SoC Protocol Implementation Verification Using Instruction-Level Abstraction Specifications

HUAIXI LU, YUE XING, AARTI GUPTA, and SHARAD MALIK, Princeton University, USA

In modern systems-on-chips, several hardware protocols are used for communication and interaction among different modules. These protocols are complex and need to be implemented correctly for correct operation of the system-on-chip. Therefore, protocol verification has received significant attention. However, this verification is often limited to checking high-level properties on a protocol specification or an implementation. Verifying these properties directly on an implementation faces scalability challenges due to its size and design complexity. Further, even after some high-level properties are verified, there is no guarantee that an implementation fully complies with a given specification, even if the same properties have also been checked on the specification. We address these challenges and gaps by adding a layer of component specifications, one for each component in the protocol implementation, and specifying and verifying the interactions at the interfaces between each pair of communicating components. We use the recently proposed formal model termed *Instruction-Level Abstraction* (ILA) as a component specification, which includes an interface specification for the interactions in composing different components. The use of ILA models as component specifications allows us to decompose the complete verification task into two sub-tasks: checking that the composition of ILAs is sequentially equivalent to a verified formal protocol specification, and checking that the protocol implementation is a refinement of the ILA composition. This check requires that each component implementation is a refinement of its ILA specification and includes interface checks guaranteeing that components interact with each other as specified. We have applied the proposed ILA-based methodology for protocol verification to several third-party design case studies. These include an AXI on-chip communication protocol, an off-chip communication protocol, and a cache coherence protocol. For each system, we successfully detected bugs in the implementation, and show that the full formal verification can be completed in reasonable time and effort.

CCS Concepts: • **Computer systems organization → Architectures**; • **Hardware → Design reuse and communication-based design**; **Equivalence checking;**

Additional Key Words and Phrases: System-on-chip, hardware protocol specification, instruction-level abstraction, formal verification, sequential equivalence checking, refinement checking

ACM Reference format:
Huaixi Lu, Yue Xing, Aarti Gupta, and Sharad Malik. 2023. SoC Protocol Implementation Verification Using Instruction-Level Abstraction Specifications. *ACM Trans. Des. Autom. Electron. Syst.* 28, 6, Article 89 (October 2023), 24 pages.
https://doi.org/10.1145/3610292

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.
Authors' address: H. Lu, Y. Xing, A. Gupta, and S. Malik, Princeton University, 1 Nassau Hall, Princeton, NJ 08544; emails: {huaixil, yuex}@princeton.edu, aarti@cs.princeton.edu; sharad@princeton.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2023 Copyright held by the owner/author(s).
1084-4309/2023/10-ART89 $15.00
https://doi.org/10.1145/3610292

1 INTRODUCTION

Modern **Systems-on-Chips (SoCs)** are complex with a large number of modules integrated on a single chip. These modules communicate and interact with each other and with memory subsystems using standard or customized protocols. These protocols themselves are complex and need to be implemented correctly to ensure correct operation of the SoC. Thus, protocol verification has received significant attention.

1.1 Protocol Verification Overview

There are two main phases in protocol verification today (Figure 1). The first focuses on a protocol specification S, whereas the second focuses on a protocol implementation I, typically at the **Register-Transfer Level (RTL)**. Each phase focuses on a different level of protocol correctness, and both are essential for correctness assurance of the SoC.

In the first phase, a high-level protocol specification S is usually modeled using **Finite State Machines (FSMs)**.[1] The specification S may be a set of interacting FSMs—one FSM for each component (i.e. implementation module) in the protocol or a single FSM capturing the interactions of all components. Thus, the first phase of the protocol verification aims to verify correctness of the protocol specification S itself. Such formal specifications are crucial, since the implementation is later verified against this specification. Our methodology assumes that such a specification model is available or can be written. Accordingly, several high-level properties H, such as no deadlock or mutual exclusion, are verified on S. We call this verification process **High-Level Specification Verification (HLSV)**, which is largely based on property verification. There is a rich body of work in property verification of protocols: for cache-coherence protocols [15, 30, 43, 51, 56], for network protocols [1, 4, 14, 16], and for communication protocols [9, 47, 50]. There are many techniques and tools such as Murphi [13], SPIN [17], the LNT formal language [21], and TLA+ specifications [26] that are used in HLSV to perform verification and guarantee correctness of the protocol specification. Thus, for the purpose of this work, we consider this problem to be well studied and assume that all high-level protocol properties of interest have already been verified on the specification S. Our focus, instead, will be on verifying that the *protocol implementation* I is correct with respect to the specification S.

The second phase in protocol verification checks that the given RTL implementation I is correct. Typically, the high-level properties H, which were checked in HLSV on the specification, are now checked on the implementation. We refer to this process as **High-Level Implementation Verification (HLIV)**. When protocol specifications are provided as informal documents, HLSV for specifications is usually skipped in practice, and HLIV is performed to check high-level protocol level properties directly on the protocol implementation. Note that RTL implementations are more complex than high-level specifications since they include performance enhancement features such as pipelining. This added design complexity, along with the complex interactions between multiple modules involved in the protocol, poses a significant scalability challenge in HLIV.

Further, in the second phase using HLIV, even if the high-level properties H are verified on the implementation I, this does not ensure *complete* verification with respect to the formal specification S. Complete verification requires checking that each step in S is correctly implemented in I—we refer to this as **Formal Specification Based Implementation Verification (FSIV)**. Both FSIV and HLIV focus on protocol implementation verification; however, FSIV goes beyond HLIV, since verifying high-level properties H may not completely capture the full specification S. For example, the cache coherence protocols—MESI and MSI—both satisfy the mutual exclusion property,

[1] We use FSM to denote a finite state transition model, with labeled inputs and (optional) outputs. For verification, this corresponds to a labeled transition system in the standard way.

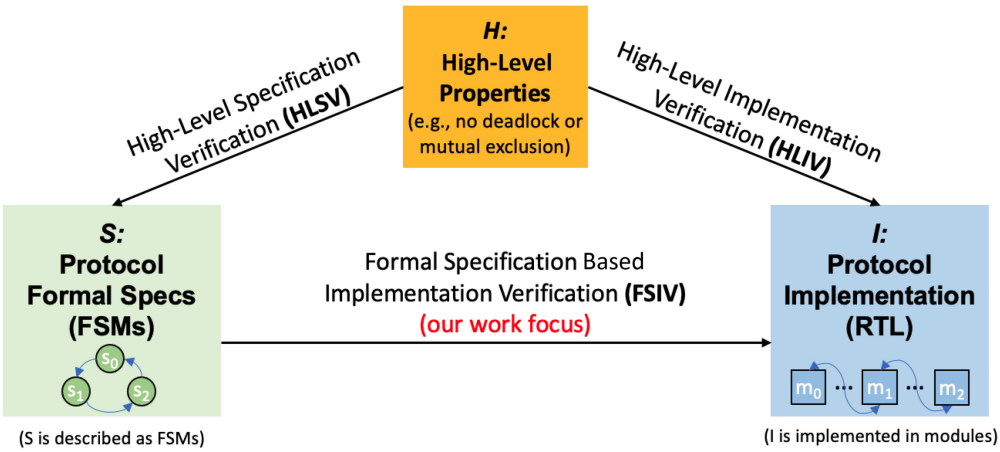

Fig. 1. Different phases of protocol verification.

but a MESI protocol implementation is not correct with respect to the MSI specification. Checking only the high-level properties H on S and I (HLSV and HLIV) cannot guarantee that I fully implements S. In practice, it is often difficult to specify a complete set of properties for FSIV. We address this gap in our article.

1.2 Challenges in FSIV

There are three key challenges in performing FSIV (assuming there is a verified protocol specification on hand).

1.2.1 Whole System Specification. The specification S of a protocol usually refers to a *whole system* instead of a modular specification for each component in the implementation. A whole system specification, such as a single FSM, specifies the full system state and transitions among them for the protocol. It does not focus on the detailed interactions among different components but rather on the protocol-level state and its transitions. However, this single FSM blurs the boundaries between individual components. For example, the states of two components may be combined into a single state variable in this FSM. In contrast, the RTL implementation of the protocol may have several distinct components, such as a separate component for each cache, and one component for the directory in a cache-coherence protocol. One approach for FSIV is to combine these components and verify their composition against the whole system specification. This relates to the second challenge in FSIV.

1.2.2 Scalability. Formal verification is quite sensitive to the size of the system state space. Whole system verification for FSIV consists of a large state space due to the cross product of the state spaces of each component in the implementation. This is a challenge for even HLIV, as the high-level properties H refer to the full system and need to be checked against the full system implementation I. This challenge is further exacerbated in FSIV, since the protocol specification contains more details than the checks for holding the high-level properties. (Both cache coherence protocols MSI and MESI satisfy the same high-level coherence properties but have different specifications, as their protocol details differ.) As a result, the verification obligations are larger for completely verifying I against the formal specification S. Further, performance enhancement features in the implementation (e.g., pipelines) add to component complexity, thereby increasing the component state space and impacting scalability.

