Making Concurrent Hardware Verification Sequential

THOMAS BOURGEAT, EPFL, Switzerland
JIAZHENG LIU, MIT, USA
ADAM CHLIPALA, MIT, USA
ARVIND, MIT, USA

Compared to familiar hardware-description languages like Verilog, rule-based languages like Bluespec offer opportunities to import modularity features from software programming. While Verilog modules are about connecting wires between submodules, Bluespec modules resemble objects in object-oriented programming, where interactions with a module occur only through calls to its methods. However, while software objects can typically be characterized one method at a time, the concurrent nature of hardware makes it essential to consider the repercussions of invoking multiple methods simultaneously. Prior formalizations of rule-based languages conceptualized modules by describing their semantics considering *arbitrary sets of simultaneous method calls*. This internalized concurrency significantly complicates correctness proofs. Rather than analyzing methods one-at-a-time, as is done when verifying software object methods, validating the correctness of rule-based modules necessitated simultaneous consideration of arbitrary subsets of method calls. The result was a number of proof cases that grew exponentially in the size of the module's API.

In this work, we side-step the exponential blowup through a set of judicious language restrictions. We introduce a new Bluespec-inspired formal language, Fjfj, that supports *sequential characterization of modules*, while preserving the concurrent hardware nature of the language. We evaluated Fjfj by implementing it in Coq, proving the key framework principle: the refinement theorem. We demonstrated Fjfj's expressivity via implementation and verification of three examples: a pipelined processor, a parameterized crossbar, and a network switch.

CCS Concepts: • **Hardware → Hardware description languages and compilation**; **Theorem proving and SAT solving**.

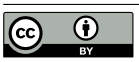
Additional Key Words and Phrases: hardware description languages, modular formal verification

ACM Reference Format:
Thomas Bourgeat, Jiazheng Liu, Adam Chlipala, and Arvind. 2025. Making Concurrent Hardware Verification Sequential. *Proc. ACM Program. Lang.* 9, PLDI, Article 228 (June 2025), 25 pages. https://doi.org/10.1145/3729331

1 Introduction

We are in the midst of a flurry of work developing new digital hardware designs. Companies that we used to think of as primarily software or service companies are now designing very sophisticated hardware for their own products. For example, Google designed their Tensor system on a chip for phones, Apple designed the M cores for laptops and tablets, and Amazon AWS designed their Graviton processors for the cloud. Even Tesla, primarily an automotive company, designed a chip for their cars. This trend can be seen as the real-world materialization of a foundational principle of computer architecture: *the more we know about the typical workload, the more efficient a machine we can design.*

Authors' Contact Information: Thomas Bourgeat, EPFL, Lausanne, Switzerland, thomas.bourgeat@epfl.ch; Jiazheng Liu, MIT, Cambridge, USA, jzliu@csail.mit.edu; Adam Chlipala, MIT, Cambridge, USA, adamc@csail.mit.edu; Arvind, MIT, Cambridge, USA, arvind@csail.mit.edu.



ACM 2475-1421/2025/6-ART228
https://doi.org/10.1145/3729331

As long as engineers continue taking advantage of that mantra, there will be a steady stream of new hardware development, and the cost of that development matters. While the hardware world outstrips software in industrial use of formal methods, dramatic opportunities for cost savings remain, given that many hardware projects devote majorities of their budgets to verification (formal and testing-based). Formal verification in the hardware industry is most commonly based on model checking, especially bounded model checking. These techniques have clear appeal in automation, but they still face classic state-space-explosion problems that limit their use to either bounded verification, relatively small hardware designs, relatively simple specifications, or all of the above.

A key advantage of many celebrated software-verification methods is *modularity* in specification and proofs [Barnett et al. 2006; Leino 2017]. A complex implementation can be broken into reusable pieces, with each specified and proved independently, perhaps depending on the specifications of other pieces but not their implementations. For instance, a class can be verified against a specification that assigns a precondition and postcondition to each method. Then methods can be proved individually, using verification-condition generation targeting SMT solvers.

The goal of this paper is to achieve the same form of modular verification for hardware modules.

An intuition about the main roadblock comes from the fact that hardware is fundamentally concurrent, and it is well-known that the specifics of software verification get much more complicated moving to concurrent programs. There, proof methods have evolved from relatively tame methods like rely-guarantee [Jones 1983] to extremely flexible frameworks like Iris [Jung et al. 2015]. The situation is simplified substantially by narrowing the concurrency idiom to *transactions*, with a runtime system to guarantee that every method call appears to execute atomically.

