Multi-Language Hierarchical Programming Interface (ml-hpi)

Unifying Verification APIs Across Languages and Integration Levels

Author: Matthew Ballance

Abstract

SoC verification demands that the same test intent execute across radically different environments: block-level simulation, subsystem integration, full-chip simulation, emulation, and silicon bring-up — and across multiple languages: SystemVerilog, C++, Python, and PSS. The Multi-Language Hierarchical Programming Interface (ml-hpi) defines a single semantic model for verification APIs that generates correct, idiomatic bindings for every language automatically. This paper motivates the problem, identifies two verification-specific requirements that make a generic IDL insufficient, and shows how ml-hpi extends naturally to PSS — letting teams derive PSS infrastructure directly from an existing SystemVerilog interface.

1. The Test Reuse Problem

From Block to Subsystem

Consider a DMA engine undergoing block-level verification. An engineer writes a UVM virtual sequence, `do_transfer`, that drives a complete memory transfer: it programs source, destination, and length registers through the block's register agent, then waits on the interrupt monitor for a done indication. The sequence runs cleanly in the block testbench because the UVM component hierarchy it depends on is exactly what that testbench provides.

Now integration begins. The same DMA engine is instantiated in a larger subsystem alongside a memory controller and a bus fabric. The subsystem testbench has an entirely different UVM hierarchy: a different top-level environment class, a system-bus VIP instead of a direct register agent, and new arbitration and ordering constraints. The `do_transfer` sequence cannot compile in this environment — it references paths that simply do not exist.

The typical response is copy-paste: an engineer clones `do_transfer`, replaces every environment reference, and maintains a second version indefinitely. The two suites drift apart; integration-level tests provide less coverage than block-level tests rather than more.

The Interface Insight

The root cause is that `do_transfer` was written against a *concrete implementation* rather than an *abstract interface*. Had the sequence interacted with the environment via a `DmaIf` handle — an object that exposes `do_transfer` as a method — then any environment capable of providing a `DmaIf` implementation could run the sequence unchanged.

This "program to an interface, not an implementation" principle is well-established in software engineering. Applying it to hardware verification introduces one additional complication: verification work spans multiple languages, each with its own mechanism for declaring interfaces. Defining `DmaIf` in SystemVerilog does not make it available in C++, Python, or PSS. Without a language-neutral specification, each new language context requires yet another hand-written interface definition and yet another set of glue code connecting it to the others.

2. What Makes Verification Interfaces Different

A straightforward IDL (Interface Definition Language) that generates abstract class declarations in multiple languages would be a reasonable starting point — but two requirements specific to verification environments make a generic IDL insufficient.

2.1 Hierarchical Context

Real designs contain many instances of the same IP: four DMA channels, eight PCIe ports, sixteen DRAM banks. A flat, procedural API cannot address a specific instance cleanly. Adding an explicit handle argument to every function (`dma_do_transfer(ctx, src, dst, len)`) works at the C ABI level but loses the structure at higher levels: callers must track raw handles, the API surface grows linearly with the number of IP types, and there is no language-supported way to express that two handles represent the same type of sub-system.

ml-hpi models each API as a class hierarchy. An interface may contain *field* members (a single named sub-interface) and *array* members (an indexed collection of sub-interfaces of the same type). A chip-level interface grouping a DMA engine and a set of UART ports might look like:

```yaml
interfaces:
  - name: dma.DmaIf
    methods:
      - name: do_transfer
        rtype: void
        params: [{name: src, type: addr}, {name: dst, type: addr}, {name: len, type: uint32}]
        attr: [{blocking: true}, {target: true}]
  - name: uart.UartIf
    methods:
      - name: send
        rtype: void
        params: [{name: data, type: uint8}]
        attr: [{blocking: true}, {target: true}]
  - name: chip.ChipIf
    members:
      - {name: dma,   kind: field, type: dma.DmaIf}
      - {name: uarts, kind: array, type: uart.UartIf}
```

In SystemVerilog this becomes:

```
interface class DmaIf;
  pure virtual task do_transfer(input longint unsigned src,
                                input longint unsigned dst,
                                input int unsigned len);
endclass
interface class UartIf;
  pure virtual task send(input byte unsigned data);
endclass
interface class ChipIf;
  pure virtual function DmaIf  dma();
  pure virtual function UartIf uarts_at(int idx);
```

```
  pure virtual function int    uarts_size();
endclass
```

A test navigates naturally: `chip.uarts_at(1).send(ch)`. Each interface instance is a typed object; no handle bookkeeping is required, and type correctness is enforced by the language. Crossing a flat DPI or FFI boundary while preserving this hierarchy is handled transparently by the generated binding layer — callers use typed objects; the generated glue handles the rest.