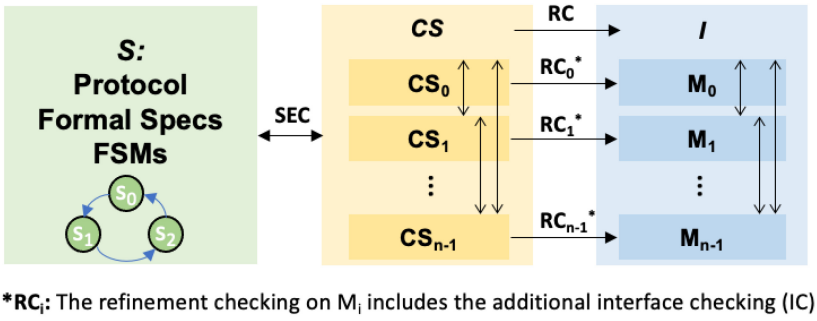

Fig. 2. Proposed ILA-based FSIV method for protocol verification.

1.2.3 Refinement Checking. Another consequence of performance-enhancing features such as pipelining is that the implementation I may have multiple steps that implement a single step in the specification S. Thus, the verification of I against S is not a cycle-by-cycle equivalence check but rather a *refinement check*, which verifies that corresponding state variables between the two models are equivalent at *specific* times, such as at the end of corresponding steps in the protocol.

Refinement Checking (RC) has been successfully used for processor verification [6, 33, 34] to verify the correctness of pipelined microarchitectures against **Instruction Set Architecture (ISA)** specifications.

The correspondence between the two is provided by a user via a refinement map, which specifies (i) a map between the specification state variables and the corresponding implementation state variables, and (ii) the correspondence points where the equivalence between these variables should be checked. Although commercial model checking tools (e.g., [7]) support **Sequential Equivalence Checking (SEC)** and property verification directly, they do not provide direct support for RC of hardware systems. Thus, the properties for RC have to be written manually.

1.3 Our Approach: Key Ideas

Motivated by these challenges, we propose a new compositional methodology for FSIV, shown in Figure 2. We introduce the component specification in the middle to bridge the gap between the high-level protocol specification and the low-level detailed RTL design.

For example, a token-based off-chip protocol (detailed in Section 4) is utilized for communication between upstream and downstream components on separate chips. However, the high-level protocol primarily focuses on the token design for data reception (write) and transmission (read) rather than specifying the upstream and downstream components individually. One write operation in the high-level specification, such as $MEM[w\_ptr] = data\_in[31:0]$, can correspond to a data processing operation ($data = data\_in[31:0]$) in the upstream component and a store operation ($MEM[w\_ptr] = data$) in the downstream component. These two abstracted operations are implemented in a more complex manner in the actual RTL design, involving buffers or pipelines. Consequently, these two operations, one each in the upstream and downstream components, can serve as a bridge between the high-level specification and the low-level RTL design.

We provide a quick overview of the key ideas.

Component Specifications. The heart of this methodology is utilizing the existing design modularity in the implementation, where we construct *formal component specifications*, denoted CS_i, for each individual component M_i in the protocol implementation. In particular, the novelty is that we use a formal model called an **Instruction-Level Abstraction (ILA)** to represent each CS_i. The ILA

has been proposed recently as an *architecture-level specification* for hardware components such as accelerators/processors [23] and other general hardware modules [58]. In this article, we utilize the compositional specification and verification methodology proposed in earlier work [59] that adds an *interface specification* (via *valid/ready* handshake signals) to the ILA model that enables compositional reasoning to ensure that the interactions between the component implementations is correct with respect to the interactions between the ILA models.

Compositional RC. For protocol verification, the ILA-based component specifications serve as a bridge between S and I, and help to decompose the overall problem of FSIV. In particular, note that I is often represented as a composition of component implementations $I = M_0 \parallel M_1 \parallel \ldots \parallel M_{n-1}$, where $\parallel$ denotes the standard parallel composition between RTL components represented as FSMs, where their inputs/outputs define the component interfaces. We view the system of interacting component specifications CS also as a composition of component specifications $CS = CS_0 \parallel CS_1 \parallel \ldots \parallel CS_{n-1}$. Each CS_i is an ILA specification that includes input/output interfaces (using handshake signals), and their composition is the parallel composition of interacting FSMs as the ILA model is an FSM. The main difference is that in the FSM for CS_i, each step (i.e., transition) denotes executing a full instruction/command at the interface, whereas in the FSM for M_i, each step denotes a state update at a clock cycle in RTL. A user-provided refinement map provides a correspondence between the two FSMs (e.g., by specifying the number of cycles or condition for completion of each instruction/command). In our methodology, we check that each M_i and its interface is a correct refinement of each CS_i and its interface, based on this given refinement map.

FSIV Verification Using ILA-Based Methodology. Our methodology decomposes the overall FSIV problem into two main verification tasks. In the first task, we check that the ILA composition CS is *equivalent* to the whole system specification S. Note that CS is at a much higher level of abstraction than I, so verifying CS against S is simpler than verifying I against S. Further, the CS_i are created such that transitions (i.e., steps) in CS correspond to transitions (steps) in S. Thus, verifying CS against S is a *sequential equivalence check* (SEC in Figure 2) rather than the more complex refinement check needed for verifying I against S.

In the second task, we check that the implementation I is a refinement of the ILA composition CS. Here, each M_i is verified against its specification CS_i (indicated as RC in Figure 2). This *includes* checking that each M_i correctly implements its interface specification in the ILA model CS_i. Including **Interface Checking (IC)** as part of RC provides a basis for *compositional* RC (i.e., if each M_i refines CS_i, then their composition I refines CS). Thus, we leverage design modularity in the implementation to create modular specifications and enable modular verification—this improves the scalability of verification.

Effectively, our component specifications CS_i enable decomposing the full FSIV problem into one SEC and multiple RC verification tasks (including IC). In practical use cases, this methodology also supports the ability to replace or enhance a single implementation module M_i and prove the implementation correctness by only redoing the RC (including IC) between M_i and CS_i. In this way, we can guarantee correctness of the implementation *incrementally* and avoid repeating the full system verification.

Although compositional verification and RC have been successfully used in other prior efforts in hardware verification [18, 25, 36], there are important differences in our work (discussed in detail in Section 6).

1.4 Contributions

This article makes the following contributions:

- We propose a new methodology for protocol implementation verification (outlined in Figure 2) that uses modular component specifications *CS* as a bridge between *S* and *I* to decompose FSIV into a set of simpler verification tasks (SEC and RC). The methodology bridges the gap between *S* and *I*, which significantly improves scalability of the FSIV problem.
- We show how ILA and related verification techniques [22, 58, 59] can be leveraged for modeling component specifications and the resulting SEC and RC verification tasks. In particular, to verify the interactions between components in the implementation, we use the interface specification (via *valid/ready* handshake signals) in the ILA models (Section 2.1) and perform additional interface checks as part of performing refinement checks (Section 3). This provides the basis for compositional RC [59] (i.e., if each M_i refines CS_i, then *I* refines *CS*). Thus, together our SEC and RC verification tasks ensure the correctness of the implementation with respect to the specification (Section 3.3).
- We implement our ILA-based protocol verification methodology and demonstrate its effectiveness in three different case studies (Section 5). These protocols are part of real designs: the AMBA AXI on-chip communication protocol [44], an off-chip communication protocol used in BaseJump STL [52], and a cache coherence protocol used in OpenPiton [3]. For these case studies, our method not only found bugs in the protocol implementations that were confirmed by designers but also performed the complete FSIV task in reasonable time and effort.

This article is organized as follows. We start by providing some background on ILAs and ILA composition (Section 2) and then describe the ILA-based protocol verification methodology (Section 3) along with an illustrative example (Section 4). We describe three protocol case studies (Section 5) and report experimental results. This is followed by a discussion of the related work (Section 6) and our conclusion (Section 7).

2 BACKGROUND

We include the relevant background on ILA specifications and compositional verification using ILAs because it is critical for understanding the proposed FSIV methodology.

2.1 ILA Specifications

ILA can be viewed as an extension of ISA, which serves as a formal specification for processors. An ISA specifies the architectural state for a processor—that is, the state that persists between instructions. In addition, an ISA specifies the decode conditions and state updates for each of its instructions. There have been many successful efforts in processor verification that check an implementation instruction-by-instruction against a formal ISA specification [6, 33, 34].

The ILA specification [23] was first introduced to extend the familiar 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 methodology were further extended for specification and verification of general hardware modules [58]. Recently, the ILA composition methodology for general hardware modules was proposed in prior work [59], where the ILA model was augmented with an interface specification.

In this work, we leverage this notion of treating commands at the interface of a general hardware module as instructions to model component specifications for protocol implementations. Here, the commands at the interfaces are the interactions between the components during the execution of the protocol.

As introduced in prior work [59], an ILA model of a component is represented as a six-element tuple, which is defined formally as follows:

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

S is a vector of state variables (state space denoted as $\mathbb{S}$) ,

W is a vector of inputs variables (including *valid/ready*, input space denoted as $\mathbb{W}$) ,

O is a vector of output variables (including *valid/ready*),

$\mathbb{O}$ (output space of O) $\subseteq \mathbb{S}$,

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 update functions

Note that J is the set of instructions, associated with the sets D and N. Each element in D specifies a condition for triggering an instruction (i.e., the interface command), and each element in N describes the state update performed for each instruction $j \in J$.