In this paper, we share our new insights on *how rule-based hardware languages can be adapted to support that style of transactional verification*. In the first section of this paper, we review key practical considerations of digital hardware design. This section serves as an introduction to our language, the framework, and its specifics.

Then we explain the key metatheorems that hold in the framework and show a simple example of a proof of refinement in the framework. Finally we demonstrate that Fjfj is suitable to do both processor and accelerator verification by evaluating our methodology on 3 examples: a pipelined processor, a programmable network switch, and a crossbar. All results presented have been mechanized in Coq.

The contributions of the paper are:

- A proof framework to reason about rule-based hardware designs *one method/rule at a time*
- A mechanization of the refinement principle, key to our reasoning strategy
- An evaluation of the methodology with specific correctness proofs for three designs showcasing interesting challenges and insights into specification of hardware designs

2 Software-Like Abstractions for Hardware Description

Let us consider standard notations for defining hardware systems, progressing through thornier and thornier challenges that motivate higher-level abstraction. By the end, we hope to have justified our new core language Fjfj, which is heavily influenced by Bluespec. An informal introduction to Fjfj follows in section 3, after which we justify a key new restriction just introduced (section 4) and present the formal semantics (section 5) and key modular proof principles (section 6).

The lowest level of abstraction we will consider in this paper is so-called *register-transfer level (RTL)* for a synchronous circuit with a single clock. At this abstraction level, a circuit is described by three components: a finite collection of Boolean state elements (registers), a Boolean function $f(s, i)$ that specifies how the state should be updated based on old state and the inputs, and another Boolean function *out* that specifies the outputs depending on the previous state and the inputs.

When given such a circuit description, its semantics are straightforward to define: $s_{t+1} = f(s_t, i_{t+1})$, $o_{t+1} = out(s_t, i_{t+1})$. The question of composing such circuits together is slighty trickier, as one must beware the risk of inadvertently tying an input to an output, without going through `s` (called a combinational loop), which would make the previous recursion not well-founded.

When one writes `z=f(x,y)` as one line among many in a hardware design, it does not indicate that a function `f` is called "at runtime." It means that a compiler should generate a Boolean function `f` and connect wires corresponding to the values `x` and `y` to the input, and it should connect the output of the Boolean function to `z`. In particular, if such a box `f` does not have outputs, it can be completely eliminated, as there is no notion of side effects.

One might be tempted to think of `f` as calling a pure function in a functional language. However, here the analogy breaks down and gives a misleading impression. For example, there is no notion of recursion in RTL. Even more important, such an analogy hides the reality that `f` will always be physically present in the hardware, and at this level of abstraction `f` *cannot not be called.* RTL simply describes a DAG of Boolean operators, with an implicitly fully concurrent model of execution.

While remaining grounded in this fairly chaotic language model (since it is a practical choice for synthesis of physical circuits), we strive to regain as much as possible of the software-verification experience in the tradition of Hoare logic. The key tool is reintroducing more of a software-style method-call abstraction, carefully tweaked to confront just enough of the realities of hardware.

Hardware circuits do often contain blocks of digital logic that are easily seen as implementing callable methods. Such a circuit canonically has an "enable" wire flowing into it, and this bit should be set to 1 on exactly those cycles when "the method is being called." In a literal, physical sense, the logic of the method runs every cycle. However, we introduce logic to *ignore* the method effects when "enable" is 0, in computing the new values of affected registers.

Consider the example of an integer counter with methods to get, set, and increment by one. When the enable bit of the set method is high, the new state (value of a register considered private to the counter) is the argument put on the wire. Otherwise, we leave the state of the module unchanged. Similarly, if the enable bit for increment is high, the new state is the application of a "+1" circuit to the old state. The get method is pure observer and does not change the state, so we need not introduce extra signals to signify to the circuit when we want to "call" this method.

In this simple regime, each method call takes one cycle. If an enable wire stays at 1 across cycles, we consider that a new call happens per cycle. In this setting, if we were thinking of a software construction `c.inc(); c.set(0); c.inc()`, one reasonable way to interpret the software semicolon would be, "wait for an arbitrary number of cycles, disabling all methods during this time." Such a counter can seemingly be characterized by explaining each method in isolation.

