulations and properties. Moreover, a single source specification ensures consistency between the generated artifacts.

Finally, being able to execute and validate the specification helps catch issues early in the development process before an RTL design is available. Even partial specifications can be executed earlier to refine the specification as it is written.

V. Related Work

Similar to USF, ILA generalizes ISA concepts for formal specifications of digital systems [9]. To bridge the abstraction gap between high-level ILA models and RTL designs, the ILA methodology relies on *valid-ready* handshakes in contrast to the time-annotation of USF with traces [16]. Further, Xing et al. present *ILAtor* to translate static ILA models into executable, untimed simulation models [17]. While this approach yields high simulation performance, the lack of timing and parallelism in instruction models limits the automatic detection of multiple driver conflicts or undefined signals.

Conventional behavioral modeling is often performed by system-level simulations, implemented as custom hardware models, using tools and frameworks such as SystemC and gem5 [18–20]. While they can achieve high simulation speeds with manual, component-specific optimizations and by exploiting abstraction with transaction-level and instruction set simulations, such models generally lack support for formal design verification [21]. Hence, they are suitable tools for design space exploration and early prototyping, but do not ensure that the later—and potentially separately—conceived design verification references indeed match these early prototypes.

Ludwig et al. address this gap between system-level simulation models and formal verification references by generating formal properties from SystemC simulation models, using *path predicate abstraction* [22]. Our USF methodology, in contrast, begins with an initially static specification that instantiates a predefined, structured specification format, or metamodel (cf. Sec. II). It is this metamodel that enables extensibility towards generating further design artifacts from a single specification, beyond simulation models and formal properties.

Oddos et al. perform *assertion-based synthesis* as an orthogonal approach to conventional assertion-based verification [23]. Their tool *Horus* generates synthesizable RTL logic from temporal logic assertions and introduces a solver circuit to detect inconsistent signal drivers from two or more discretely specified assertions. We adopt this solver concept into a software-based runtime checker to detect both gaps and inconsistencies in our models (cf. Sec. III-B).

In recent years, Large Language Models (LLMs) have been increasingly used across the entire development process of integrated systems. While LLMs are powerful tools for code generation and summarization, they introduce uncertainty. Prompting LLMs with natural language faces the same challenges as writing a good informal specification and potentially generates faulty design artifacts. Still, in the context of formalizing hardware specifications, two noteworthy approaches address uncertainty and bugs in LLM-generated code. Qui et al. present an LLM-based validation workflow for hardware simulation testbenches using a form of mutation testing [24]. Mendoza et al. use LLMs to generate temporal logic assertions from natural language specifications by decomposition into sub-translations for simplification and failure analysis [25].

VI. Conclusion

The specification of modern integrated systems poses significant challenges in writing and interpretation. In particular, informal natural language documents are prone to errors. This work, in contrast, uses formal specifications to define general digital hardware components. Based on this format, we present a code generator that automatically transforms formal specifications into behavioral models. Executing these models for use case simulation is our primary tool for validating that the specification accurately expresses the expected system behavior. Furthermore, the generated simulation models include runtime checks that analyze all updates to the global system state and automatically detect gaps and inconsistencies in the specification. While our presented application examples indicate that the simulation performance of the generated behavioral models can still be improved, the fact that they are automatically generated from the same source used to create formal properties for design verification is a significant effort and time reduction.

References

[1] H. D. Foster, "2024 wilson research group IC/ASIC functional verification trend report," Siemens EDA, 2025.

[2] H. D. Foster, "2024 wilson research group FPGA functional verification trend report," Siemens EDA, 2025.

[3] S. Ray, I. G. Harris, G. Fey, and M. Soeken, "Multilevel design understanding: From specification to logic," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016.

[4] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern SoC design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[5] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "Nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," in *Computer Aided Verification*, C. Enea and A. Lal, Eds., ser. Lecture Notes in Computer Science, Springer Nature Switzerland, 2023, pp. 383–396.

[6] R. Kunzelmann, E. Baerens, D. Gerl, M. Bhadra, N. Schwarz, and W. Ecker, "A universal specification methodology for quality ensured, highly automated generation of design models," in *2024 27th Workshop on Methods and Description Languages for Modeling and Verification of Circuits and Systems (MBMV)*, VDE Verlag, 2024, pp. 90–98.

[7] K. Devarajegowda, "Model-based generation of assertions for pre-silicon verification," Ph.D. dissertation, Technical University of Kaiserslautern, 2021.

[8] R. Kunzelmann, A. Sridhar, D. Gerl, L. V. Boga, and W. Ecker, "Automated generation of interval properties from trace-based function models," in *2024 Design and Verification Conference and Exhibition United States (DVCON US)*, Accellera Systems Initiative, 2024.

[9] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 1, 2018.

[10] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds., Dordrecht: Springer Netherlands, 2008, pp. 129–146.

[11] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*, 2nd ed. Springer US, 2004, ISBN: 978-1-4020-8027-2.

[12] E. Boerger and R. Staerk, *Abstract State Machines*, 1st ed. Springer Berlin Heidelberg, 2003, ISBN: 978-3-540-00702-9.

[13] J. Bormann, "Complete functional verification," Ph.D. dissertation, Technical University of Kaiserslautern, 2009.

[14] Arm Limited. "Multi-layer ahb v2.0 technical overview," Accessed: Jan. 16, 2026. [Online]. Available: https://developer.arm.com/documentation/dvi0045/latest/

[15] S. C. Hill, J. Jelemensky, and M. R. Heene, "Queued serial peripheral interface for use in a data processing system," Patent US4816996A, 1989.

[16] Y. Xing, H. Lu, A. Gupta, and S. Malik, "Compositional verification using a formal component and interface specification," in *2022 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022.

[17] Y. Xing, A. Gupta, and S. Malik, "Generalizing tandem simulation: Connecting high-level and RTL simulation models," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 154–159.

[18] P. R. Panda, "SystemC: A modeling platform supporting multiple design abstractions," in *Proceedings of the 14th International Symposium on Systems Synthesis*, ser. ISSS '01, Association for Computing Machinery, 2001.

[19] F. Bacchini et al., "TLM: Crossing over from buzz to adoption," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07, Association for Computing Machinery, 2007.

[20] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

[21] K. A. Rudkowski, S. Ahmadi-Pour, and R. Drechsler, "Comparing methods for the cross-level verification of systemc peripherals with symbolic execution," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[22] T. Ludwig, J. Urdahl, D. Stoffel, and W. Kunz, "Properties first—correct-by-construction RTL design in system-level design flows," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 3093–3106, 2020.

[23] Y. Oddos, K. Morin-Allory, and D. Borrione, "From assertion-based verification to assertion-based synthesis," in *VLSI-SoC: Technologies for Systems Integration*, J. Becker, M. Johann, and R. Reis, Eds., Springer Berlin Heidelberg, 2011, pp. 94–117.

[24] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "CorrectBench: Automatic testbench generation with functional self-correction using LLMs for HDL design," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025.

[25] D. Mendoza, C. Hahn, and C. Trippel, "Translating natural language to temporal logics with large language models and model checkers," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024.