This ILA definition focuses not only on module specification but also considers the interface specification for interacting and communicating with other modules. Outputs are defined to support the composition of ILA models. This allows connecting inputs and outputs of different ILA models to enable their interaction. We note that most modules in protocol implementations use simple handshake signals for correct communication. In general terms, an interface between two ILA models is specified in terms of two handshake signals—*valid* and *ready*—in the outputs/inputs of ILA models, as shown in Figure 3(i).

2.2 Interface Specification Using Handshake Signals

Protocol implementations often use simple handshake signals in their interface (e.g., a *valid/ready* handshake among modules) so that different modules can interface with each other correctly. Conceptually (although implementations vary), a *valid* signal is set to high when a module is prepared to send the payload to another module, and a *ready* signal is set to high when a module is prepared to receive the payload from another module. Only when *valid* and *ready* are both set to high in the respective modules can the payload be transferred from one module to another.

An example ILA model including four handshake signals is shown in Figure 3(i), where $valid\_o$ and $ready\_o$ are outputs of this model, say P, and $valid\_i$ and $ready\_i$ are inputs from another ILA model, say Q. P and Q are interacting with each other based on these handshake signals.

In Figure 3(ii), we show an example interface specification for the handshake signals in the ILA model P—that is, how $valid\_o$ and $ready\_o$ (the output variables labeled in each state) are updated by P, depending on its own current state and its inputs $valid\_i$ and $ready\_i$. The inputs change will trigger different instructions in different states. We refer to these as *instructions with handshake operations* in the ILA models.

There are two kinds of handshake operations, say "send" and "receive." For the instructions with "receive" operation, the decode function includes the condition that $valid\_i \wedge ready\_o == 1$, and for the instructions with "send" operation, the decode function includes the condition that $valid\_o \wedge ready\_i == 1$. The decode function can also include other information, such as input or state variables, to trigger different state update functions for other state variables. Whenever some interaction occurs between two ILA models, there should always be an instruction with "send" operation decoded in the sender ILA model *and* an instruction with "receive" operation decoded in the receiver ILA model.

For example, Figure 3(iii) shows the ILA specification (including the interface) for a component P (the same as for component Q). There are 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

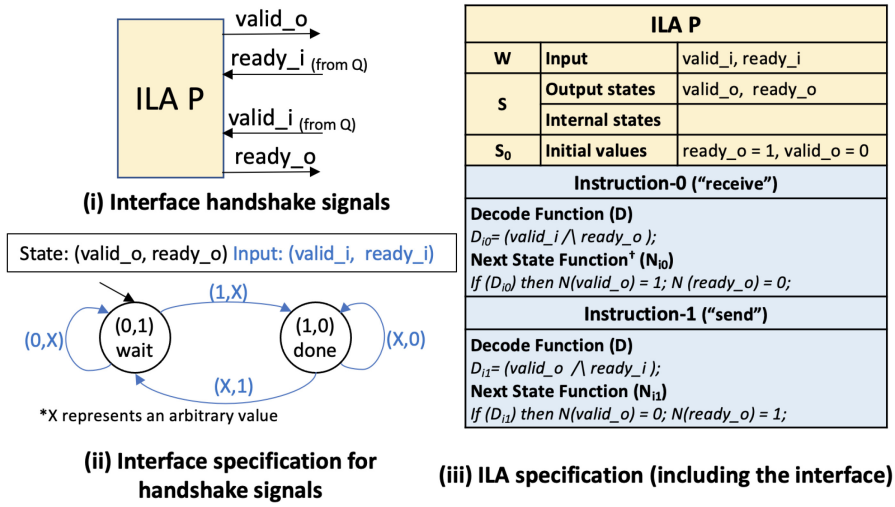

Fig. 3. ILA interface specification: handshake signals and interface instructions.

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*—that is, 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.

Importantly, by considering the handshake signals as inputs/outputs at the interface of a component, the overall problem of specifying communication between protocol 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.

2.3 ILA Composition

In the setting of this work, an ILA model can be viewed as a Moore FSM, where $\mathbb{O} \subseteq \mathbb{S}$. 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 $C = A1 \parallel A2 = \langle S_C, W_C, O_C, S_{0C}, \delta_C\rangle$, defined as follows:

$$
\begin{aligned}
S_C &= S1 \times S2 \\
W_C &= W1 \cup W2 \setminus ((W1 \cap O2) \cup (W2 \cap O1)) \\
O_C &= O1 \cup O2 \setminus ((W1 \cap O2) \cup (W2 \cap O1)) \\
S_{0C} &= S1_0 \times S2_0 \\
\delta_C &: (S_C \times W_C) \rightarrow S_C \text{ is the state transition function.} \\
&\delta_C((S1, W1), (S2, W2)) = (S1', S2'), \text{where} \\
&S1' = \begin{cases} N1_j(S1, W1) & \text{if } \exists j. D1_j(S1, W1) = 1 \\ S1 & \text{otherwise} \end{cases} \\
&S2' = \begin{cases} N2_k(S2, W2) & \text{if } \exists k. D2_k(S2, W2) = 1 \\ S2 & \text{otherwise.} \end{cases}
\end{aligned}
$$

Each state of the composition C is a pair comprising 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 execution of an instruction in one or both components.

This definition also generalizes in a straightforward manner to a composition of n ILA models. We use ILA composition to construct an FSM for $CS = CS_0 \parallel CS_1 \parallel \ldots \parallel CS_{n-1}$ in Figure 2.

2.4 ILA-Based Refinement Verification

As discussed in Section 1.3, the key requirements for the component specifications CS_i are (i) modeling of interface signals and specifying the interactions between components, and (ii) enabling RC for each component implementation M_i. Both these requirements are well met by the ILA model and supported through the verification tools in the open source ILAng platform [22]. In particular, we will use ILA-based verification to prove that each module M_i is a refinement of the specification CS_i, denoted as $M_i \triangleleft CS_i$.

For performing a refinement check, the ILA tools automatically generate a set of verification properties—one per instruction—by using a given *refinement map*. Essentially, the refinement map specifies *what* to check and *where* to check equivalence, since the two 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*) 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 correctness of a pipelined hardware implementation against a sequential ISA/ILA [6, 33, 34]. The per-instruction properties that are generated by ILA-based refinement verification can be checked using standard open source [35] or commercial model checking tools [7].

Since the specification also includes the handshake signals (*valid/ready*), this RC also checks the handshake signals at instruction completion time. In addition, **Pre-Completion Checks (PCCs)** are generated to ensure that the interface signals have correct values even before the instruction completion points. The following result from prior work [59] states that if implementation *I1* is a refinement of specification S1 and implementation *I2* is a refinement of specification *S2*, then the composition of *I1* and *I2* is a refinement of the composition of *S1* and *S2*.

Lemma (Composition Lemma [59]). *If all refinement checks on M_i and CS_i pass, then the composition $M = M_0 \parallel M_1 \parallel M_2 \ldots \parallel M_{n-1}$ is a refinement of the composition $CS = CS_0 \parallel CS_1 \parallel CS_2 \ldots \parallel CS_{n-1}$.*

It is worth emphasizing that other existing methodologies or tools do not provide automated generation of a complete set of properties for RC for hardware modules other than processors. Thus, the ILA component specifications are quite valuable for this purpose and enable leveraging standard model checkers.

3 ILA-BASED METHODOLOGY FOR FSIV

The two major verification tasks in the ILA-based FSIV methodology as outlined in Figure 2 are as follows:

- SEC between the composition of ILA models $CS = \|_i\ CS_i$ and the formal specification S.
- RC between each ILA model CS_i and its corresponding implementation module M_i, where IC is included in this task to ensure correctness of component interactions.

The first task is a standard verification problem, and the second task is based on using the ILA models and the interface specifications described in the previous section (Section 2). We describe these tasks in detail in this section.

3.1 SEC Task

The SEC task guarantees that the outputs of two models being checked are equivalent at each step under the same inputs. There are many techniques for doing SEC [37]. Conventional algorithms for solving the sequential equivalence problem first build the so-called product machine, which is the parallel composition of the two models being verified. Then a model checker can check that the corresponding outputs of the two models are identical in every state of the product machine reachable from the initial states. Many new approaches have been proposed to tackling the inherent state explosion problem, such as induction [27] and structural approaches [53].

As mentioned earlier, the ILA composition CS has abstracted away many of the performance-enhancing implementation details, which helps reduce the complexity of the SEC between CS and S. Note that the CS_i are created such that transitions (i.e., steps) in CS correspond to transitions (steps) in S, and this structural similarity can be utilized by the model checker to further improve the SEC solution.

In our implementation, we use the ILAng platform [22] to translate the CS_i ILA models and generate the ILA composition for use with a commercial model checking tool (JasperGold) [7] to perform SEC.

3.2 RC Tasks

The second main task in our proposed methodology is to perform refinement checks to guarantee that the protocol implementation M is a refinement of the ILA composition CS. Following the ILA compositional verification methodology, there are two sub-tasks needed for this. The first is to verify that each implementation component M_i refines its ILA specification CS_i, denoted as $M_i \triangleleft CS_i$, and the other is to do the additional PCCs on the interface against the interface specification.

