Compositional Verification Using a Formal Component and Interface Specification

Yue Xing*, Huaixi Lu*, Aarti Gupta, and Sharad Malik
Princeton University, Princeton, USA
yuex@princeton.edu, huaixil@princeton.edu, aartig@cs.princeton.edu, sharad@princeton.edu

***Abstract*—Property-based specification such as SystemVerilog Assertions (SVA) uses mathematical logic to specify the temporal behavior of RTL designs which can then be formally verified using model checking algorithms. These properties are specified for a single component (which may contain other components in the design hierarchy). Composing design components that have already been verified requires additional verification since incorrect communication at their interface may invalidate the properties that have been checked for the individual components. This paper focuses on a specification for their interface which can be checked individually for each component, and which guarantees that refinement-based properties checked for each component continue to hold after their composition. We do this in the setting of the Instruction-level Abstraction (ILA) specification and verification methodology. The ILA methodology provides a uniform specification for processors, accelerators and general modules at the instruction-level, and the automatic generation of a complete set of correctness properties for checking that the RTL model is a refinement of the ILA specification. We add an interface specification to model the inter-ILA communication. Further, we use our interface specification to generate a set of interface checking properties that check that the communication between the RTL components is correct. This provides the following guarantee: if each RTL component is a refinement of its ILA specification and the interface checks pass, then the RTL composition is a refinement of the ILA composition. We have applied the proposed methodology to six case studies including parts of large-scale designs such as parts of the FlexASR and NVDLA machine learning accelerators, demonstrating the practical applicability of our method.**

I. Introduction

Formal verification of hardware is performed by checking an implementation (typically an RTL model) against a formal specification. These specifications are typically provided as a set of properties in SystemVerilog Assertions (SVA) [1] or property specification language (PSL) [2]. These properties use mathematical logic (e.g., linear temporal logic [3]) and are formally verified on the implementation model using a model checker [4]. The properties are specified for a single component in the design (which may contain other components in the design hierarchy). In this paper, we focus on properties of individual components that prove that their RTL implementations are *refinements* of high-level specifications – we refer to these as refinement-based properties.

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. This research is also funded in part by NSF award number 1628926, XPS: FULL: Hardware Software Abstractions: Addressing Specification and Verification Gaps in Accelerator-Oriented Parallelism, and the DARPA POSH Program Project: Upscale: Scaling up formal tools for POSH Open Source Hardware.

*These authors contributed equally to this work

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA © 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…
https://doi.org/10.1145/3508352.3549341

Composing components that have already been verified as correct refinements requires additional verification since incorrect communication at their interfaces may invalidate the refinement-based properties already checked for the individual components. *This paper proposes a specification for their interfaces that can be provided and checked individually for each component. The checks guarantee that the refinement-based properties checked for each component continue to hold after their composition.* We do this in the setting of the Instruction-level Abstraction (ILA) specification and verification methodology.

The ILA methodology provides a uniform specification for processors, accelerators and general modules at the instruction-level, and the automatic generation of a complete set of correctness properties for checking that the RTL model is a refinement of the ILA specification. The ILA is a generalization of the instruction set architecture (ISA) of processors. Huang et al. [5] proposed the ILA for accelerator designs by modeling the commands on the MMIO (memory-mapped-input-output) interface of accelerators as instructions which update the architectural state variables. As with processors, the architecture state variables are those that are persistent across instructions. Xing et al. [6] further generalized the ILA instruction-level modeling to general hardware modules. The commands received at the inputs to a module are treated as instructions that update the architecture state variables (i.e., variables that are persistent across the commands). The ILA methodology supports *refinement-based verification* by auto-generating per-instruction correctness properties from the ILA specification for checking the RTL implementation. Essentially, each property checks that if the ILA specification model and the RTL design start in states that correspond (according to a refinement map), then after an instruction executes to completion, the resulting states also correspond.

So far the ILA refinement-based verification methodology has been applied to a single component, e.g., a RISC-V core or a cache module. However, when verified RTL modules are composed by connecting their corresponding pins, incorrect communication between them may invalidate the refinement checks, which assumed correct inputs. This would therefore require additional verification of the composed RTL models, including possibly re-verifying properties for each component. This is undesirable – we would like the per-component refinement checks to continue to hold following the composition.

We address this problem by proposing a *compositional* specification and verification methodology using ILA models for individual components and generating *interface checks* to ensure correct communication by each individual component. The goal of this methodology is to guarantee that if each RTL component (RTL1, RTL2) is a refinement of its respective ILA specification (ILA1, ILA2) and the interface checks pass, then the RTL composition is a refinement of the ILA composition. We refer to this methodology as *ILA-based compositional refinement*.

However, there are some challenges in meeting this goal:

- Challenge 1 – Interface specification: The existing ILA specification focuses on a single component and lacks the specification of interface behavior.
- Challenge 2 – Compositional refinement checking: The refinement checking for individual RTL models only checks the state variables corresponding to the ILA specification at specific times (e.g., when an instruction commits), which is inadequate to check the communication with other components. Unlike processors, where the architectural state variables are globally visible only after the commit point, the interface signals for general modules are visible to other components at all time steps, i.e., even before instruction completion. Thus, additional checks are needed before instruction completion.

We address these challenges by first extending the ILA specification with an *interface specification* (via valid-ready handshake signals) for inter-ILA communication. Next, we generate corresponding *interface checks* (SVA properties) to ensure that inter-RTL communication correctly implements inter-ILA communication. The interface checks contain two parts: (1) checks for interface signals at the end of instruction, which become part of refinement checking, and (2) checks for interface signals before instruction completion (§III-D). Note that these interface checks are targeted to verify only the communication between components. Note also that our overall approach includes checking the implementation of each component against its *functional* specification and its *communication interface* specification, where the functional aspect is separately handled via specification and verification of valid instructions for each module. This is in contrast to well-known assume-guarantee reasoning [7], where the two kinds of specifications are combined in environment assumptions, which capture the inputs/input-sequences that are valid at the interface of a component.