Note, however, that it seems that the arbitrary number of cycles for a semicolon must be at least one, as if `c.inc()` already wrote a value to the counter, it is incoherent to ask to rewrite a new value the same cycle. Presuming an appropriate delay between method calls, we might hope to construct an ordered trace of method calls, such that our "object" seems to behave as if those calls were executed serially. Sadly, this model is a poor fit to the realities of hardware circuits. Consider:

- `getA` and `getB`, getters for two private fields `A` and `B`, which would be implemented in hardware by two registers
- `setA` and `setB`, the corresponding two setters

Problem introduced by concurrency. Now we consider setting the enablement bits for `setA` and `setB` simultaneously, and we connect the output of `getA` to the argument of `setB` and the output of `getB` to the argument of `setA` (naively trying to swap the field values). The next state obtained by this driving of the wires is equivalent to neither `setA(getB()); setB(getA())` nor the other order. Driving wires this way leads to a race that creates a behavior not sequentially explainable, so

we would need to specify a new relation to say what it means to do `setA` and `setB` "simultaneously." More generally, we would potentially need to specify an exponential number of relations to describe what is the effect of calling any subset of methods simultaneously. That approach was taken in previous work on verification for Bluespec-style hardware modules, Kami [Choi et al. 2017].

For example, a Kami proof of a queue module requires a proof case for enqueue and dequeue happening at once. The case may quickly be ruled out as contradictory because the two methods both write a common register, which Bluespec must forbid. However, that reasoning about a contradiction must be carried out explicitly for every subset of the methods exported by a module.

It is also tempting to think of method-call semantics as similar in hardware to software, where (without recursion) we may simply substitute every method call with a suitably rewritten version of the method's definition. The trouble is that a hardware method corresponds to a subgraph of a physical circuit, and one consequence is that a method may only be called once per cycle. To support multiple calls, we would need to clone the method. An inlining transformation may in general hide that restriction and allow illegal (physically impossible) executions to proceed.

To recap, we have managed to fake a form of method calls, but when we call multiple methods "at the same time," the semantics become burdensome. If we forbid calling two arbitrary methods of the same module in the same cycle (i.e. never setting two enable wires to one), we sidestep the problem, but that would be too drastic as it would prevent writing most interesting designs. Let us instead adopt a slight variation: allow an arbitrary number of observation methods (value methods like gets) per cycle, while at most one action method (updating the state, like set) may be called.

Logical atomicity versus clock cycle. At this point, even a novice hardware designer is ready to protest that our restriction of allowing only one update method is excessive, standing in the way of achieving performance through parallelism. Consider a queue module in a pipeline. One would want to be able to enqueue and dequeue from it each cycle, since otherwise we miss the point of improving parallelism through pipelining. A key insight of Bluespec is to promise to show programmers an abstraction of *logical* cycles where we may enforce one method call per object, while using clever compilation to realize a *physical* reality of greater concurrency. Hence, even though our restriction states that a single action method of a submodule can be called within the definition of a parent action method or parent *rule* (to be introduced in the following paragraph), on each *clock cycle* multiple methods may be called, simply restricted to be made by different rules.

So far the objects interfaced with through methods and the corresponding circuits with ready and enable signals we outlined actually map to each other one-to-one fairly straightforwardly. However, there exists one very common and central hardware pattern that does not have a standard software counterpart. We introduce that pattern now and explain how it is not too difficult to model with a software counterpart. This pattern is called *rules*.

Consider a hardware circuit that performs a multiplication by repeated addition.

```
inputs in_a, in_b
output reg [63:0] res
reg [31:0] a, b
reg [5:0] cnt

// Wire to request multiplication of two numbers
if (EN_multiply) {a = in_a; b = in_b; cnt = 0; res = 0;}

// When the method is not called, the circuit continues,
// computing the binary multiplication algorithm.
if (!EN_multiply & cnt < 32) {cnt = cnt+1; res = res + (b[cnt] ? a<<cnt : 0);}
```

As conceptually each action-method call runs in exactly one cycle, the background computation (last `if` above) is not part of the method call. It instead is better thought of as another transition, a spontaneous method that is always called when possible, without arguments: an epsilon transition. Such a piece of logic is called a rule in Bluespec, where a module may contain an arbitrary number of rules, each available to execute nondeterministically and atomically.

The previous example also brings us to the last common pattern that requires introduction and careful semantic treatment. Notice the delicacy in this example stemming from the fact that the result of the multiplication is unstable until computation finishes, so it is not easy to know when we can use the value on the `result` wire. In hardware, the design pattern is simply to add another "ready" wire (set to `cnt == 32` here). Similarly to the way one cannot dequeue from an empty queue, arbitrary circuits might have arbitrary guard conditions corresponding to the usage of some of their externally facing methods (or to the execution of internal rules).