3.2.1 Modular RC. Recall that when $M_i \triangleleft CS_i$, the RTL component M_i and its ILA specification CS_i are shown to have equivalent outputs at corresponding points specified in a given refinement map (Section 2), which is provided by the user. In our application, we use a refinement map that provides the following information:

- *Mapping between outputs of interest in CS_i and M_i*: The set of outputs includes all system-level outputs, outputs from one component that are connected as inputs to another, and the handshake signals that are used in interface specifications.
- *Correspondence points that indicate the end of each instruction*: This is because that the number of steps to complete an instruction can be different in an ILA model CS_i (which is not cycle-accurate) and the RTL model M_i (which is cycle-accurate).

Note that the ILA models include interface instructions that specify the updates to the handshake signals according to the interface specifications. We then use the standard ILA-based refinement verification methodology [58] to perform the component refinement checks. This involves using the given refinement map to automatically generate correctness properties, which are then checked by a standard model checker.

Thus, if a refinement check $M_i \triangleleft CS_i$ passes, then the RTL component M_i and its ILA specification CS_i are shown to have equivalent outputs (including handshake signals) at the end of each instruction as specified in the refinement map.

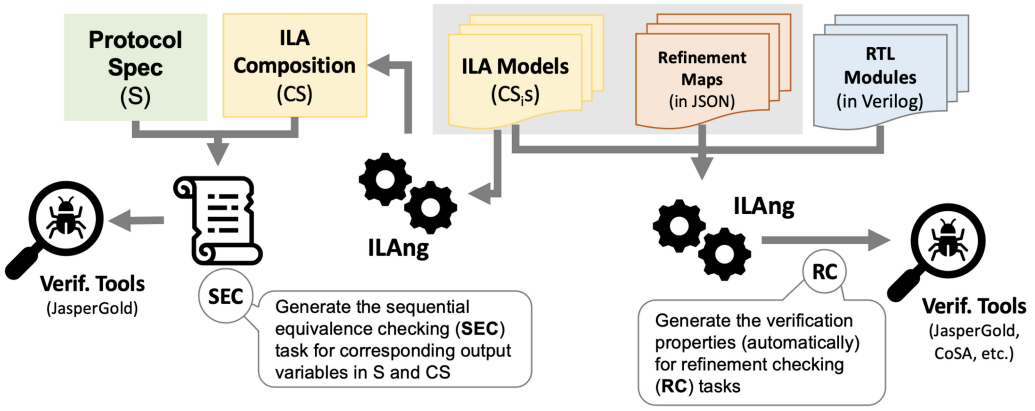

Fig. 4. Verification flow of FSIV using ILA. The ILA models and refinement maps in the gray shaded box need to be written manually.

3.2.2 Additional Interface Checks. Note that checking $M_i \triangleleft CS_i$ focuses on checking equivalence of specified outputs at the end of each instruction (as specified in the refinement map). However, checking equivalence only 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 timesteps (i.e., even before instruction completion). Therefore, we perform additional checks to ensure that the interface signals have correct values even *before* the instruction completion points.

These are included in the following two additional PCCs:

- *PCC1*: For each M_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 M_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 ensure that the payload is correctly transferred as per the ILA interface specification. As we can see, these two checks are performed per instruction, and we include them in RC itself. Note that all properties for individual RC between CS_i and M_i and pre-completion checking for each module are automatically generated, providing a systematic verification methodology.

3.3 Summary: FSIV Tasks and Correctness

The key idea in our ILA-based methodology is to leverage the modularity in the protocol implementation I to provide a high-level intermediate specification CS_i for each component M_i in I. These intermediate specifications CS_i play the role of a bridge between the high-level formal specification S and the low-level implementation I.

The complete verification flow is shown in Figure 4. After creating CS_i in the form of ILA models, we automatically generate properties for the RC tasks, based on user-provided refinement maps. These properties are verified by a standard model checker. For the SEC task, we construct the ILA composition (using utilities available in the ILAng platform [22] and formulate the SEC problem, which is checked by a standard model checker. Note that our use of CS_i in the two FSIV tasks provides the following benefits, in comparison to a monolithic refinement check $I \triangleleft S$:

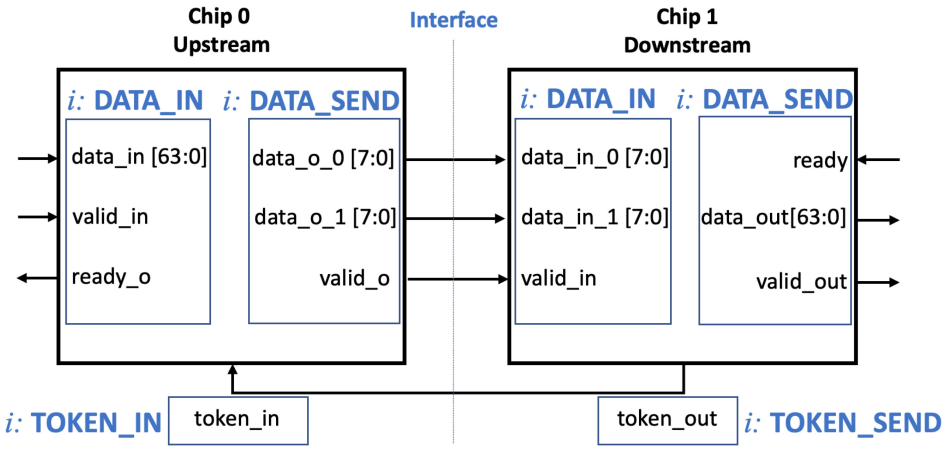

Fig. 5. Instructions and interface signals for the off-chip communication protocol. *i: DATA_IN, i: DATA_SEND, i: TOKEN_IN*, and *i: TOKEN_SEND* are the instructions. *Data_in (64), valid_in (1)* and so forth are the interface signals with the bit width inside the parentheses.

- *Task 1*: The high-level CS_i make it easier to check the sequential equivalence of CS and S.
- *Task 2*: Each refinement check $M_i \triangleleft CS_i$ is simpler than checking the monolithic refinement check $I \triangleleft S$.

Section 5 provides empirical evidence of these benefits for practical case studies. The composition lemma shown in Section 2.4 provides a connection between the two tasks.

FSIV Correctness. If the sequential equivalence check passes and all refinement checks pass, then the implementation I is correct with respect to the specification S. To see this, note that when all refinement checks pass, then each M_i refines CS_i, and the composition I of all implementation components M_i is a refinement of CS (according to the composition lemma (Section 2.4)). When the sequential equivalence check passes, then CS is equivalent to S. Thus, I is correct with respect to S.

4 ILLUSTRATIVE EXAMPLE: OFF-CHIP PROTOCOL

The off-chip communication protocol in BaseJump STL [52] has two components: an upstream module (named *Upstream*) and a downstream module (named *Downstream*) as shown in Figure 5. It uses tokens to ensure correct data communication between the two modules. The target of our verification is checking that the implementation is correct with respect to this token-based communication protocol. Figure 5 also shows how we model the instructions for each module based on the interface signals. The 64-bit data is transferred from *Upstream* to *Downstream*, so we have the *DATA_IN* and *DATA_SEND* instruction for each module to indicate when the data comes in and goes out, respectively. Further, *TOKEN* can be sent from *Downstream* back to the *Upstream* so that *Upstream* can determine whether to receive more data from the environment. As a result, we have the instruction *TOKEN_SEND* in *Downstream* and the instruction *TOKEN_IN* in *Upstream* to model this token feature. According to the protocol specification document, the data transfer between two different chips (*Upstream* and *Downstream*) is limited to 8 bits per channel. Accordingly, the 64-bit data_in *Upstream* is transferred in four steps to *Downstream* via the 8-bit channels of *data_o_0* and *data_o_1* and then sent out as 64-bit data_out by *Downstream*. The main job of the protocol in this design is to guarantee that all data received in *Upstream* should be eventually sent out by *Downstream*, which is the data integrity property of the communication.

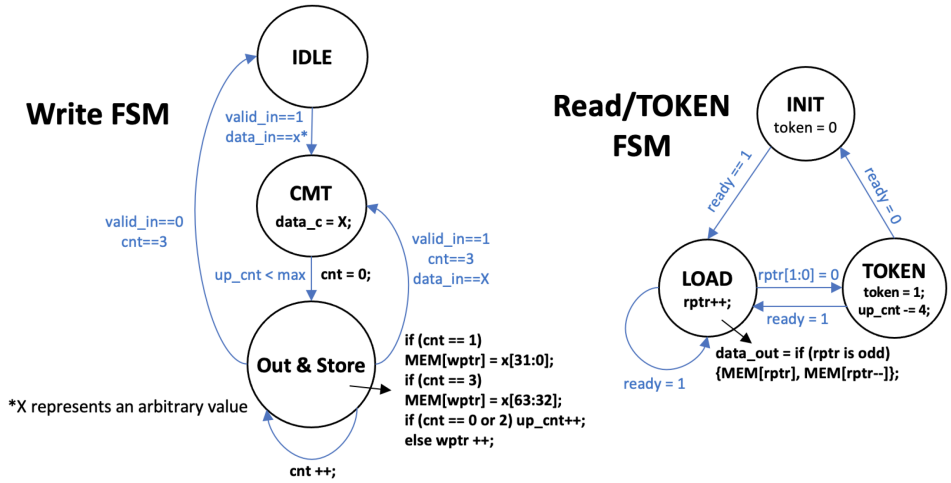

Fig. 6. Specification, S, of the off-chip protocol.