As with refinement-checking in the ILA methodology, the interface specification and interface checking are done per component on its ILA and RTL models. This leverages design modularity in the implementation and the specification to enable modular verification, thereby improving the scalability of verification. Note that our methodology works with a given modular design and does not propose how to partition a design. We demonstrate the practical benefits of our proposed methodology through six case studies.

Overall, this paper makes the following contributions:

- We propose a new methodology leveraging Instruction Level Abstractions (ILAs) for compositional specification and verification that enables *compositional refinement*, i.e., if individual RTL implementations are refinements of their corresponding ILAs and their interface checks pass, then their composition is a refinement of the composition of their individual ILAs.
- To verify the interactions between components in the implementation, we include an interface specification (via valid-ready handshake signals) in the ILA models (§III) and perform additional interface checks (§III). This provides the basis for compositional refinement checking.
- We have implemented our compositional modeling and verification methodology and demonstrated its effectiveness through six different case studies (§IV). These case studies are parts of real designs: an 8051 microprocessor [8], a secure SoC comprising an 8051 and an AES accelerator [9], FlexASR Processing Element (PE) [10], NVDLA convolution core [11], an off-chip communication protocol used in BaseJump STL [12], and AMBA AXI on-chip communication modules [13]. For several of these case studies our method found bugs in the RTL implementation that were confirmed by designers.

II. Background: ILA Models

A. ILA Specification

The Instruction Level Abstraction (ILA) is a generalization of the Instruction Set Architecture (ISA), which serves as a specification for processors. An ISA specifies:

- the architectural state variables for a processor, i.e., the state variables that persist between instructions
- the decode condition for each instruction
- the architectural state update for each instruction.

There have been several successful efforts in processor verification that check an implementation instruction-by-instruction against a formal ISA specification [14]–[17].

The ILA specification [5] was introduced to extend the notion of an ISA to accelerators. It does so by treating the *commands* at the interface of the accelerator as "instructions." The ILA specification and ILA-based verification methodology were further extended for specification and verification of general hardware modules [6]. In this paper, we further leverage this notion of treating commands at the interface of a general hardware module as instructions to also model component interactions.

As introduced in [5], an ILA model of a component is represented as a five-element tuple: $\langle S, W, S_0, D, N\rangle$, where S, W denote the vectors of state and input variables, respectively, and S_0 is a vector of initial values of the state variables. The set of instructions J is associated with the sets D and N. D is a set of decode functions (each specifies a condition for triggering an instruction, i.e., the interface command), and N is a set of next state functions (each describing the state update performed by an instruction) for each instruction $j \in J$, respectively. Formally, an ILA model A is defined as follows:

$$
\begin{aligned}
\text{A} \quad = \quad & \langle S, W, S_0, D, N\rangle, \text{ where} \\
S \quad & \text{is a vector of state variables (state space: } \mathbb{S}) \\
W \quad & \text{is a vector of inputs variables (input space: } \mathbb{W}) \\
S_0 \quad & \text{is a vector of initial values of the state variables} \\
D \quad & = \{D_j : (\mathbb{S} \times \mathbb{W}) \rightarrow \mathbb{B}, j \in J\} \text{ is a set of decode functions, } \quad \mathbb{B} = \{0, 1\} \\
N \quad & = \{N_j : (\mathbb{S} \times \mathbb{W}) \rightarrow \mathbb{S}, j \in J\} \text{ is a set of next state functions}
\end{aligned}
$$

Note that this ILA definition focused on a single module specification. It did not consider any specification for communicating with other modules. Filling this gap via an interface specification is one of the contributions of this paper (§III).

B. ILA-based Refinement Verification

For performing a refinement check, the ILA methodology automatically generates a set of verification properties – one per instruction – by using a user-provided *refinement map*. Essentially, the refinement map specifies *what* to check and *when* to check for equivalence of corresponding states, since the ILA and RTL models are at different levels of abstraction and one step at the ILA level may correspond to multiple steps at the RTL. Intuitively, each property (called a commutating diagram correctness property [15]) checks that when the ILA specification and the RTL implementation start in equivalent corresponding states (as specified in a refinement map) at the start of an instruction, then after the instruction finishes execution (as specified in a refinement map), the resulting corresponding states are also equivalent. Refinement maps can also handle checking the correctness of a pipelined hardware implementation against a sequential ISA/ILA [5], [15], [16]. The per-instruction properties that are generated by ILA-based refinement verification can be checked using standard open-source [18] or commercial model checking tools [19]. In the rest of this paper, we will use $RTL_i \triangleleft ILA_i$ to denote that RTL_i is a refinement of ILA_i (for component i) and RC to denote performing a refinement check for each component in a design.

It is worth emphasizing that other existing methodologies or tools do not provide *automated generation of a complete set of properties for refinement checking* for hardware modules other than processors. Thus, the ILA component specifications are very valuable for this purpose and enable leveraging standard model checkers for verification of processors, as well as accelerators and general modules.

III. Compositional Verification

This section describes our proposed ILA-based compositional verification methodology. It starts with a motivating example which demonstrates the challenge with reasoning about composed designs, followed by an overview of the existing ILA modeling and verification of individual components for that example. Then, we introduce the interface specification which we add to the existing ILA specification, and show how it captures synchronous communication between ILA models. Finally, we introduce additional per-component interface checks based on this interface specification. Their combination with the existing refinement checks guarantees compositional refinement, which establishes correctness of the composition of RTL modules.