2.2 Blocking Calls

The second requirement concerns time. Whether a method is *blocking* — whether it must consume simulated clock cycles before returning — is a property of the operation, not of the language. A `do_transfer` that waits for a done interrupt must block; a `get_status` register read need not.

The complication is that different languages express blocking differently. In SystemVerilog a blocking call must be a `task`; a non-blocking call must be a `function` — mixing them is a compile error. In Python the natural form is an `async def` coroutine so the cocotb event loop remains responsive. In PSS, *target-time* actions (which may block) are structurally separate from *solve-time* computations (which must not).

ml-hpi captures `blocking`, `target`, and `solve` as per-method attributes. Generators emit the correct construct for each language automatically — `pure virtual task` in SV, `async def` in Python, a callback-based completion import for C. The interface author states *what* each method does; the generator decides *how* each language expresses it.

3. A Common Semantic Model

The Fragmentation Problem

Without a shared specification, the same logical operation accumulates a separate definition in every language context:

Language / context	Expression of `do_transfer`
C testbench	`dma_do_transfer(ctx, src, dst, len)` — flat C, explicit handle
C++ model	`dma->do_transfer(src, dst, len)` — virtual method, custom class
SystemVerilog UVM	`env.dma_agent.seq_if.do_transfer(...)` — deep UVM path, task
Python / cocotb	`await dma.do_transfer(src, dst, len)` — coroutine
PSS	`dma_transfer` action, `exec target_action` body

Each definition is hand-written and maintained independently. A signature change requires five coordinated edits; a type mismatch silently compiles in one language and fails at simulation time in another.

One Model, Many Representations

The core of ml-hpi is a *semantic model* — a language-neutral description of interfaces, methods, types, hierarchy, and per-method attributes. JSON/YAML is one way to record that model, but not the only way. A SystemVerilog `interface class` hierarchy is itself a complete, valid expression of the same model: tasks map to blocking methods, functions to non-blocking ones, indexed accessors to array members. So is a C++ pure-virtual class hierarchy or a Python ABC.

A team whose primary language is SystemVerilog need not adopt a new file format. They author their interface in SV and the ml-hpi toolchain *derives* the JSON/YAML intermediate representation automatically, then generates bindings for C++, Python, C, and PSS in one step:

```
author SV interface class  →  ml-hpi derives IDL  →  C++, Python, PSS, C bindings
       (maintained by team)       (hidden artifact)      (never hand-written)
```

Because all representations share the same underlying model, the toolchain can also *compare* specifications expressed in different languages, detecting drift between a SV interface and a C++ class that are supposed to represent the same API.

4. Test Portability in Practice

Writing a Portable Test

With `DmaIf` defined (in SV or YAML), a UVM sequence becomes genuinely portable:

```
class DmaTransferSeq extends uvm_sequence;
  DmaIf dma;    // injected - no hardcoded env path
  task body();
    dma.do_transfer(SRC_ADDR, DST_ADDR, LENGTH);
  endtask
endclass
```

The sequence holds a `DmaIf` handle that is injected by the testbench at run time. It contains no reference to any UVM component path. It compiles and runs in any environment that provides a `DmaIf` implementation.

Block-Level Environment

The block testbench creates a `DmaIfBlockImpl` — a class implementing `DmaIf` that drives the DMA's register bus directly through the block-level register agent. `DmaTransferSeq` is unaware of this; it calls `dma.do_transfer(...)` and awaits the task completion. All existing directed and constrained-random sequences written this way run without modification.

Subsystem-Level Environment

The subsystem testbench provides a `DmaIfSubsysImpl` — a different implementation of the same `DmaIf` interface. Internally it routes transactions through a system-bus VIP over a shared interconnect; timing, arbitration, and bus width are all different. The sequence file is **identical**. The subsystem team writes one implementation class and immediately inherits the full block-level test suite as integration tests — tests that are now verifying end-to-end paths through the interconnect rather than just the DMA block in isolation.