In this section, we illustrate the steps in our methodology, with examples of ILA models and ILA-based verification using refinement maps.

High-Level Specification S**.** To verify this token-based protocol, we first wrote a high-level formal specification S. Figure 6 shows a simplified version of this specification comprising two FSMs: one for writing data, and the other for reading data and token control. In Figure 6, the blue labels on edges are guards, and output assignments are shown in bold in each state or along an edge. The *Write* FSM ensures that the data received in the *CMT* state will be stored in two pieces in a buffer after four steps (*cnt* is used to record each step), and the *Read/TOKEN* FSM makes sure that the token signal is generated correctly after a certain number of data are loaded based on the read logic. *Write* FSM increments the *up_cnt*, whereas *Read* FSM decrements *up_cnt*. Composing these two FSMs ensures that *up_cnt* is calculated correctly, and no more data can be stored when *up_cnt* reaches the maximum value, indicating that the buffer is full. One possible failure scenario is that if *up_cnt* is not maximum when the buffer is full, some data will be confirmed to be received but not stored in the buffer, leading to data lost during the communication. A set of high-level properties H is verified for S. For example, a liveness property expressed in temporal logic ($h_0 \in H$) is used to check that there is no data loss during the data transfer between chips:

$$\mathbf{AG}((state == CMT) \Rightarrow \mathbf{AF}(data\_out == data\_c)).$$

When *Write* FSM is in state *CMT*, it means that the data is confirmed to be received and it should eventually be transferred to *data_out*. If not, then the data is lost under the protocol.

Implementation I**.** The RTL design is large (7,478 LoC in Verilog), comprising the upstream and downstream modules described previously, with many additional performance-enhancing features such as integrating buffers in both upstream and downstream modules. Thus, a single operation in the component specifications for both modules will take many cycles in the RTL to complete. For comparison with traditional methods, our experiments (described later in Section 5) also report the results for verifying some high-level properties on S and on I. However, as discussed earlier, even when high-level properties H are verified on the implementation I, that may not be enough to provide assurance that the implementation correctly implements the token-based specification. For example, a regular handshake communication protocol without the token feature (e.g., the example design in Figure 3) can guarantee the same high-level property (e.g., no data loss liveness

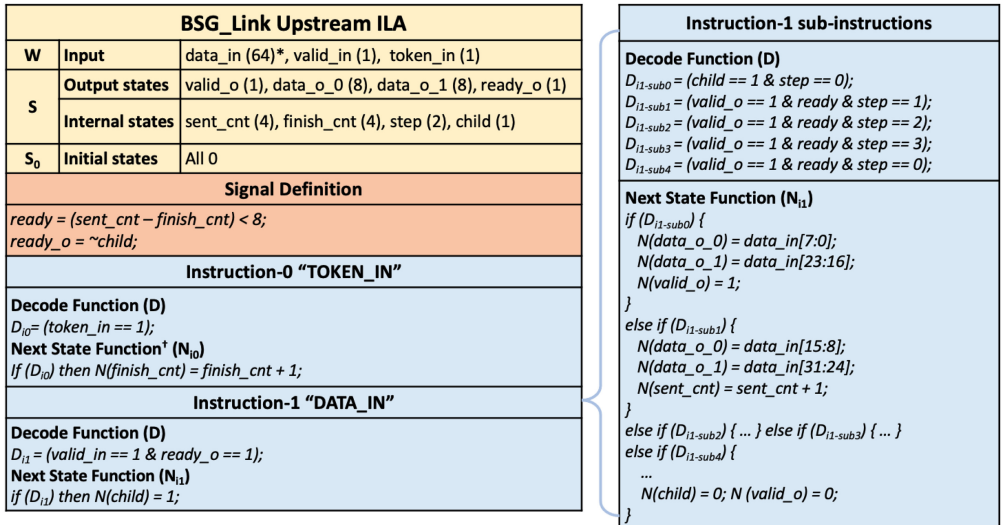

Fig. 7. ILA model for *Upstream* (partial).

property), but the implementation does not follow the token-based protocol specification. This gap is addressed in FSIV, which guarantees *completeness* of the implementation verification against a given specification.

Component Specifications *CS*. We create the component specifications for an upstream module and a downstream module in the implementation. One can see from Figure 6 that it is not easy to derive these component specifications from *S*, since the specification FSMs are for writing and reading, respectively, and not for these individual components. Note also that the interface information between modules is not easy to obtain from the specification *S*.

We created ILA models for the upstream and downstream components, based on commands at the interface of the modules as shown in Figure 5. Although our methodology adds this step of creating ILA component specifications, in our view the resulting benefits to FSIV justify this extra step. The (partial) ILA model for the upstream component is shown in Figure 7 (defined in Section 2.1). The *Upstream* ILA model has two instructions and five sub-instructions.[2] Each (sub-)instruction is defined by its own decode and state update function. We can see that the two instructions receive information from other modules, and the sub-instructions generate the outputs and send information to other modules.

The interface specification of *Upstream* is also included in Figure 7. For the receiving part, the instructions are based on the *valid_in* and *ready_o* signals and update function for *ready_o* is shown in the specification. For the sending part, however, we only have *valid_o* specified in the interface specification, whereas the ready signals are implicitly specified as being always high (leading to a simplified handshake). Similarly, the *Downstream* ILA model is built to specify the downstream module and its interface.

FSIV Task for SEC. We consider a given RTL design with one upstream and two downstream modules. We construct *CS* using the composition of an ILA *Upstream* model and two ILA

[2]Sub-instruction: a step in an instruction execution that results in a visible state update [23].

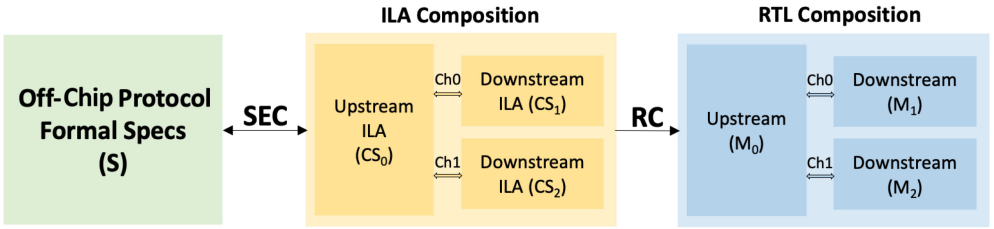

Fig. 8. FSIV tasks for the off-chip protocol.

Upstream ILA	Upstream RTL
State Map	
valid_o	io_valid_out
data_o_0	io_data_r_o[7:0]
data_o_1	io_data_r_o[15:8]
ready_o	piso_ready_o
sent_cnt	pos_r_cnt_r+neg_r_cnt_r
finish_cnt	w_cnt_b_ch0+w_cnt_b_ch1
step	#cnt#
child	fifo_valid_o
Interface Map	
data_in	core_data_in
valid_in	core_valid_in
token_in	core_token
Instruction Map	

	start condition	commit condition
TOKEN_IN	decode	1 cycle
...	...	...

Auto-generated property for "TOKEN_IN" instruction

```
(#state mapping#)
(ila.valid_o == &rtl.io_valid_out) &&
(ila.data_o_0 == rtl.io_data_r_o[7:0]) && (...)&&
(ila.ready == rtl.piso_ready_o) &&
(#interface mapping#)
(ila.data_in == rtl.core_data_in) &&
(ila.valid_in == rtl.core_valid_in) &&
(ila.token_in == rtl.core_token) &&
(#decode#)
(ila.token_in == 1)  &&
(ila' = ila.next_state()) →
(#commit condition & check#)
X [ (ila'.finish_cnt == rtl.w_cnt_b_ch0 +
rtl.w_cnt_b_ch1)  && (...)]
```

Fig. 9. Refinement map for *Upstream* (partial) and an auto-generated property.

Downstream models, each of which also includes the interface specifications. Then, we perform SEC between S and CS, as shown on the left in Figure 8. The SEC verifies that given the same inputs, the outputs of CS are equivalent to the outputs of the formal specification S in every transition. This guarantees completeness and correctness of CS. We perform SEC using a commercial model checking tool (JasperGold).

FSIV Tasks for RC. We next check that each component M_i in the implementation is a correct refinement of its ILA component specification CS_i, as shown on the right in Figure 8. This guarantees that the composition of the implementation modules is a correct refinement of the composition of the ILA models. The properties to be checked for each modular refinement check are generated automatically by the existing ILA tools using a refinement map [22]. The (partial) refinement map for this example is shown in Figure 9. It it composed of three main parts: a state map, an interface map, and an instruction map. The state map provides the correspondence between the ILA architectural state variables and the RTL registers. The interface map relates the inputs so that when a command is presented to the ILA, the corresponding inputs will also be presented to the RTL. The instruction map specifies the start and finish condition of each instruction in the RTL. For example, for the `TOKEN_IN` instruction, the start condition is its decode function, and the finish condition (when to check) is one clock cycle after the instruction starts.