A. Motivating Example

We start with a motivating example of the BaseJump offchip protocol design [12]. This protocol has two components: an upstream controller module (which is in the `Upstream` chip) and a downstream controller module (which is in the `Downstream` chip) as shown in Figure 1. The data communication is uni-directional from `Upstream` to `Downstream`, where the transfer is limited to 8-bit words at a time. Accordingly, a 64-bit input $data\_in$ in `Upstream` is transferred in a four-step pipeline to `Downstream` via 8-bit channels $data\_o\_0$ and $data\_o\_1$, and then sent out as 64-bit output $data\_out$ by `Downstream`. The design uses a token-based protocol to coordinate the two modules and to ensure no loss of data. A user-provided set of properties or an automated ILA approach [5] (which we use) can be used to ensure correctness.

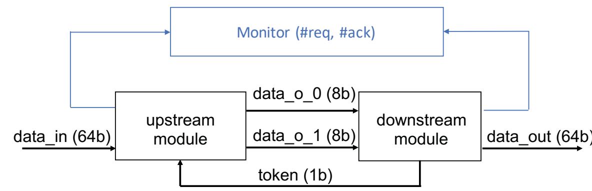

Fig. 1: BaseJump Off-chip Protocol Design (design in black; monitor used by the verification procedure in blue).

TABLE I: Time/Memory Usage of the Monolithic Verification the Off-chip Protocol Design.

Verification Task	Design Size	Time	Memory
Property (1)	7478 LoC / 3187 state bits	bounded proof 399 steps in 24 h	682 MB

Here is an example property expressed in SVA:

$$assert\{\#req >= \#ack\} \quad (1)$$

In this property, the signals req and ack are two monitor signals where req is set to high when there is some data d input to the upstream module, while ack is set to high when the same data d is output from the downstream module. Thus, Property 1 checks that the number of data output from downstream module does not exceed the number of data input into the upstream module. Table I shows the time and memory usage of checking this property using JasperGold, a commercial model checker [19] on the full design. (Any model checker may be used for this purpose.) This property is not fully proved even with a 24-hour time limit, i.e., the model checker fails to provide an *unbounded* proof, although the bounded model checking (BMC) engine provides a bounded proof with no bug up to 399 cycles. Note that *monolithic verification* of the full design (with the upstream and downstream control modules *together*) poses a scalability challenge to a state-of-the-art model checker. As we will show later, a *compositional verification* of the upstream and downstream control modules using the proposed ILA-based methodology exploits design modularity during verification, and thereby improves scalability.

B. ILA Modeling and Verification of Individual Components

The first step of our approach is to leverage the existing ILA modeling and verification techniques [6] for individual components. The ILA models for individual RTL modules of BaseJump are shown in Figure 2. These models are instruction-level specifications for the individual RTL modules. Each module has the *DATA_IN* and *DATA_SEND* instructions to indicate when the data comes in and goes out, respectively. The *TOKEN*, as introduced in § III-A, is included in the upstream and downstream ILA models (*TOKEN_SEND* instruction in the downstream module, and *TOKEN_IN* instruction in the upstream module).

With the per-module ILA model as a specification, the ILA-based verification methods (§II-B) can be applied to verify

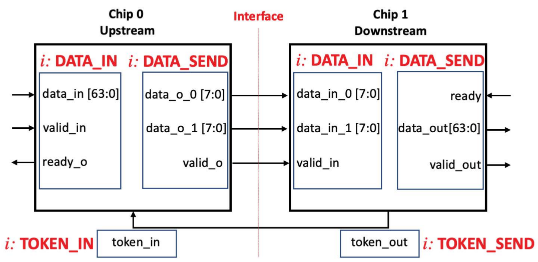

Fig. 2: Instructions and Interface Signals for the Off-chip Communication Protocol. i:DATA_IN, i:DATA_SEND, i:TOKEN_IN, i:TOKEN_SEND are the instructions. Data_in(64), valid_in(1) etc. are the interface signals (bit-width shown inside the parentheses).

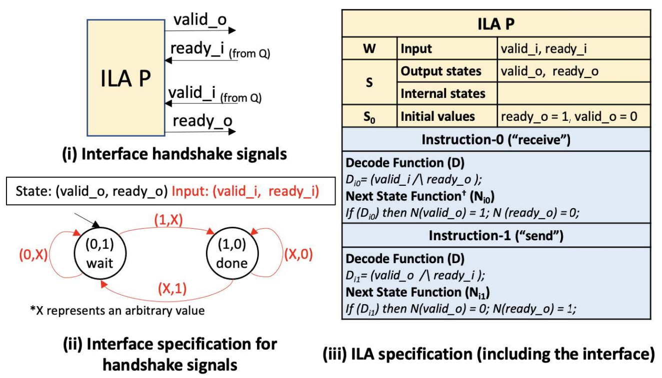

Fig. 3: ILA Interface Specification: Handshake Signals and Interface Instructions.

each RTL module, i.e., that each RTL module refines the corresponding ILA model. However, since the per-module ILA *does not specify any interface for communicating with other ILAs*, the two ILAs for the upstream and downstream modules by themselves do not provide a complete specification for the full RTL design composed of the two modules. It is this gap that we fill through the *ILA interface specification*.

C. Compositional Refinement using ILA Models

1) Augmenting the ILA model with the Interface Specification: To support the composition of ILA models, we first define *outputs* in ILA models. This allows connecting the outputs of an ILA model to the inputs of another ILA model to enable their communication. Then, we consider the interface between two models, as defined by their inputs and outputs.

In our study of designs ranging from HLS-generated designs [20] to manually implemented RTL designs [12], we noted that most modules use simple handshake signals for correct communication. Motivated by this observation, we specify an interface between two modules in terms of two handshake signals – *valid* and *ready* – in the outputs/inputs of ILA models, as shown in Fig. 3 (i). Note that these signals may be implemented in different ways (e.g., *ready* signals may always be high in some design, or may only be high when *valid* is low in other designs) – for now we focus on the *specification* of this classic handshake mechanism. Further, an ILA model may have many channels in its interface, where we consider a channel as connecting the output of one ILA to the input of another ILA. We model each channel using a separate pair of valid/ready handshake signals.