In Bluespec, ready signals are handled by the compiler: the compiler introduces a ready signal for every method and rule, and calls to unready methods and execution of unready rules are blocked (the corresponding enable signal of a method will always be set to 0, when the method is not ready), which will exert backpressure that prevents parent rules from firing. To give a preview of what such a multiplier would look like in a rule-based language, the programmer would simply write the following code and rely on the compiler to compile to RTL, adding the required signals.

```
module mkMul(MultiplierIfc);
  Reg#(Bit#(6)) cnt <- mkReg(0);
  Reg#(64) result <- mkReg(0);
  Reg#(32) a <- mkReg(0);
  Reg#(32) b <- mkReg(0);

  rule compute if (cnt < 32);
    cnt <= cnt + 1; result <= result + (b[cnt] ? a<<cnt : 0);
    // (the barrel shifter on previous line could easily be optimized,
    // but it goes beyond the goal of this example)
  endrule

  method Action mul(Bit#(32) in_a, Bit#(32) in_b);
    a <= in_a; b <= in_b; cnt <= 0; result <= 0;
  endmethod

  method Bit#(64) result() if cnt == 32;
    return result;
  endmethod
endmodule
```

The absence of control signals in the high-level source is a key advantage of rule-based languages for verification. The ready signals *do not appear* in the semantics of the language. Rule-based languages shift responsibility for control signals to the compiler.

We now have all our pieces of restricted Bluespec: methods and rules, each able to impose guards, all sitting within arbitarily deep tree hierarchies of modules. Each rule or method is allowed at most one side-effecting method call to any given submodule.

3 Fjfj Informally

Our top goal is hardware development where each module can be proved individually to refine an appropriate specification, and then such a library of modules can be assembled into a variety of different verified systems, without needing to revisit module implementations or proofs. To that end, following the analysis from the prior section, we introduce Fjfj.

Consider a canonical "hello world" example of computer architecture, a two-stage pipeline computing a pure function. We will implement a module computing $g \circ f$. The implementation of such a module would be given by the following Lisp-inspired syntax (which Coq parses).

```
Local Instance submodules : instances := #|
    QueuePkg.mkQueue1 inp;
    QueuePkg.mkQueue1 mid;
    QueuePkg.mkQueue1 out |#.

Definition pipe_enq := (action_method (el) {enq inp el}).
Definition do_f := (rule (begin (set el {first inp}) {deq inp} {enq mid (f el)})).
Definition pipe_deq := (action_method () {deq out}).
Definition pipe_first := (value_method () {first out}).
Definition do_g := (rule (begin (set el {first mid}) {deq mid} {enq out (g el)})).

Global Instance mkPipeline : module _ :=
    module#(rule [do_f; do_g] vmet [pipe_first] amet [pipe_enq; pipe_deq]).
```

Because Fjfj is a DSL embedded in Coq, we first identify the different syntactic elements of such snippets. We use Coq forms like `Definition NAME : COQTYPE := BODY.` or `Local Instance NAME : COQTYPE := BODY.` to define Coq-level values. The bodies, on the right sides of the `:=` symbols, are the actual Fjfj terms and objects. The Fjfj form `(begin a b c d)` used in the body of the `do_f` rule is syntax for a sequence of actions or expressions.

The submodules ("private fields" of the main module) of this example are three queues, which are the only part containing mutable state. The `pipe_enq` method simply enqueues a value in `inp`, while the `do_f` rule picks the first element from `inp` and enqueues it in `mid` after modifying it by function `f`. It also dequeues `inp`. Curly braces delimit the concrete syntax for method calls. The rest of the example has a similar explanation, where `vmet` stands for value (pure) method and `amet` for action (side-effecting) method. Contrary to curly braces that delimit the syntax for method calls, parentheses like `(g el)` denote the syntax for calling a combinational function. Semantically combinational functions are simply pure functions, and they can simply be inlined. Finally, the `set el` form needs to be considered carefully: it is not like a software variable assignment. Such local variables are simply wire labels, giving names to intermediate expressions, reminiscent of `let` in functional languages. Importantly, setting a variable is not the same thing as calling the write method of a register, as setting a variable does not cause a side effect on any submodule.

The top-level language semantics is *repeated nondeterministic choice of a rule to execute.* Hence, the compiler in charge of compiling the rule-based description to a circuit is left free to construct an arbitrary scheduling circuit to pick which sequence of rules to schedule each cycle. In rule-based languages, functional correctness of designs should never be impacted by scheduling choice. While rules execute nondeterministically on their own, methods need to be called.