The right side in Figure 9 shows an example property that is generated automatically for the `TOKEN_IN` instruction, by using the refinement map shown on the left side of the figure. It checks the

Table 1. Results for SEC Verification Experiments

Protocol Design	Specification Statistics		ILA Composition Statistics		SEC Verification	
	Size (LoC)	No. of State Bits	Size (LoC)	No. of State Bits	Time (s)	Memory (MB)
AXI Protocol	484	343	1,628	360	0.9	7.35
Off-Chip Link	183	91	589	171	106	4.36
Cache Coherence	871	646	1,529	634	490.2	6.12

following: if the ILA model and the RTL implementation execute the instruction with equivalent corresponding starting states, and the ILA applies the next state function and the RTL finishes the instruction based on the commit condition, then the ILA's architectural states (denoted as ila') should be equivalent to the corresponding RTL states. (The symbol $\rightarrow$ denotes logic implication and the temporal logic operator **X** denotes the next cycle in the RTL.) To make it clearer, the property is shown divided into several parts corresponding to different parts in the refinement map (with comments for the different parts shown in shaded background). Similar properties are automatically generated for each instruction and are formally verified using a model checker.

We include the handshake signals and interface specifications in the ILA models and the corresponding refinement maps, and perform two PCCs (as described in Section 3.2.2). For example, one pre-completion checking property that is generated for the valid signal is as follows:

$$assert\ \{valid == 0 \mid => \text{nexttime \$stable } (valid) \text{ until "commit condition of DATA\_IN"}\}.$$

In effect, the refinement checks (including interface checks) guarantee that the composition of RTL modules for the off-chip protocol is a refinement of the composition of the corresponding ILA models. Together with SEC, this completes all FSIV tasks.

5 CASE STUDIES

In this section, we demonstrate the applicability and effectiveness of our proposed ILA methodology for FSIV through three case studies: the on-chip AXI communication protocol [44], an off-chip communication protocol used in BaseJump STL [52] (discussed in Section 4), and a cache-coherence protocol used in OpenPiton [3].[3] We successfully verified all three protocol implementations and detected some bugs that were confirmed by the designers. The open source ILAng platform [23] was used for ILA tools, and JasperGold [7] 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 SEC are shown in Table 1, and for RC (which includes IC) in Table 2. These tables summarize the statistics of the protocol specifications, implementations (size ranging from 800 to 11,000 LoC), ILA models (implemented in C++, where the size is a rough measure for the human effort involved), and refinement maps (written in JSON format, where again the size reflects human effort). We report the runtime and memory required for verification.

5.1 AXI Protocol

The widely used on-chip AXI communication protocol is a burst-based data transfer protocol, where the communication channels use a *valid/ready* handshake mechanism. It consists of five channels: two for reading and three for writing. The reading and writing channels are separated. Data can be transferred from a leader module to a follower module only when ready and valid

[3]Source code for all models and verification properties is available at https://github.com/anonymized-hierarchical-verification.

Table 2. RC Verification Experiments

Protocol	Design Statistics			ILA Model Statistics			Verification			
	Protocol Modules	RTL Size (LoC)	No. of State Bits	No. of Instr.	ILA Size (LoC)	No. of State Bits	Ref-Map Size (LoC)	Bug-Found Time (s)	Proof Time (s)	Memory Usage (MB)
AXI Protocol	Follower	828	372	9	167	159	77	0.01	0.11	7.8
	Leader	871	403	11	184	289	109	0.01	0.23	9.7
Off-Chip Link	Upstream	2,982	713	7	144	146	284	0.3	756.6	253.5
	Downstream	5,453	2,474	4	101	98⋆	196	–	38.2	89.1
CCP	L2 Cache	10,924	2,844⋆	8	596	340⋆	272	0.7	1214	2,270

⋆Not including memory block.

signals are both asserted in one channel, as required by the handshake mechanism. (We are following the inclusive language guidelines being recommended by standards bodies like IETF [28] in replacing "master-slave" in technical references with "'leader-follower.")

In past work, simulation-based testing [19] was explored for similar protocols. For formal verification, HLIV for this protocol has been done using model checking [32], where some properties were verified by the CoSA model checker [35]. However, the properties were written manually based on the specification, and as with HLIV in general, there is no guarantee of their completeness for checking the implementation against the protocol specification.

For our ILA methodology, we built four ILA models: one each for reading and writing channels in each of the leader and follower modules. All ILA models include the interface specification on the ready/valid handshake signals in the protocol. We then composed the ILA models and checked the sequential equivalence (SEC) against the AXI specification—this finished in 0.9 seconds.

Next, we used the ILA tool to perform RC between each ILA component model and its corresponding RTL module. We found two bugs in the follower and one bug in the leader components through RC. The AXI protocol requires that the interface data be asserted 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 reading 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 0.01 second. 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 were verified in 0.1 and 1 second, respectively.

5.2 BaseJump Off-Chip Protocol

The off-chip communication protocol used in BaseJump [52] is not a simple *valid/ready* handshake protocol. Due to the constraints of the I/O interface, data has to be fragmented and there is no ready signal for the data transfer (which is modeled in the interface specification as ready being true all the time). Instead, a token-based feature is used in the protocol to evaluate the ready condition and ensure there is no data loss during the communication. The protocol design includes an upstream and a downstream module as described in detail earlier (Section 4).

SEC and RC Verification Results. The correctness of the composed ILA models was checked using SEC, which finished in less than 2 minutes. During RC, one interface bug was identified in the upstream module. The implementation incorrectly transferred the invalid data, which is not allowed in the specification. The bug was found within 0.3 second. Following a discussion with the designer, this bug was fixed by adding appropriate environment constraints that were being assumed by the designer. After the bug fix, RC for all modules was completed successfully in 15 minutes.

Table 3. Results for HLSV and HLIV on the Off-Chip Protocol

Verification Task	Design Size	No. of Properties	Proof Time	Proof Memory
HLSV on Spec S	183 LoC/ 91 state bits	2	2,593.5 s	6.36 MB
			425.2 s⋆	18.43 MB
HLIV on Impl I	7,478 LoC/ 3,187 state bits	2	Bounded proof for 399 steps in 24 h	682 MB

⋆We apply data abstraction in HLSV to speed up the verification, and the number of state bits in this case will be less than 91.

Comparison with HLSV and HLIV. We used this case study to perform a direct comparison with the traditional verification methods for HLSV and HLIV. We checked two high-level properties on the specification S and on the implementation I, respectively. The properties are expressed as **System Verilog Assertions (SVA)** and verified by a commercial model checker (JasperGold). To express these properties, we added monitor signals *req* and *acq*. The signal *req* is set to high when there is some data x coming to *Upstream*, whereas the signal *ack* is set to high when the same data x is coming out from *Downstream* (x is modeled as a symbolic value to capture all possible values in the verification). The first property is a safety property: the number of *req* is at least the number of *ack*s: $assert\{\#req >= \#ack\}$. The other property is a liveness property that every *req* leads to an *ack*: $assert\{req \mid => \#\#[0:\$]\ ack\}$. These two properties are checked on S and on I, and the results are shown in Table 3.

Note that HLSV is useful to get some confidence in the protocol specification model S. HLSV completed within about 40 minutes. We also experimented with abstracting the data from 64 bit to 8 bit (using standard data abstraction), which speeds up the verification to less than 8 minutes. However, checking the same properties directly on the implementation I does not scale well, due to the large size and complexity of the full RTL design. In our experiment, we could only finish a bounded proof for 399 steps in 24 hours. Further, these two properties checked on the implementation are *not* a complete set of properties based on the specification S. In contrast, our ILA-based methodology for FSIV proves the correctness of I with respect to S in less than 20 minutes (the total for SEC and RC tasks described earlier).

5.3 OpenPiton Cache Coherence Protocol

The cache coherence protocol used in OpenPiton [3], an open source many-core processor, is a directory-based MESI coherence protocol comprising three components: a private cache (L1), a directory (L2), and memory (MEM).

We first performed HLSV on a formal model S of the protocol. Two high-level properties (mutual exclusion and a data property) were verified for S. For n number of cores in S, with $n = (2, 4, 16)$, S was verified in 11.7 seconds, 50.6 seconds, and 2 hours, respectively. Although parameterized model checking for HLSV for cache coherence protocols is well studied (Section 6), we did not use it in our experiments due to the commercial model checker not directly supporting it.

Then, we created an ILA model for each component in the protocol, and performed SEC between the composition of ILA models and the formal model S—this took 9 minutes. Next, we performed RC between the ILA models and the corresponding implementation modules. Both ILA models and RTL modules communicate through a *valid/ready* handshake interface, where the command at the interface can be executed when valid and ready are both set to high in the modules sending and receiving, respectively. In Table 2, we show the results for RC of only the L2 cache, since it is the most complicated component and had a bug. All other implementation modules were verified to be correct.

The L2 cache implementation has dual parallel pipelines modeled as two independent ILA ports: `PIPE1-port` and `PIPE2-port`. We found one bug while verifying `PIPE1-port`. There was a typo in the informal document saying that pipeline register `msg_flag_2` is used, whereas `msg_flag_3` is needed. In our verification, a counterexample trace exposing this bug was found in 0.7 second. After confirming the bug with the design group member and fixing it by correcting the variable name, the L2 cache implementation was verified in 20 minutes.