Intuitively, a valid signal indicates that the output of an ILA model is valid, while a ready signal indicates that an ILA model is ready to read its input. The data transferred through the channel is referred to as the payload and we require that the payload can be transferred only when both the valid and ready signals for the channel are high (i.e., active). When there are multiple channels for transferring data (e.g., multiple valid/ready or one valid/multi-ready or multi-valid/one ready), the specification (ILA) must decide how to resolve this through its instructions' decode and state update functions. For example, if there is one valid and multiple ready signals, the ILA specification must decide whether to wait for all ready signals or only one of them to be high, by using a suitable decode condition of the handshake signal that requires all or one ready signals to be high, respectively. This allows specifications with an arbitrary number of valid/ready signals.

More formally, we define the augmented ILA model A as:

$$A = \langle S, W, O, S_0, D, N \rangle, \text{ where}$$

- S is a vector of state variables (state space: $\mathbb{S}$)
- W is a vector of input variables (including valid/ready, input space: $\mathbb{W}$)
- O is a vector of output variables (including valid/ready, output space: $\mathbb{O}$, $O \subseteq S$
- S_0 is a vector of initial values of the state variables,
- D $= \{D_j : (\mathbb{S} \times \mathbb{W}) \rightarrow \mathbb{B}, j \in J\}$ is a set of decode functions, $\mathbb{B} = \{0, 1\}$
- N $= \{N_j : (\mathbb{S} \times \mathbb{W}) \rightarrow \mathbb{S}, j \in J\}$ is a set of next state functions

2) Interface Specification using Handshake Signals: Conceptually (although implementations vary), a *valid* signal is set to high when a module is prepared to send the payload to another module; a *ready* signal is set to high when a module is prepared to receive the payload from another module. A payload is transferred from one module to another only when *valid* and *ready* are both set to high in the respective modules.

We model the interface specification using such handshake signals in the ILA specification of a component. Note that in this setting, the payload received by an ILA is an instruction for that ILA with associated data values. An example ILA component that includes four handshake signals is shown in Fig. 3(i), where $valid\_o$ and $ready\_o$ are outputs of this component, say P, and $valid\_i$ and $ready\_i$ are inputs from another component, say Q. P and Q are communicating with each other based on these handshake signals. Here we focus on the handshake signals and omit the payload associated with the handshake. In this example, we assume that module Q has the same specification as model P.

In Fig. 3 (ii), we show an example interface specification for the handshake signals in ILA component P, i.e., how $valid\_o$ and $ready\_o$ (the output variables labeled in each state) are updated by P depending on its current state and its inputs $valid\_i$ and $ready\_i$. In the state "wait," P is ready to receive a new instruction. If P sees a valid input ($valid\_i$) from component Q (i.e., a possible new instruction at its interface), then it will decode and execute the instruction, and transition to the state "done." In the "done" state, P's valid output ($valid\_o$) is high while its ready output ($ready\_o$) is low, indicating that P can send results (from its recently executed instruction) to Q, but it is not yet ready to receive a new instruction from Q in this example. P will wait in this state (self-loop) as long as Q is not ready. When Q indicates that it is ready (and receives the payload from P), then P can transition back to its "wait" state where it is ready to receive a new instruction. Note that in this specification, an instruction is executed by P along its transition from "wait" to "done," while an instruction is executed by Q along P's transition from "done" to "wait." Next, we discuss how these instructions are modeled along with the interface specification.

3) Instructions with Handshake Operations: We now describe how the interface specification is modeled in the form of *instructions with handshake operations* in the ILA models. In particular, Fig. 3 (iii) shows the ILA specification (including the interface) for a module P (the same as for module Q). For ease of discussion, we focus only on the handshake signals in module P; other outputs in the interface simply carry the payload but are not involved in synchronizing the communication. Based on the handshake signals, we define two instructions in the ILA model – the first has a "receive" operation (corresponds to the transition from state "wait" to state "done"), and the other has a "send" operation (corresponds to the transition from state "done" to state "wait").

With these two instructions, an ILA model P can correctly communicate with an ILA model Q when their respective instructions with "send" and "receive" operations are *synchronized*, i.e., if the second instruction with "send" is decoded in the sender P's ILA model, the first instruction with "receive" is decoded in the receiver Q's ILA model *at the same time*. This synchronicity condition guarantees that the payload is correctly transferred from ILA model P to ILA model Q.

We would like to emphasize that although this handshake specification resembles a standard handshake between *asynchronous* concurrent processes, i.e., processes that may not operate synchronously, our goal here is to adapt it for specification of *synchronous* components that are implemented in RTL. Thus, it is important to identify and specify the synchronicity condition that ensures correct communication between RTL modules in the implementation.

More generally, we include the handshake operations "send" and "receive" *as part of instructions* in an ILA model, where the other part of each instruction captures its associated functional requirement. For an instruction with a "receive" operation, its decode function includes the condition that $valid\_i \wedge ready\_o$; and for an instruction with a "send" operation, its decode function includes the condition that $valid\_o \wedge ready\_i$. Note that the decode function would also include other conditions on input and state variables to trigger state update functions according to the functional requirement. The synchronicity condition ensures that whenever two ILA models communicate there is an instruction with a "send" operation decoded in the sender ILA model *and* an instruction with a "receive" operation decoded in the receiver ILA model.