One question left to understand this snippet is the definition of the queue modules. One valid implementation would be a one-element queue, defined in the same language using only registers:

```
Local Instance submodules : instances := #|
    reg.mkReg valid;
    reg.mkReg data |#.
```

```
Definition enq := (action_method (el)
                    (if (not {read valid})
                        (begin {write valid 1} {write data el})
                        abort)).

Definition first := (value_method ()
                        (if {read valid} {read data} abort)).

Definition deq := (action_method ()
                      (if {read valid} {write valid 0} abort)).

Global Instance mkQueue1 : module _ :=
    module#(vmet [first] amet [enq; deq]).
```

When a method does not succeed because the method dynamically aborts, the abort propagates to the caller, causing the caller to abort, all the way to an ancestor rule. In other words, a rule might not be ready because a nested submodule that it is trying to call indirectly is not ready. Note however that another rule might (and typically will) be ready. For example, when we start a processor, because there are no instructions in-flight, there is a single rule that will be ready: the fetch rule. Other rules will be aborted because they will typically try to call the dequeue methods of empty queues of instructions.

Unusual aspects. Consider a simple register `r`. The object view of a register is simply a module with two methods: a value method `read` and an action method `write`. Now consider trying to replicate this software-style coding pattern: `r.write(2); value = r.read()`. One might expect `value` to be 2, but that is not the behavior in rule-based languages. Instead `value` gets the old value. Indeed, if we look at the module boundary with the registers, there are simply wires to declare if we want to do a write this clock cycle. The hardware cannot *sequentially call* methods of the register. It can only call all the methods desired simultaneously, and these simultaneous calls will look like first the `read` method was called and then the `write` method was called. So when one has compiled a submodule with the discipline to think of it like a software object, the effect of calling multiple of its methods within the clock cycle is fixed at compile time of the module, and the effect is one specific ordering. For registers, during one atomic step of execution, all state reads view *initial values*, even if state writes have happened in between within the atomic action.

Another interesting observation is that it is impossible to do separate compilation of a submodule and then allow a parent to call the same action multiple times within the same clock cycle, as there is only one set of wires corresponding to the method call. Notice that this restriction is qualitatively very different from the language restriction we mentioned earlier (restriction of one action method per *logical cycle*): it is not a limitation of our language but rather a *structural hazard* in the hardware that exists in every hardware language.

In a circuit, all reads of a register naturally read the version from before the cycle began. It would also be prohibitively expensive to allow some state updates to persist while blocking others due to aborts, requiring extensive use of demultiplexing gates with complex Boolean signals. The idea of aborts might seem unusual to hardware engineers, who are not used to transactions, but it is actually very commonly used without being named, through carefully crafted *control logic*. Both Bluespec and FjFj include it because of how it enables clean modularity, where each module may impose its own conditions on method callability, without burdening callers with the details.

One might wonder what are the basic primitive modules that we can start from. In a traditional setup where we only do design, we would just have registers holding bounded-size data as primitive building blocks. It is known that rule-based programs using only such registers as leaf submodules

can be compiled to efficient RTL [Greaves 2019; Nikhil 2004]. This paper does not discuss the synthesis of Fjfj. Because the language is morally a restrictive subset of Bluespec, we typically start from a Bluespec design and one-to-one translate it (currently manually) to Fjfj. This way we can use the Bluespec compiler on the original source design for synthesis. The composition of a proof of design correctness and correctness of a rule-based language compiler as in Bourgeat et al. [2020] is not in-scope for this paper.

Importantly, Fjfj allows us to define new primitive modules with custom semantics: the basic building blocks are not limited to registers. When we introduce arbitrary primitive modules, we lose synthesizability (the hardware word for "compilability"), but we gain ease of expressivity and better modeling facilities, easing verification. Those nonsynthesizable primitive modules will typically be used both for the top-level specification and for intermediate specifications during verification, but they will not be present in the actual implementation design. We now explain how we represent the semantics of our so-called primitive modules.

3.1 Primitive Modules and Semantics of Modules

The semantics of a module, noted $\mathbb{M}$, is described by the following collection that conceptually represents a labelled transition system (LTS). Note that, for simplicity in this exercise (and our implementation), we enforce that all method arguments and return values are unbounded natural numbers (we encode pairs and other types in natural numbers). In the following let T be the *internal module type*: the type of data that the module manipulates. Also take $\mathbb{P}$ as the type of logical propositions.

- A collection of *rules*' transition relations, where each rule is an update relation on state: $r : T \to T \to \mathbb{P}$.
- A collection of *value methods*' observation relations, where each method is an observation relation on the state: $v : T \to \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ which relates a state, an argument, and a returned value.
- A collection of *action methods*' transition relations, where each method is a state-transformation relation: $a : T \to \mathbb{N} \to T \to \mathbb{P}$ which relates a state, an argument, and a new state.