SoC, Emulation, and Silicon

The same pattern scales further. A C++ virtual platform, a Python cocotb harness, and a bare-metal C driver each provide their own `DmaIf` implementation; `DmaTransferSeq` is never touched. Coverage closed at block level travels with it — subsystem and SoC verification resources focus on integration defects, not re-verifying known block behavior.

5. Extending to Portable Stimulus (PSS)

PSS and Platform Interfaces

PSS (Portable Test and Stimulus Standard) expresses verification intent as a portable action graph. PSS actions call platform functionality through `function import` declarations: the PSS file specifies the signature; the platform side (C, C++, or SV) provides the body. Without a shared interface spec, teams hand-write three synchronized artifacts per function per platform — `function import`, DPI/FFI stub, and dispatch logic — and keep them in sync across every environment.

The SV → IDL → PSS Pipeline

ml-hpi eliminates this duplication. Because the toolchain can derive the IDL from an existing SV `interface class`, and because the IDL captures all attributes PSS needs (`blocking`, `target`, `solve`), the toolchain can generate complete PSS `component` and `function` declarations in the same step that produces the C++ and Python bindings. The user-facing workflow is:

```
SV interface class  --> (ml-hpi derives IDL) --> PSS component + function imports
                                             +--> C++ bindings
                                             +--> Python bindings
                                             +--> DPI glue
```

The IDL is never seen by the user; the PSS infrastructure is never hand-written. The SV `interface class` remains the single maintained artifact.

The value goes beyond eliminating boilerplate. PSS natively thinks in hierarchical terms — a test author writes `comp.dma.do_transfer(...)` or `comp.uarts_at(1).send(...)` against a component tree that mirrors the design hierarchy. Without ml-hpi, realizing that hierarchical view requires manually flattening it to a set of PSS `function import` declarations, writing C dispatch stubs that reconstruct the context, and then tunneling back up through a DPI export layer into a hierarchical UVM environment — hand work spread across three languages that must stay synchronized. ml-hpi derives all of that translation automatically from the shared semantic model: the component tree, the function signatures, and the DPI glue are generated together and are guaranteed consistent.

Reusing Existing Implementations

PSS portability rests on the same foundation as UVM sequence reuse. The PSS action graph calls `dma.do_transfer(...)` through the component-based API. At block level the registered implementation is exactly `DmaIfBlockImpl` — already used by UVM sequences, not a new artifact. At subsystem level `DmaIfSubsysImpl` serves both UVM sequences and PSS-generated tests unchanged. The PSS model never changes; only which implementation is registered differs across execution contexts.

6. Conclusion

The cost of duplicating verification interfaces across languages and integration levels is hidden but substantial. Teams spend engineering cycles re-expressing the same API, maintaining diverging copies, and debugging mismatches between hand-written binding layers. ml-hpi addresses this by providing a common semantic model for hierarchical verification APIs — one that captures the two verification-specific requirements (instance context and blocking character) that generic IDLs miss, and that can be expressed in or derived from whichever source language a team already uses.

The result is a single interface definition that yields idiomatic, correct bindings for SystemVerilog, C++, Python, C, and PSS simultaneously. Tests written against that interface travel from block to subsystem to SoC without modification. The same spec that enables UVM sequence reuse automatically generates the PSS `function import` infrastructure needed for portable stimulus — turning a one-time interface authoring investment into leverage across every level of the verification hierarchy.

Appendix: ml-hpi at a Glance

Concept	Description
Interface	Named set of methods and sub-interface members
Field member	Single named sub-interface (e.g., `dma`)
Array member	Indexed collection of sub-interfaces (e.g., `uarts[n]`)
`blocking` attribute	Emitted as `task` in SV, `async def` in Python, callback import in C
`target` / `solve` attributes	Maps to PSS target-time / solve-time

Language	Abstract interface	Generated glue
SystemVerilog	`interface class`	DPI export/import stubs, root registration
C++	Pure-virtual class	FFI dispatch stubs
Python	`abc.ABC` subclass	`ctypes` wrappers / async adapters
C	Function-pointer struct	Flat C ABI wrappers
PSS	`component` + `function import`	Target-time exec body stubs

Authoring workflows

```
Direct:   YAML/JSON spec  --> all language bindings

Derived:  SV interface class  --> (ml-hpi derives IDL)  --> C++, Python, C, PSS bindings
```

References and Project Links

ml-hpi project repository https://github.com/fvutils/ml-hpi

Full specification, source code, and language binding generators are available at the project repository. Documentation is published at https://fvutils.github.io/ml-hpi.