In this way, our strategy for specifying a component interface in terms of handshake operations allows them to be easily incorporated into instructions in the ILA model for each component. Thus, similar to the original ILA model that specifies how the architectural state variables are updated by instructions, the augmented ILA model specifies how the architecture state variables *and* interface handshake signals are updated by instructions with handshake operations. Effectively, the augmented ILA model ensures that each component executes a new instruction only when the interface handshake signals have specific values, e.g., some new instruction can be received and executed by a component only after its previous instruction with send operation has been received by the other component(s). Importantly, by considering the handshake signals as inputs/outputs at the interface of a component, the overall problem of specifying communication between components in a system is decomposed into a *modular interface specification* for each component. *This modular specification is critical in enabling modular per-component verification, thereby improving verification scalability.*

4) ILA Composition: In the setting of this paper, we view an ILA model as a Moore finite state machine (FSM) where $O \subseteq S$. Thus, a composition of ILA models is a standard composition between interacting FSMs, where an output of one FSM can be connected to an input of another FSM. More formally, consider two ILA models $A1 = \langle S1, W1, O1, S1_0, D1, N1\rangle$ and $A2 = \langle S2, W2, O2, S2_0, D2, N2\rangle$. The parallel composition C of $A1$ and $A2$, is an FSM $C : A1 \parallel A2 = \langle S_C, W_C, O_C, S_{C0}, \delta_C\rangle$, defined as follows:

$$\begin{aligned}
S_C &= S1 \times S2\\
W_C &= W1 \cup W2 \setminus ((W1 \cap O2) \cup (W2 \cap O1))\\
O_C &= O1 \cup O2 \setminus ((W1 \cap O2) \cup (W2 \cap O1))\\
S_{C0} &= S1_0 \times S2_0\\
\delta_C &: (S_C \times W_C) \rightarrow S_C \text{ is the state transition function.}\\
&\delta_C((S1, W1), (S2, W2)) = (S1', S2'),\text{where}\\
&S1' = \begin{cases} N1_j(S1, W1) & \text{if } \exists j. D1_j(S1, W1) = 1\\ S1 & \text{otherwise}\end{cases}\\
&S2' = \begin{cases} N2_k(S2, W2) & \text{if } \exists k. D2_k(S2, W2) = 1\\ S2 & \text{otherwise.}\end{cases}
\end{aligned}$$

Each state of the composition C is a pair comprising the states of $A1$ and $A2$ in the usual way. The state transition function δ_C updates each part of this pair if there exists an associated instruction (j for $A1$, k for $A2$) whose decode condition is true. Thus, each transition in C corresponds to the execution of an instruction in one or both components. *Note that by including the synchronicity condition for handshake operations in the decode conditions of the specified instructions, we ensure that communication between ILA models happens synchronously.*

This definition generalizes in a straightforward manner to a composition of n ILA models. An FSM for $C : A_0 \parallel A_1 \parallel \ldots \parallel A_{n-1}$ can be constructed where the state of the composition is a vector comprising the states of A_0, A_1, ..., A_{n-1}. A payload transfer between any pair of ILA occurs when a send instruction in one component and a receive instruction in the other are synchronized in the composition.

D. Compositional Refinement with Interface Checking

Recall that in RC, when $RTL_i \triangleleft ILA_i$, the RTL component RTL_i and its ILA specification ILA_i are shown to have equivalent outputs at corresponding points specified in a given refinement map provided by the user (§II-B). The augmented ILA models presented in this paper include interface instructions that specify the updates to the handshake signals according to the interface specifications. We can then use the standard ILA-based refinement verification methodology [6] to perform the component refinement checks, which now include checking the handshake signals at the end of each instruction – this forms the first part of interface checking.

Note that checking $RTL_i \triangleleft ILA_i$ focuses on checking the equivalence of specified outputs at the *end* of each instruction, as specified in the refinement map. However, refinement checking at instruction completion points is not enough for the interface signals. Unlike processors and accelerators, where the architectural state is visible only at the end of an instruction, the handshake signals at the interface are visible at all time steps, i.e., even before instruction completion. Therefore, we also need to ensure that the interface signals have correct values *even before the instruction completion points*. Specifically, we perform the following two additional *pre-completion checks (PCCs)*:

- *PCC1*: For each RTL_i, the *valid* output is not set to high before the completion of the instruction that asserts the valid signal. This ensures that the payload is not transferred before it is available.
- *PCC2*: For each RTL_i, the *ready* output is not set to high before the completion of the instruction that asserts the ready signal. This ensures that the module is actually ready to receive the payload.

These two checks form the second part of interface checking and ensure that the payload is correctly transferred as per the ILA interface specification. Note that these checks focus on communication only and can be generated automatically. This is different from standard assume-guarantee reasoning [7] where one needs to verify given guarantees under environment assumptions for each module. As we show later §IV, the bugs that we find with these two checks can help strengthen environment assumptions in some designs. It is also important to note that properties for RC and PCC are automatically generated, providing a systematic verification methodology.

Theorem 1 [Compositional Refinement]: If for all components i, the refinement checks and the additional PCC checks on RTL_i and ILA_i pass, then the composition RTL_C : $RTL_0 \parallel RTL_1 \parallel RTL_2 \ldots \parallel RTL_{n-1}$ is a refinement of the composition $ILA_C : ILA_0 \parallel ILA_1 \parallel ILA_2 \ldots \parallel ILA_{n-1}$.

Proof Sketch: Consider a pair of interacting modules RTL_i and RTL_j, and their specifications ILA_i and ILA_j, respectively. Since $RTL_i \triangleleft ILA_i$ and $RTL_j \triangleleft ILA_j$, this ensures that the payload values match between RTL_i and ILA_i and also between RTL_j and ILA_j. Furthermore, the handshake signals in the two models match at the end of each instruction, and the additional PCCs ensure that the handshake signals are correctly implemented at all steps before the end of each instruction. Thus, the payloads between RTL_i and RTL_j match the payloads between ILA_i and ILA_j, and their transfers between RTL_i and RTL_j are implemented correctly. Therefore, the composition of RTL_i and RTL_j refines the composition of ILA_i and ILA_j. This reasoning can be applied pairwise to n components, thereby proving the claim.