The three fields of $\mathbb{M}$ sometimes will be derived from actual pieces of syntax for rules and methods and submodules (*i.e.* see section 5 for the actual definition of [[M]]), but for the *primitive modules*, they are directly defined as three relations and a type in Coq. Note that these relations are permitted to be nondeterministic, a possibility we will heavily take advantage of in specifications but not synthesizable implementations.

3.2 Primitive Modules

In general, a Fjfj design is a tree of modules, bottoming out in primitive modules, whose existence we essentially axiomatize. That is, while a synthesizable module is a syntax tree in a language to be defined, a primitive module is represented by its specification, directly as the three-tuple introduced above. A register module has a simple specification, where the state is just a value of the datatype being stored. Another common primitive module is an unbounded queue, which implements the same interface as our queue example from earlier but with simpler internals. For instance, its deq method is axiomatized as follows.

```
Definition deq _arg st newst := ∃ (h : N) (t : list N),
    st = #(cons h t) /\ newst = #t.
```

The # operator injects into a universal "dynamic" type (a pair of a tagged type and a value) which the framework adopts for module-state representation to avoid unwieldy Coq bookkeeping with dependent types. Note that the method cannot be called if the internal state is not of the form

`cons h t`. Therefore, the relation is partial and implies a guard: one can only dequeue from this module if there is an element in it. Fjfj allows a designer to use the full power of Coq's higher-order logic to define the semantics of primitive modules.

3.3 Restrictions and Subtleties

As we outlined earlier, delaying the effects of the action methods of the submodules to the end of each action requires us to clarify the thorny question of the semantics of calling several actions in a rule, e.g. the case where write is called twice on the same register, or the case where one both enqueues and dequeues from a queue. While there are several ways to define the semantics in a way that is both sound and realizable in circuits, we take the opinionated stand to rule out such issues by simply forbidding calling two action methods of the same module in a single rule. That is, the semantics detect dynamically when a rule tries to call two action methods of the same submodule and we abort the rule. Hence, the following rules do nothing.

```
Definition aborts1 := (rule (begin {write r 3} {write s 0} {write s 1})).
Definition aborts2 := (rule (begin {enq queue 1} {deq queue})).
```

This restriction does not prevent a program from calling two action methods of *different* modules.

The partial definition of the relations that constitute the semantics of a module is important to keep in mind: it highlights that the domain of the relation is encoding the abort semantics. Every rule and action method is a partial transition relation: for example in our first complete module `mkPipeline`, when the `mid` queue is empty, it is impossible to dequeue from it. Hence, the `do_g` rule of our pipeline module cannot fire when `mid` is empty.

4 Evaluating Language Restrictions

Before moving on to more formalism, we pause to justify our decision to restrict action-method calls to one per receiver module per logical cycle. We performed a corpus study to evaluate how widely followed this convention is in Bluespec code.

We implemented a compiler pass for the open-source Bluespec compiler[1] that analyzes method-invocation patterns and checks for violations of our language restrictions. Specifically, the pass identifies cases where Bluespec code would not be directly translatable to Fjfj. Our analysis operates at the granularity of synthesized modules, examining method calls within both rules and methods. Table 1 summarizes the results.

4.1 Methodology

Our goal was to evaluate how frequently these restrictions are violated in large-scale Bluespec developments that are currently beyond the reach of verification. We analyzed five Bluespec designs of varying complexity: (1) a processor design from a Bluespec course, (2) a switch implementation described later, (3) a simple pipelined superscalar processor, (4) a large superscalar, multicore, out-of-order processor capable of booting Linux [Zhang et al. 2018], and (5) a fairly large accelerator: an H.264 video decoder.

The total codebase spans approximately 100K lines of code, with the out-of-order processor comprising the majority. Our analysis operates on the elaborated design, examining all the modules synthesized by the compiler. The absolute number of rules and methods should be taken with a grain of salt, as earlier phases turn parameterized modules into specialized copies.

4.2 Analysis of Violations

The most common kind of violation is a false alarm: warnings due to the conservativity of our analysis. These are calls to the same action methods that are in two disjoint paths of branches.