6 RELATED WORK

Our work is related to many past efforts that have addressed different aspects of protocol verification and to various techniques that have been applied in general hardware verification.

6.1 HLSV for Protocol Verification

Property-Based Verification. As mentioned in Section 1.1, there is a rich body of work in property verification of protocols: for cache-coherence protocols [15, 30, 43, 51, 56], for network protocols [1, 4, 14, 16], and for communication protocols [9, 47, 50]. Tools such as Murphi [13] and SPIN [17], and specification languages such as TLA+ [26] and LNT [21], have been successfully used to specify and check correctness of protocol specifications. However, these approaches are largely based on *monolithic* verification, without much use of compositional techniques.

Parameterized and Compositional Techniques. Some protocols deal with a large number of components, which lead to scalability concerns in verification. For instance, a cache coherence protocol can have a parametric number of n elements as part of the protocol (n can be very large). The CMP (Chou-Mannava-Park) method [51] applies parameterized model checking [11, 38] to address this. In this method, verification of the cache coherence protocol for a parametric number of components (cores) is done using abstraction with a fixed and small number of components. Another method to improve the scalability is to use flow-based specifications [43, 51]. The flow-based description can generate quite intuitive and powerful invariants of the protocol, speeding up the proof process. However, these works focus on the correctness of a high-level protocol specification only; the refinement relation between the specification and an RTL implementation is difficult to determine and is not verified. Our work fills this gap using the proposed ILA-based methodology.

High-Level Models for Hardware Modules. In addition to modeling protocols, there is prior work on high-level hardware specifications. For instance, SystemC, transaction-level modeling (TLM), and Bluespec [42, 45, 49] are used to specify the requirements of the RTL design. These high-level models help raise the level of abstraction and hence improve the scalability in system integration or software/hardware co-design. However, the verification in these works is generally simulation based, with formal verification used mostly at the component level but not at the system level. There is also some work in high-level formal modeling of interface signals [12], where the RTL module interface signals are characterized into four sorts: *to-sync*, *to-port*, *from-sync*, and *from-port*. This enables the interfaces among multiple modules to be formally analyzed to avoid bugs such as combinational loops. However, this work focuses mainly on the well connectedness of a system but not on the functional correctness of each design module. In contrast, our work leverages the ILA as a modular component specification that provides for a *complete* formal verification for the component implementation. Following this verification, the composition of our ILA models serve as a sound abstraction of the implementation and is then checked against the protocol specification using SEC.

6.2 HLIV for Protocol Verification

Property-Based Verification and Assertion Generation. In RTL design verification, property-based verification using SVA [8] or Property Specification Language (PSL) [24] is commonly used.

However, it is still challenging to *automatically* generate these properties from high-level models. Some prior work automates the generation of assertions based on simulation traces [48, 54] or from high-level properties for the high-level models [5, 20, 46]. However, as we highlight, it is difficult to write a complete set of properties or to evaluate the completeness of a set of properties for RC of RTL designs. In our work, this challenge is addressed via the ILA methodology that provides a full functional specification with automated generation of a complete set of properties for RC based on this specification. Furthermore, the properties are generated as standard SVA, which are supported by commercial model checking tools (e.g., JasperGold) to verify the RTL implementations.

Property Specifications with Intermediate Representations. Intermediate specifications in the form of a low-level main FSM have been used in previous work [39–41]. This low-level main FSM is handwritten based on the implementation to capture the main control states. Manually written interval properties based on this FSM are then checked on the implementation. It is difficult to determine the completeness of this set of properties, especially since this low-level main FSM is derived from the low-level implementation without verifying this FSM against the high-level specification.

6.3 FSIV in Hardware Verification

Compositional Verification. Compositional reasoning has also been applied to hardware implementation verification [18, 25, 36]. However, there are two important differences with our work that we would like to emphasize. First, the earlier efforts were not applied for protocol verification but were used to check the correctness of processor microarchitectural features (e.g., out-of-order execution, speculative branching) by checking a series of successive refinements of the given specification model. In our methodology, we explicitly perform SEC between the specification model S and the composition CS of the component specifications. This check was not performed in prior efforts. Second, these earlier efforts required more complicated refinement relations that specify temporal properties that relate components to the reference model. In comparison, our refinement mapping is much simpler—it specifies the corresponding state variables in the two models and when they should be checked for equivalence.

RC for Protocols. Refinement proofs between high-level protocol specifications and low-level implementations have been explored using the TLA+ [29] specification language. Extensive research on modular refinement proofs for hardware systems has been conducted, such as the study of Liu et al. [31] that focuses on verifying processor arrays using Mocha [2] model checking tools. To verify a hardware design using such tools, it is necessary to manually write both the specification and the implementation semantics as a state transition system and then express verification goals in terms of this system. As a result, these studies did not bridge the gap between the high-level specification and the low-level RTL design, still requiring reliance on manual translation from RTL to the model.

6.4 Theorem Prover in Protocol Verification

Theorem provers have also been used to verify hardware designs (e.g., HOL to verify an academic microprocessor [57]). For protocol verification, such as cache-coherence proofs, mechanized theorem provers have been utilized as well. In the work of Vijayaraghavan et al. [55], a directory-based MSI protocol from a real design is verified using the Coq proof assistant. These approaches, including the ones verified using model checking earlier, focus on verifying a model of the actual system rather than directly verifying the RTL implementation. Another effort [10] uses a Bluespec-like language to describe the protocol, which is further synthesized into an RTL design. This technique

does not apply directly to an existing RTL design. It is worth noting that a considerable number of hardware designs are manually written at RTL, involving many performance-enhancing features; as a result, model checking continues to dominate hardware verification. Our work aims to fill the gap between the high-level protocol model and the low-level RTL design.

7 CONCLUSION

In this article, we proposed a new protocol implementation verification methodology based on using ILAs as modular component specifications. The ILA models play the role of a bridge between the verified formal specification S and the RTL implementation I. Existing ILA tools can check the sequential equivalence between a composition of the ILA models and S, and also support RC for each component in I against its modular ILA specification. The compositionality of this RC adds to verification scalability.

We demonstrated the ILA-based methodology on three hardware protocol case studies by performing complete FSIV. We found bugs in all three protocol implementations that were confirmed by designers. We also showed that successful protocol implementation verification could be completed in reasonable time despite the large RTL design state space. Overall, although the proposed methodology requires formal protocol and component specifications, the two-step problem decomposition (SEC and RC) and compositional RC enable significant improvement in protocol implementation verification.

REFERENCES

[1] Gul Agha, C. Gunter, Michael Greenwald, Sanjeev Khanna, Jose Meseguer, Koushik Sen, and Prasannaa Thati. 2005. Formal modeling and analysis of DoS using probabilistic rewrite theories. In *Proceedings of the International Workshop on Foundations of Computer Security (FCS'05)*.

[2] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. 1998. MOCHA: Modularity in model checking. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 1427. Springer, 521–525.

[3] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An open source manycore research framework. *ACM SIGPLAN Notices* 51, 4 (2016), 217–232. https://doi.org/10.1145/2872362.2872414

[4] Bruno Blanchet. 2012. Security protocol verification: Symbolic and computational models. In *Proceedings of the International Conference on Principles of Security and Trust*. 3–29.

[5] Marc Boulé and Zeljko Zilic. 2008. Automata-based assertion-checker synthesis of PSL properties. *ACM Transactions on Design Automation of Electronic Systems* 13, 1 (2008), Article 4, 21 pages. https://doi.org/10.1145/1297666.1297670

[6] Jerry R. Burch and David L. Dill. 1994. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*. 68–80. https://doi.org/10.1145/196244.196577

[7] Cadence Design Systems Inc. 2018. JasperGold: Formal Property Verification App. Retrieved November 26, 2019 from http://www.jasper-da.com/products/jaspergold-apps/

[8] Eduard Cerny, Surrendra Dudani, John Havlicek, and Dmitry Korchemny. 2015. *SVA: The Power of Assertions in SystemVerilog*. Springer International Publishing.

[9] Yean-Ru Chen, Wan-Ting Su, Pao-Ann Hsiung, Ying-Cherng Lan, Yu-Hen Hu, and Sao-Jie Chen. 2010. Formal modeling and verification for network-on-chip. In *Proceedings of the 2010 International Conference on Green Circuits and Systems*. 299–304. https://doi.org/10.1109/ICGCS.2010.5543050

[10] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), Article 24, 30 pages.

[11] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. 2004. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design*. Lecture Notes in Computer Science, Vol. 3312. Springer, 382–398. https://doi.org/10.1007/978-3-540-30494-4_27

[12] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. 2021. Wire sorts: A language abstraction for safe hardware composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*. 175–189. https://doi.org/10.1145/3453483.3454037

[13] D. Dill, A. Drexler, A. Hu, and C. Yang. 1992. Protocol verification as a hardware design aid. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, Los Alamitos, CA, 522–525. https://doi.org/10.1109/ICCD.1992.276232

[14] Bruno Dutertre. 2007. Formal modeling and analysis of the Modbus protocol. In *Proceedings of the International Conference on Critical Infrastructure Protection*. 189–204.