IV. CASE STUDIES

There are no automated tools that directly address our problem space – the specification and verification of refinement-based properties of a composition of hardware components. Further, general property-based specification depends on properties written *manually* by a designer/verification engineer, which does not allow a head-to-head comparison with our largely automated methodology. Instead, we demonstrate the applicability and effectiveness of our proposed ILA-based composition methodology through six case studies [*]: the BaseJump off-chip communication design [12], an AXI communication design [13], an 8051 microprocessor as a composition of its sub-modules [8], a secure SoC [9] (composition of 8051 and an AES accelerator), the Processing Element in a speech recognition accelerator FlexASR [10], and the convolution core in the Nvidia Deep Learning Accelerator (NVDLA) [11].

We successfully verified all six case studies and detected some bugs that were confirmed by the designers. The open-source ILAng platform [21] was used for ILA tools and JasperGold [19] was used as the model checker. All experiments were performed on a Dell Server with a 2.3 GHz 28-core Intel Haswell processor and 224 GB of RAM, running RedHat Linux 5 OS. The experimental results for verification, including the RCs and PCCs are provided in Table II.

A. BaseJump Off-chip Link

We built the ILA models with the augmented outputs and interface signals for the upstream and downstream modules in the BaseJump [12] off-chip link design (§ III-A). During pre-completion checking (PCC) one bug was identified in the upstream module. The implementation incorrectly transferred invalid data, which is not allowed in the specification. The bug was found within 0.3s. After checking with the designers, we found that the cause was a missing requirement on the external inputs. We fixed this bug by adding environmental constraints on those inputs, after which verification for all modules completed successfully in 15 min. In comparison with user-specified property-based verification for the composed RTL design which did not complete in 24 hours (§ III-A), the ILA compositional verification methodology decomposed the original verification problem into per-component RC and PCC checks. These proof obligations were finished in reasonable time, demonstrating the verification scalability enabled by our modular methodology.

B. AXI Design

The widely-used on-chip AXI communication protocol [13] is a burst-based data-transfer protocol where the communication channels use a valid-ready handshake mechanism. Data can be transferred from a leader module to a follower module only when ready and valid signals are both asserted in one channel, as required by the handshake mechanism.

[*]Source code for all models and verification properties is available at https://github.com/yuex1994/ICCAD22_composition.

TABLE II: Experimental Results for Case Studies: Statistics of RTL Designs (Lines of Code (LoC) in Verilog, Number of State Bits), ILA Models (Number of Instructions, Lines of Code in C++ using ILAng, Number of State Bits), Refinement Maps (Lines of Code in Json using ILAng) and Verification Time/Memory for RCs and PCCs per module.

Case Study	Design Statistics			ILA Model Statistics			Verification			
	Modules	RTL Size (LoC)	# of state bits	# of instrs	ILA size (LoC)	# of state bits	Ref-Map (LoC)	Bug Found Time (s)	Proof Time (s)	Memory Usage (MB)
Off-chip Protocol	Upstream	2982	713	7	144	146	286	0.3	756.6	253.5
	Downstream	5453	2474	6	101	98	196	-	38.2	89.1
AXI OH Design	Leader	871	403	11	184	289	109	0.01	0.23	9.7
	Follower	828	372	9	167	159	77	0.01	0.11	7.8
8051 Micro-processor	Decoder	2636	30	5	479	30	63	-	0.23	19.5
	Datapath	2987	273	20	861	229	142	-	11.9	667
	Mem Interface	1096	304	12	342	220	101	-	0.79	45
Secure SoC	micr-processor	5938	645	255	723	274	716	-	2749.2	297
	AES	1217	1728	16	520	575	232	-	97.4	235
PE Module in FlexASR	PE Core	39098	9270	12	1203	1269	256	4.1	2716.4	344.1
	Activation Unit	15885	9025	20	1394	775	250	-	2284.9	587.7
Convolution Core in NVDLA	SC	101846	60874	12	385	41	139	-	109.5	63.41
	MAC	54602	72927	6	228	1609	365	-	2601	968
	ACC	22450	67032	22	443	767	442	-	362	170.1

We built four augmented ILA models: one each for reading and writing channels, in each of the leader and follower modules. We then performed the per-component RC and PCC checks. We found two bugs in the follower and one bug in the leader components through the RC. The ILA specification of each module requires that the interface data be unchanged until the receiver is ready, but the design fails to implement this feature in both the leader and the follower. Another bug in the follower read channel is that the data address should be updated based on an internal state variable instead of an input variable. These bugs were found very quickly, in about 0.01s. We confirmed the bug with the designer and fixed the bugs by keeping the interface data unchanged until the receiver is ready and correcting the address computation logic. After fixing the bugs, the follower and leader modules successfully passed all checks in 1s.

C. 8051 Microprocessor

We applied our methodology to an open-source 8051 microprocessor [8]. It comprises three modules: a decoder, a datapath, and a memory interface. The decoder receives the instruction, decodes it, and communicates with the datapath, which contains the registers for computation. The memory interface communicates with the external instruction/data memory and holds the program counter. It also communicates with the decoder for sending the instruction and receiving the branch address for the program counter. We built an ILA model for each of the three modules and applied RC and PCC. The verification for these three modules finished successfully in 0.23s, 11.9s, and 0.79s, respectively.

D. Secure SoC

We verified a design for a secure SoC [9] which includes two parts: an 8051 microprocessor and an AES encryption accelerator. The processor communicates with the accelerator through an MMIO interface with a valid/ack handshake mechanism. It can configure the accelerator, trigger a task on the accelerator, and poll it for completion. Earlier work [5], [9] has developed the ILA models (without an interface specification) for the two components. We extended these two models with outputs for MMIO interaction and the interface handshake mechanism. We performed RC and PCC for the two components, and the verification completed in less than an hour in total.