[1] https://github.com/B-Lang-org/bsc

Table 1. Restriction violations across different designs

Design	Rules	Methods	Rule Violations	Method Violations
Course Processor	73	12	0	0
Switch	123	212	0	0
Simple Superscalar	126	12	6	0
OoO Processor	1,936	1,212	22	3
H.264 Decoder	100	158	5	0

We found the first violation in a Bluespec implementation of superscalar queues, implemented in an unsafe fragment of Bluespec (using direct wires). In Fjfj, such modules would be reasonably hand-considered as primitive modules.

More interestingly, all other violations are at uses of these superscalar queues. These violations involve multiple enqueue methods (e.g., `enq1`, `enq2`) intended for direct calls from one rule: for example, the `decode` rule might decode one or two instructions each time the rule is called (hence calling potentially both `enq1`, `enq2` of the execute stage). Such designs clearly violate our restrictions.

We briefly confirmed that two different refactorings can bring the relevant code into compliance with our restriction. We can refactor the design by merging multiple enqueue methods into a single method with optional parameters (e.g., `enqEither1Or2(x, Maybe y)`). Alternatively, maybe a more modular and Bluespec-friendly pattern would be to split such a `decode` rule into two rules `decode_1` and `decode_2`. For this small superscalar processor, we experimented with both alternatives with no noticeable impact on quality of the circuit generated.

Finally, there is one violation in the H.264 decoder due to a mysterious zero-bit module called *sched_hack*. The module seems to be used as a ghost module to communicate scheduling constraints to the Bluespec scheduler implicitly. We can achieve the same effect without using this hack, using pragmas.

In summary, we conclude that the method-call restriction is already quite commonly followed in Bluespec code "in the wild," and simple refactorings suffice to remedy most violations.

5 Formal Semantics

The goal of the formal semantics is to specify what value gets returned by each value method (and when value methods can be called) and what are the state updates caused by the rules and action methods. That is, we want to define the transition relation of rules and methods, from pieces of Fjfj syntax (presented in Figure 1) as a composition of the relations forming the semantics of the submodules (presented in Figure 2 and Figure 3).

We write the judgment $(\alpha_1, \Gamma_1, \beta_1) \dashv[\text{action\_expr}]\mapsto (\alpha_2, \Gamma_2, \beta_2)$ to say that action_expr transforms an initial triple, of requested action calls on submodules, environment, and requested value calls on submodules, into a new triple. Note that action expressions do not return values; they only have side effects on the submodules. A Γ value records local variables. We will often say that expressions do not have side effects because they cannot update submodules. Note however from the previous section that both actions and expressions are allowed to update the local environment. The reason is that in rule-based languages, these local variables are not actually updating any state; they correspond to giving names to intermediate wires.

An α value records requested calls to action methods in the form of a partial function from submodule identifiers to action methods requested on them. A β value is structured differently due to support for multiple value-method calls to a module per logical cycle: it is simply a set of pairs of submodule identifiers and value methods called on them.

We also define a similar judgment for a value expression, which instead of returning an updated α data structure (which would not make sense, as value expressions do not request actions on submodules), returns a value. We write $(\Gamma_1, \beta_1) \dashv[\text{value\_expr}]\rightarrow_{\text{value}} (\Gamma_2, \beta_2, ret)$ to indicate that the value expression takes a pair of an environment and a set of *value* method calls performed so far, relating them to a new environment, a new set of value-method calls performed, and a value returned by the value expression: *ret*.

Actually, both judgments must be parameterized further by the states *st* of the submodules of the current module. It is usually clear from context which state we run expressions in, but where needed for clarification, we will add $st \vdash$ before an assertion.

We now explain one of the rules that exercises the most interesting design choices.

$$\frac{(\Gamma, \beta) \dashv[e]\rightarrow_{\text{value}} (\Gamma', \beta', v_{\text{arg}}) \qquad (\text{inst}, \text{vm}) \notin \beta' \qquad (st[\text{inst}], v_{\text{arg}}, ret) \in [[\text{inst}]].\text{vmethods}[\text{vm}]}{(\Gamma, \beta) \dashv[\{\text{vm inst } e\}]\rightarrow_{\text{value}} (\Gamma', \beta' \cup \{(\text{inst}, \text{vm})\}, ret)}\ \textsc{Call}$$

A call to a value method (vm) of a submodule (inst) is possible only if this precise value method was not already called so far (it is not in β'). Moreover, we need to look at the semantics of the corresponding value methods of the submodule *e.g.* [[inst]].vmethods[vm]. By definition, this object is a set of triples, such that $(sub_{st}, arg, ret) \in [[\text{inst}]].\text{vmethods}[\text{vm}]$ whenever calling the value method with argument *arg* when the submodule has state sub_{st} returns the value *ret*. Note, as was pointed out in subsection 3.1, that if a pair of a state and an argument is not in the domain of the relation, then the method cannot be called.

