Work-in-Progress: Building an Interpreter for an Imperative Hardware Interface Specification Language

Nikil Shyamsunder, Francis Pham, Ernest Ng, Adrian Sampson, and Kevin Laeufer
Department of Computer Science
Cornell University, Ithaca, USA
Email: {nvs26, fdp25, eyn5, asampson, laeufer}@cornell.edu

***Abstract*—Composition of hardware modules is known to be error-prone: various components often require different timing of control and data signals. Solutions to this problem have involved standardizing generic handshaking protocols, raising the level of interaction to method calls that are compiled to standard interfaces, and interface specifications in temporal logics and type systems. We are currently working on an alternative approach. Our new domain-specific language (DSL) adapts the syntax of concrete unit tests or Universal Verification Methodology (UVM) drivers to precisely specify interface constraints from the perspective of an external component driving the design under test (DUT). We report our initial language design considerations and the challenges we have faced in writing an interpreter which executes concrete tests based on our interface specifications.**

I. Introduction

The testing and debugging of hardware designs at the Register-Transfer Level (RTL) consumes a significant portion of development time [1]. Assertion-based verification is a common paradigm for catching functional bugs in RTL [2][3]. Languages like SystemVerilog Assertions and the Property Specification Language let designers embed temporal checks directly into their testbenches [4] [5]. UVM provides a rich factory and transaction-based framework for testbenches [6].

At this level of abstraction, designers must interact with a module by driving values onto input ports over time and observing the resulting behavior. While this interface behavior is fundamental to functional correctness, it is rarely specified explicitly. Instead, designers rely on a mix of SystemVerilog unit tests, UVM testbenches, temporal assertions, and monitor/scoreboard infrastructure to encode expected transactions—implicitly scattering the protocol across disparate artifacts, making them susceptible to communication bugs [7].

To address this issue, we propose a new domain-specific language, PROTOCOLS, to make interface specifications explicit. Each protocol describes how a single transaction unfolds over time on a design under test (DUT), in the spirit of a concise, imperative unit test. The PROTOCOLS language distills the core ideas behind concrete testbenches — signal driving, temporal sequencing, and assertions — into a simple, cycle-accurate DSL that models inputs and expected outputs directly.

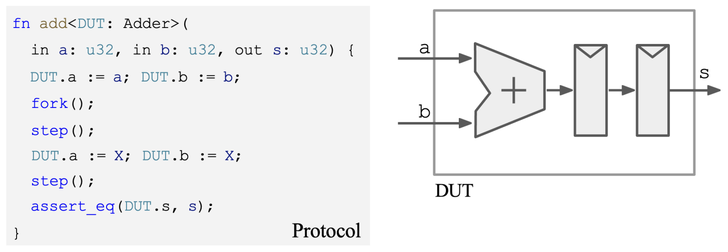
A protocol is a single source of truth for the hardware interface specification, which allows us to reuse the same transaction description across multiple verification contexts. We will be able to use the same protocol to drive concrete transactional tests, perform randomized testing, and generate UVM-style drivers and sequencers. It may also serve as a guide for other formal tools with precise, cycle-accurate interface constraints — all without modifying the original protocol definition.

II. Language Features

The PROTOCOLS language is designed to make the specification of the cycle-level RTL interface behavior clear and concise, using familiar syntax and abstractions inspired by hardware and software design. A protocol is described using an `fn` definition containing a list of imperative statements:

- `symbol := RHS` assigns the value of the `RHS` expression to the DUT input port `symbol`. The right-hand side expression may be an arbitrary value, represented by ✗ .
- `step(`n`)` advances the clock by n cycles.
- `fork()` allows for concurrent protocol execution.
- `assert_eq(`e_1`, `e_2`)` tests equality between e_1 and e_2.
- `while` and `if/else` blocks allow for control flow.