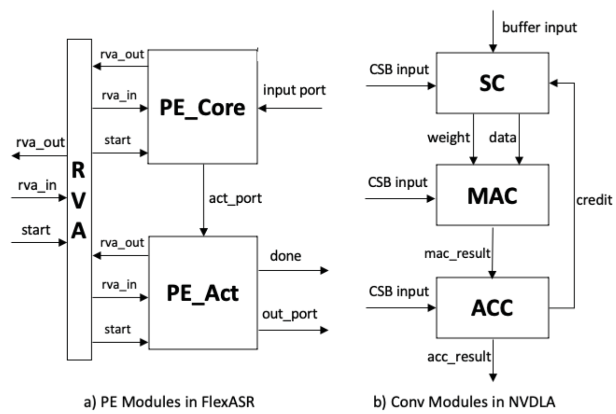

Fig. 4: Modules in FlexASR and NVDLA accelerators.

E. Processing Elements in FlexASR

FlexASR [10] is an accelerator for speech and natural language processing (NLP) tasks that supports various recurrent neural networks. As shown in Fig. 4a, a Processing Element (PE) in FlexASR mainly comprises three modules: a Ready Valid Addressing (RVA) wrapper, a PE core, and an activation unit. We abstract the RVA wrapper, since it is very simple and diverts some MMIO commands to other modules. The PE core receives input weights through the *input port* from Global Buffer (GB), while the activation unit performs vector operations on the accumulated results, outputting the final results back to the GB. We developed the ILA models and found a bug when performing RC on the PE core module. The internal state in the PE core was incorrectly updated (from *OUT* state to *IDLE* state, instead of to *PRE* state) when there is no output. On checking with the designer, we found that this bug was caused by an unsafe optimization during high-level synthesis of the design. Besides detecting this bug, we verified all the modules in the PE within 90 minutes.

F. Convolution Core in NVDLA

NVDLA [11] is an open-source configurable hardware accelerator targeting inference operations in deep learning applications. In this case study, we focus on the convolution core of NVDLA which comprises a sequencer controller (SC), a multiply-accumulate array (MAC), and a separate accumulator (ACC), as shown in Fig 4b. The CSB inputs are MMIO commands, which configure the modules' functionality (e.g., interpret data as 8-bit or 16-bit integers). These three modules are cascaded: the SC module receives inputs from outside (e.g., a buffer) and outputs the weight and data to the MAC module; the MAC sends its calculated results to the ACC module; the ACC module accumulates these values and outputs the final result to other modules outside the convolution core. The ACC module also gives the `credit` to the SC module to indicate whether the SC module can receive more values from outside. We built ILA models for each module, modeling complex arithmetic functions such as multiplication as uninterpreted functions. Verification (RC and PCC) of these three modules finished successfully in less than 1 hour.

V. Related Work

Our work is broadly related to efforts in hardware specification, interface specification, compositional verification, and protocol verification.

a) Hardware Specifications: As described earlier (§I), SystemVerilog assertion (SVA) [1], property specification language (PSL) [2], and instruction-level abstraction (ILA) [5] provide formal logic-based hardware specifications which can be used for verification. In addition, there are other high-level hardware specifications used in practice. SystemC [22] extends C++ for system-level functional models, and Transaction Level Modeling (TLM) [23] further abstracts the communication and computation for modeling hardware designs. These models help raise the level of abstraction and hence improve scalability in software/hardware co-design/simulation. However, formal hardware verification with SystemC/TLM specifications remains challenging because of the gap between the C++ language-based semantics and RTL register-transition-based semantics. Our proposed approach leverages the ILA model that captures architecture states and their updates using instructions – this enables application of well-known processor verification techniques for RTL refinement checking.

BlueSpec Verilog (BSV) [24] is a rule-based language for hardware design specification and implementation. A design is specified by guarded rules, where each rule is an atomic state-transition unit. BlueSpec relies on a scheduling algorithm to schedule multiple enabled rules. BlueSpec has been used for compositional reasoning [25]–[27], where a verified specification replaces a detailed component implementation (e.g., replacing a pipelined processor by an ISA) in the verification of a design with many components. However, due to the atomicity of rules, there is no mechanism to specify or check synchronous behavior between interacting BSV components. In contrast, our augmented ILA-based interface specification includes a handshaking mechanism that provides a synchronous semantics for component interactions.

b) Interface Specifications: The Wire Sorts language [28] also leverages interface specifications for compositional reasoning for RTL designs. However, its focus is mainly on checking types of connectivity between modules, e.g., combinational loops, and not on functional correctness. In contrast, our work formally verifies component implementations against their architecture-level specifications, and their composition via RC and PCC.

There are earlier efforts on specification of interfaces [29], [30], such as interface automata. They provide formal models for interface behaviors and theories for the composition of interface models such that these models are compatible and the composition is sound. However, these models only focus on interface behavior and the internal functionality of modules is abstracted away. In contrast, our approach includes functional verification of the RTL components and their composition through RC and PCC.

c) Compositional Verification: Compositional reasoning [31], [32] has also been applied to the verification of hardware implementations such as processor RTL designs [33]. These are similar to our approach in that they decompose the verification into sub-tasks of verifying "units of work" (the unit is similar to the component in our paper, e.g., speculative branching unit, ALU, reservation station, etc.), where each sub-task is more tractable for a model checker. However, their verification technique is based on assumptions/guarantees or mutually inductive invariants, which have to be provided by a user. In contrast, our focus is mainly on communication and synchronization of the composition of modules, where we require a user to provide a refinement mapping. As mentioned earlier, assume-guarantee reasoning [7] typically combines functional and communication requirements in environment assumptions at interfaces of modules. In contrast, we separate these two sets of requirements, specifying the former as valid instructions at an interface.

d) Protocol Verification: Protocol specification and verification have also been studied before. The CMP (Chou-Mannava-Park) method [34]–[36] uses flow-based models for protocol designs, e.g., cache coherence protocols. It addresses the scalability problem by using parameterized model checking, which abstracts a parameterized number of components (e.g., cache blocks) into a fixed and small number of components. However, these works focus on correctness of high-level protocol specifications and not on RTL implementations. Our work (with the case studies of Off-chip Protocol and AXI) fills this implementation-verification gap.