[15] Cindy Eisner, Irit Shitsevalov, Russ Hoover, Wayne Nation, Kyle Nelson, and Ken Valk. 2000. A methodology for formal design of hardware control with application to cache coherence protocols. In *Proceedings of the 37th Annual Design Automation Conference (DAC'00)*. ACM, New York, NY, 724–729. https://doi.org/10.1145/337292.337757

[16] Hugues Evrard. 2020. Modeling the raft distributed consensus protocol in LNT. *Electronic Proceedings in Theoretical Computer Science* 316 (April 2020), 15–39. https://doi.org/10.4204/eptcs.316.2

[17] J. Holzmann Gerard. 2003. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional.

[18] Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Pasareanu. 2018. Compositional reasoning. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 345–383.

[19] Perumalla Giridhar and Priyanka Choudhury. 2019. Design and verification of AMBA AHB. In *Proceedings of the 2019 1st International Conference on Advanced Technologies in Intelligent Control, Environment, Computing, and Communication Engineering (ICATIECE'19)*. IEEE, Los Alamitos, CA, 310–315.

[20] Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. 2018. Towards fully automated TLM-to-RTL property refinement. In *Proceedings of the 2018 Design, Automation, and Test in Europe Conference and Exhibition (DATE'18)*. 1508–1511. https://doi.org/10.23919/DATE.2018.8342253

[21] Birgit Hofer, Radu Mateescu, Wendelin Serwe, and Franz Wotawa. 2018. Using LNT formal descriptions for model-based diagnosis. In *Proceedings of the 29th International Workshop on Principles of Diagnosis*. 1–8.

[22] Bo Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. 2019. ILAng: A modeling and verification platform for SoCs using instruction-level abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, Vol. 11427. Springer, 351–357.

[23] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification. *ACM Transactions on Design Automation of Electronic Systems* 24, 1 (Dec. 2018), Article 10, 24 pages. https://doi.org/10.1145/3282444

[24] IEEE. 2005. *IEEE Standard for Property Specification Language (PSL)*. IEEE Standard 1850-2005. IEEE.

[25] Ranjit Jhala and Kenneth L. McMillan. 2001. Microarchitecture verification by compositional model checking. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 2102. Springer, 396–410.

[26] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. 2003. Checking cache-coherence protocols with TLA+. *Formal Methods in System Design* 22, 2 (March 2003), 125–131. https://doi.org/10.1023/A:1022969405325

[27] Zurab Khasidashvili, Marcelo Skaba, Daher Kaiss, and Ziyad Hanna. 2004. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'04)*. IEEE, Los Alamitos, CA, 58–65.

[28] Mallory Knodel and Niels ten Oever. 2020. Terminology, Power and Inclusive Language. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-knodel-terminology-02, accessed on: 2023-07.

[29] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional.

[30] Yongjian Li, Kaiqiang Duan, Yi Lv, Jun Pang, and Shaowei Cai. 2016. A novel approach to parameterized verification of cache coherence protocols. In *Proceedings of the 2016 IEEE 34th International Conference on Computer Design (ICCD'16)*. 560–567. https://doi.org/10.1109/ICCD.2016.7753341

[31] Xiaojun Liu. 1999. Formal specification and verification of a dataflow processor array. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design: Digest of Technical Papers (ICCAD'99)*. IEEE, Los Alamitos, CA, 494–499.

[32] Makai Mann. 2019. AXI Protocol Checker for the OH! Implementation of the AXI Protocol. Retrieved November 26, 2019 from https://github.com/upscale-project/case-studies/tree/master/axi

[33] Panagiotis Manolios and Sudarshan Srinivasan. 2008. A refinement-based compositional reasoning framework for pipelined machine verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16 (2008), 353–364. https://doi.org/10.1109/TVLSI.2008.918120

[34] Panagiotis Manolios and Sudarshan K. Srinivasan. 2005. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'05)*. https://doi.org/10.1109/ICCAD.2005.1560183

[35] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan. 2018. CoSA: Integrated verification for agile hardware design. In *Proceedings of 2018 International Conference on Formal Methods in Computer-Aided Design (FMCAD'18)*. 1–5. https://doi.org/10.23919/FMCAD.2018.8603014

[36] Kenneth L. McMillan. 1997. A compositional rule for hardware design refinement. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 1254. Springer, 24–35.
[37] M. N. Mneimneh and K. A. Sakallah. 2005. Principles of sequential-equivalence verification. *IEEE Design & Test of Computers* 22, 3 (2005), 248–257. https://doi.org/10.1109/MDT.2005.68
[38] Kedar S. Namjoshi and Richard J. Trefler. 2016. Parameterized compositional model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, Vol. 9636. Springer, 589–606. https://doi.org/10.1007/978-3-662-49674-9_39
[39] Minh D. Nguyen, Max Thalmaier, Markus Wedler, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. 2008. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 11 (2008), 2068–2082.
[40] Minh D. Nguyen, Max Thalmaier, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, and Jörg Bormann. 2009. A re-use methodology for formal SoC protocol compliance verification. In *Proceedings of the 2009 Forum on Specification and Design Languages (FDL'09)*. IEEE, Los Alamitos, CA, 1–6.
[41] Minh D. Nguyen, Max Thalmaier, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, and Jörg Bormann. 2009. A re-use methodology for formal SoC protocol compliance verification. In *Proceedings of the 2009 Forum on Specification and Design Languages (FDL'09)*. IEEE, Los Alamitos, CA, 1–6.
[42] Rishiyur Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMCODE'04)*. IEEE, Los Alamitos, CA, 69–70.
[43] John O'Leary, Murali Talupur, and Mark R. Tuttle. 2009. Protocol verification using flows: An industrial experience. In *Proceedings of the 2009 International Conference on Formal Methods in Computer-Aided Design (FMCAD'09)*. 172–179. https://doi.org/10.1109/FMCAD.2009.5351126
[44] Andreas Olofsson, Roman Trogan, Fred Huettig, Ola Jeppsson, and Peter Saunderson. 2016. Epiphany eLink AXI. Retrieved November 26, 2019 from https://github.com/aolofsson/oh/tree/master/axi
[45] Preeti Ranjan Panda. 2001. SystemC: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*. 75–80.
[46] Laurence Pierre. 2021. Refinement rules for the automatic TLM-to-RTL conversion of temporal assertions. *Integration* 76 (2021), 190–204.
[47] Paul Regnier, George Lima, and Aline Andrade. 2009. A TLA+ formal specification and verification of a new real-time communication protocol. *Electronic Notes in Theoretical Computer Science* 240 (2009), 221–238. https://doi.org/10.1016/j.entcs.2009.05.054
[48] Frank Rogin, Thomas Klotz, Gorschwin Fey, Rolf Drechsler, and Steffen Rulke. 2008. Automatic generation of complex properties for hardware designs. In *Proceedings of the 2008 Design, Automation, and Test in Europe Conference and Exhibition (DATE'08)*. 545–548. https://doi.org/10.1109/DATE.2008.4484908
[49] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. 2005. Transaction level modeling in SystemC. *Open SystemC Initiative* 1 (2005), 1–17.
[50] Abhik Roychoudhury, Tulika Mitra, and Satyanarayana R. Karri. 2003. Using formal techniques to debug the AMBA system-on-chip bus protocol. In *Proceedings of the 2003 Design, Automation, and Test in Europe Conference and Exhibition (DATE'03)*. IEEE, Los Alamitos, CA, 828–833.
[51] Murali Talupur and Mark R. Tuttle. 2008. Going with the flow: Parameterized verification using message flows. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*. 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.14
[52] Michael Bedford Taylor. 2018. INVITED: Basejump STL: SystemVerilog needs a standard template library for hardware design. In *Proceedings of the 55th Annual Design Automation Conference (DAC'18)*. 1–6. https://doi.org/10.1109/DAC.2018.8465909
[53] C. A. J. Van Eijk. 2000. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19, 7 (2000), 814–819.
[54] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. GoldMine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the 2010 Design, Automation, and Test in Europe Conference and Exhibition (DATE'10)*. 626–629. https://doi.org/10.1109/DATE.2010.5457129
[55] Muralidaran Vijayaraghavan, Adam Chlipala, and Nirav Dave. 2015. Modular deductive verification of multiprocessor hardware designs. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 9207. Springer, 109–127.
[56] Thuy Duong Vu, Li Zhang, and Chris Jesshope. 2008. The verification of the on-chip COMA cache coherence protocol. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology*. 413–429.
[57] Phillip J. Windley. 1995. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers* 44, 1 (1995), 54–72.

[58] Yue Xing, Huaixi Lu, Aarti Gupta, and Sharad Malik. 2021. Leveraging processor modeling and verification for general hardware modules. In *Proceedings of the 2021 Design, Automation, and Test in Europe Conference and Exhibition (DATE'21)*. 1–6.

[59] Yue Xing, Huaixi Lu, Aarti Gupta, and Sharad Malik. 2022. Compositional verification using a formal component and interface specification. In *Proceedings of the 2022 IEEE/ACM International Conference on Computer Aided Design (ICCAD'22)*.

Received 26 January 2023; revised 29 May 2023; accepted 8 July 2023