Modern hardware modules often support concurrent transactions, especially in pipelined or superscalar designs, which are modeled in PROTOCOLS using `fork()` semantics. In PROTOCOLS, transactions executed in parallel may write to shared input ports on the DUT, but advancing time for coroutines synchronized at `step` boundaries is only valid once the DUT's inputs are in a consistent and fully resolved state.



```
fn add<DUT: Adder>(
  in a: u32, in b: u32, out s: u32) {
  DUT.a := a; DUT.b := b;
  fork();
  step();
  DUT.a := X; DUT.b := X;
  step();
  assert_eq(DUT.s, s);
}
```

Fig. 1. A protocol specification for an adder.

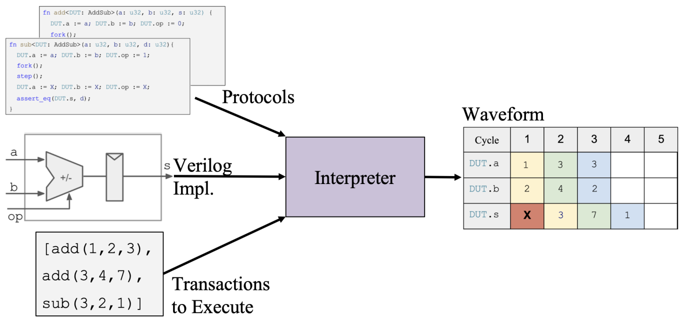
III. Interpreter

We chose to implement a concrete interpreter for PROTOCOLS to rapidly prototype the language and execute

https://doi.org/

concrete tests against real hardware modules. This enabled us to validate language features early and lay the groundwork for direct next steps like randomized testing.

The PROTOCOLS interpreter is driven by 3 inputs: (i) the interface specification in our PROTOCOLS language, (ii) the DUT implementation in Verilog, and (iii) a list of transactions to execute. The interpreter then runs the selected transactions and generates a waveform as well as error messages if a disagreement is detected.



```
fn add<DUT: AddSub>(a: u32, b: u32, s: u32) {
  DUT.a := a; DUT.b := b; DUT.op := 0;
  fork();
```

```
fn sub<DUT: AddSub>(a: u32, b: u32, d: u32){
  DUT.a := a; DUT.b := b; DUT.op := 1;
  fork();
  step();
  DUT.a := X; DUT.b := X; DUT.op := X;
  assert_eq(DUT.s, d);
}
```

```
[add(1,2,3),
 add(3,4,7),
 sub(3,2,1)]
```

Waveform

Cycle	1	2	3	4	5
DUT.a	1	3	3		
DUT.b	2	4	2		
DUT.s	X	3	7	1	

Fig. 2. The PROTOCOLS definitions for `add` and `sub` operations, along with a Verilog implementation and transaction list, are executed to produce a waveform.

A primary consideration during implementation was ensuring a consistent view of DUT inputs across all parallel transactions before allowing the simulation time to advance. An execution of a transaction within the interpreter may fail for two distinct reasons. First, multiple transactions may assign incompatible values to an input pin in the same cycle; this is a failure due to incorrect protocol definitions — parallel transactions must always be consistent with respect to each other. Second, assertions may fail, which generally points towards a bug in the DUT.

To determine if a consistent view of the inputs exists, we use fixed-point iteration and enforce a convergence condition: DUT input values are treated as evolving over a series of rounds *within a single clock cycle* until a consistent assignment is reached. Each input port of the DUT is associated with a value state that tracks how inputs change across scheduling rounds to ensure convergence. The possible value states are:

- `OldValue`(c): The stable value c of the port from the previous simulation cycle.
- `NewValue`(z): A new value z proposed during the current cycle by one or more concurrent transactions.
- `NoConstraint`: Indicates that no transaction has applied a concrete constraint to this input port value during the current scheduling cycle.

Each assignment in a transaction can be one of two types:

- `Concrete`: Proposes a definite value for the port.
- `DontCare`: Indicates that the current protocol does not constrain the value for this cycle, allowing other coroutines or the scheduler to determine the input freely.