VI. Conclusions

In this paper, we propose an ILA-based compositional specification and verification methodology that supports compositional refinement. We extend the ILA model with an interface specification to model synchronous communication between ILA models. In addition to component refinement checks, we propose additional interface checks to guarantee compositional refinement, i.e., if individual RTL implementations are refinements of their corresponding ILA specification models and the interface checks pass, then their composition is a refinement of the composition of the ILA models. We have applied our proposed methodology to six case studies, all from real designs, and found bugs and/or completed refinement checking and interface checking – demonstrating the practical applicability and effectiveness of our approach.

REFERENCES

[1] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny, *SVA: The power of assertions in SystemVerilog*. Springer International Publishing, 2015.

[2] IEEE-Commission, "IEEE standard for Property Specification Language (PSL)," *IEEE Std 1850-2005*, 2005.

[3] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 1977, pp. 46–57.

[4] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[5] 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 Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 1, pp. 1–24, 2018.

[6] Y. Xing, H. Lu, A. Gupta, and S. Malik, "Leveraging processor modeling and verification for general hardware modules," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[7] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and models of concurrent systems*. Springer, 1985, pp. 123–144.

[8] S. Teran and S. Jaka, "8051 micro controller," 2016, [Online]. Available: http://opencores.org/project,8051, accessed on: 2022-04.

[9] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based synthesis of instruction-level abstractions for SoC verification," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2015, pp. 160–167.

[10] T. Tambe, E.-Y. Yang, G. G. Ko, Y. Chai, C. Hooper, M. Donato, P. N. Whatmough, A. M. Rush, D. Brooks, and G.-Y. Wei, "A $25mm^2$ SOC for IOT devices with 18ms noise-robust speech-to-text latency via Bayesian speech denoising and attention-based sequence-to-sequence DNN speech recognition in 16nm FinFET," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64. IEEE, 2021, pp. 158–160.

[11] NVIDIA, "NVIDIA Deep Learning Accelerator," 2018, [Online]. Available:www.nvdla.org, accessed on: 2022-04.

[12] M. B. Taylor, "INVITED: BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design," in *DAC*, 2018, pp. 1–6.

[13] A. Olofsson, R. Trogan, F. Huettig, O. Jeppsson, and P. Saunderson, "Epiphany eLink AXI," 2016, [Online]. Available:https://github.com/aolofsson/oh/tree/master/axi, accessed on: 2022-04.

[14] P. Manolios and S. Srinivasan, "A refinement-based compositional reasoning framework for pipelined machine verification," *TVLSI*, vol. 16, pp. 353 – 364, 05 2008.

[15] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *CAV*, 1994.

[16] P. Manolios and S. K. Srinivasan, "A complete compositional reasoning framework for the efficient verification of pipelined machines," in *ICCAD*, 2005.

[17] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with isa-formal," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 42–58.

[18] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "CoSA: Integrated verification for agile hardware design," in *FMCAD*, 2018, pp. 1–5.

[19] Cadence Design Systems, Inc., "JasperGold: Formal Property Verification App." 2018, [Online]. Available:http://www.jasper-da.com/products/jaspergold-apps/, accessed on: 2022-04.

[20] Xilinx, "Vivado Design Suite User Guide," 2021, [Online]. Available:https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis, accessed on: 2022-04.

[21] B. Y. Huang, H. Zhang, A. Gupta, and S. Malik, "ILAng: A modeling and verification platform for socs using instruction-level abstractions," in *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems conference series, TACAS 2019 held as part of the 22nd European Joint Conferences on Theory and Practice of Software, ETAPS 2019*. Springer Verlag, 2019, pp. 351–357.

[22] P. R. Panda, "SystemC: A modeling platform supporting multiple design abstractions," in *Proceedings of the 14th international symposium on Systems synthesis*, 2001, pp. 75–80.

[23] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez, "Transaction level modeling in SystemC," *Open SystemC Initiative*, vol. 1, no. 1.297, 2005.

[24] R. Nikhil, "Bluespec System Verilog: Efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 2004, pp. 69–70.

[25] A. C. Wright, "Modular SMT-based verification of rule-based hardware designs," Ph.D. dissertation, Massachusetts Institute of Technology, 2021.

[26] T. Bourgeat, C. Pit-Claudel, and A. Chlipala, "The essence of Bluespec: A core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 243–257.

[27] J. Choi, M. Vijayaraghavan, B. Sherman, and A. Chlipala, "Kami: A platform for high-level parametric hardware specification and its modular verification," 2017.

[28] M. Christensen, T. Sherwood, J. Balkind, and B. Hardekopf, "Wire sorts: A language abstraction for safe hardware composition," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 175–189.

[29] L. De Alfaro and T. A. Henzinger, "Interface automata," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 109–120, 2001.

[30] L. d. Alfaro and T. A. Henzinger, "Interface theories for component-based design," in *International Workshop on Embedded Software*. Springer, 2001, pp. 148–165.

[31] K. L. McMillan, "A compositional rule for hardware design refinement," in *CAV*, 1997, p. 24–35.

[32] D. Giannakopoulou, K. S. Namjoshi, and C. S. Pasareanu, "Compositional reasoning," in *Handbook of Model Checking*, 2018, pp. 345–383.

[33] R. Jhala and K. L. McMillan, "Microarchitecture verification by compositional model checking," in *CAV*, 2001, p. 396–410.

[34] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *2008 Formal Methods in Computer-Aided Design*, 2008, pp. 1–8.

[35] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *FMCAD*. Springer Berlin Heidelberg, vol. 3312, pp. 382–398.

[36] K. S. Namjoshi and R. J. Trefler, "Parameterized compositional model checking," in *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 589–606.