Before the first round, all input values are `OldValues` from the previous cycle. A `Concrete` assignment of value

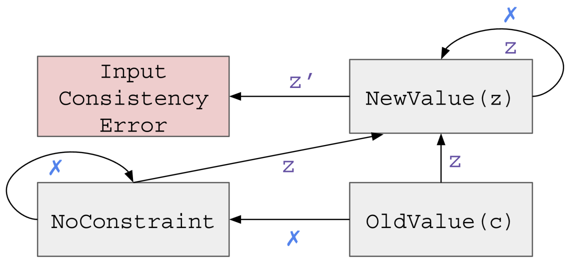


Fig. 3. Input port assignment semantics, where the initial state is `OldValue`(c), for arbitrary c, z, and z' with $z \neq z'$.

z updates a port in `OldValue(_)` or `NoConstraint` to `NewValue`(z), unless another thread has already proposed a conflicting `Concrete` assignment z'. A `DontCare` assignment converts `OldValue(_)` to `NoConstraint`, but never overrides an existing `NewValue`.

In each round of fixed-point convergence, an arbitrary ordering of threads is decided, and each thread runs until its next `step()`, proposing assignments as it executes. This process is repeated, and convergence is achieved when the input state after iteration i is equivalent to that after $i-1$. During this convergence phase, `assert` statements are ignored. If conflicting `Concrete` assignments arise, the transactions throw errors.

Once convergence is achieved, the threads are re-executed up to their `step()` point with assertions enabled, and any remaining ports in the `NoConstraint` state are randomized. Finally, simulation time advances, and all input values are marked as `OldValue` for the next cycle.

```
fn invert<DUT: inverter>(in a: u1, out b: u1) {
  DUT.a := DUT.b;
}
```

Iteration	1	2	3
(in) DUT.a	OldValue(0)	NewValue(1)	Error
(out) DUT.b	1	0	
Assignment	DUT.a := 1	DUT.a := 0	

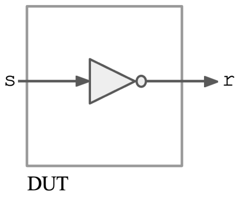


Fig. 4. Fixed-point iteration for a cycle where the DUT is a combinational inverter and the input port is initially 0. Even with a single coroutine, the Protocol fails due to introducing a cyclic dependency: the input `DUT.a` oscillates without reaching a stable value, triggering an error after 2 iterations.

IV. PRELIMINARY RESULTS & NEXT STEPS

We have used PROTOCOLS to specify and test several examples, such as an adder, a register file, and an AES-128 encryption implementation [8]. Looking ahead, a protocol can serve as the foundation for transactional testbenches, formal I/O sequence specification, and driver or sequencer generation in traditional verification frameworks like UVM. Additionally, by executing and capturing protocol traces, we can both verify correct behavior and reverse-engineer waveforms into human-readable protocol instances—bridging formal methods and test-driven development.

References

[1] H. Foster *et al.*, “The 2022 wilson research group functional verification study,” *Siemens. com*, 2022.

[2] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, “A survey on assertion-based hardware verification,” *ACM Comput. Surv.*, vol. 54, no. 11s, Sep. 2022. [Online]. Available: https://doi.org/10.1145/3510578

[3] Y. Tao, “An introduction to assertion-based verification,” in *2009 IEEE 8th International Conference on ASIC*, 2009, pp. 1318–1323.

[4] A. B. Mehta, *Systemverilog assertions and functional coverage*. Springer, 2020.

[5] “Ieee standard for property specification language (psl),” *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[6] N. B. Harshitha, Y. G. Praveen Kumar, and M. Z. Kurian, “An introduction to universal verification methodology for the digital design of integrated circuits (ic’s): A review,” in *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*, 2021, pp. 1710–1713.

[7] J. Ma, G. Zuo, K. Loughlin, H. Zhang, A. Quinn, and B. Kasikci, “Debugging in the brave new world of reconfigurable hardware,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 946–962. [Online]. Available: https://doi.org/10.1145/3503222.3507701

[8] N. I. of Standards, T. (NIST), M. J. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. D. Jr., “Advanced encryption standard (aes),” 2001-11-26 00:11:00 2001. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=901427