Derived semantics of rules. A transition from state *st* to state *st'* by rule *R*, noted $st \rightarrow_R st'$, is defined as follows:

$$\begin{aligned}&\exists \alpha.\ st \vdash (\emptyset, \emptyset, \emptyset) \dashv[R]\rightarrow (\alpha, *, *) \wedge \\ &\quad (\forall\ \text{inst} \in \text{dom}(\alpha).\ (st.\text{inst}, \alpha(\text{inst}).arg, st'.\text{inst}) \in [[\text{inst}]].\text{amethods}[\alpha(\text{inst}).\text{method}]) \wedge \\ &\quad (\forall\ \text{inst} \in \text{dom}(st) \setminus \text{dom}(\alpha).\ st.\text{inst} = st'.\text{inst})\end{aligned} \tag{1}$$

In other words, a new state is a valid transition when *every new submodule state either corresponds to a transition of an action method being called on the submodule if an action is called on the submodule, or the submodule is left unchanged if no action was called on the submodule.*

We then define the semantics of a rule *R* as:

$$[[R]] := \{(st, st') | st \rightarrow_R st'\}$$

This judgment can be derived only if all the value methods of the submodules run without aborting.

Derived semantics of action methods. Very similarly to the semantics of a rule, we can use the same judgment rules to define the transitions for an action method $st \rightarrow_{am,arg} st'$ by:

$$\begin{aligned}&\exists \alpha.\ st \vdash (\emptyset, \{\text{arg} : arg\}, \emptyset) \dashv[am]\rightarrow (\alpha, *, *) \wedge \\ &\quad (\forall\ \text{inst} \in \text{dom}(\alpha).\ (st.\text{inst}, \alpha(\text{inst}).arg, st'.\text{inst}) \in [[\text{inst}]].\text{amethods}[\alpha(\text{inst}).\text{method}]) \wedge \\ &\quad (\forall\ \text{inst} \in \text{dom}(st) \setminus \text{dom}(\alpha).\ st.\text{inst} = st'.\text{inst})\end{aligned} \tag{2}$$

And as expected we pose the semantics of an action method as follows:

$$[[\text{am}]] := \{(st, arg, st') |\ st \rightarrow_{\text{am},arg} st'\}$$

Derived semantics of value methods. Finally, the value methods (which only return values and do not need to update state) return values given by the already-explained judgment $* \dashv[*]\rightarrow_{\text{value}} *$:

$$st \rightarrow_{vm,arg} ret := (\emptyset, \{\text{arg} : arg\}, \emptyset) \dashv[vm]\rightarrow_{\text{value}} (*, *, ret)$$

value_expr	::=	v	Variable
		| c	Constant
		| (f value_expr ... value_expr)	Pure combinational
		| (set v value_expr)	Set variable
		| (if value_expr value_expr value_expr)	
		| abort	
		| {value_name instance value_expr}	Pure method call
action_expr	::=	(if value_expr action_expr action_expr)	
		| value_expr	
		| (begin action_expr ... action_expr)	
		| pass	
		| {action_name instance value_expr}	Side-effecting method call
base	::=	instance := module	Submodule
		| action_name := (action_method (v) action_expr)	
		| value_name := (value_method (v) value_expr)	
		| (rule action_expr)	
module	::=	base* | axiomatic specification	

Fig. 1. Syntax of Fjfj

We define the semantics of value methods as expected:

$$[[\text{vm}]] := \{(st, arg, ret) |\ st \rightarrow_{\text{vm},arg} ret\}$$

We use the notation $x \rightarrow_M^* x'$ to signify that there exists a sequence of *rules* of M to go from state x to x'.

Definition of a New Fjfj Module

Given:

(1) $S = \{\mathbb{m}_i | i \in [0, k)\}$, a family of submodule denotations (each composed of an internal state type and three relations, see subsection 3.1)
(2) $R = \{r_i | i \in [0, n_{rules})\}$ action expressions (pieces of syntax)
(3) $V = \{vm_i | i \in [0, n_{vmethods})\}$ value expressions (pieces of syntax)
(4) $A = \{am_i | i \in [0, n_{amethods})\}$ action expressions (pieces of syntax).

We define the semantics of a nonprimitive module (S, R, V, A) as follow:

$$\begin{gathered}[[(S, R, V, A)]] = \{ \\ \text{Internal type} := \mathbb{m}_0.T \times \cdots \times \mathbb{m}_k.T; \\ \text{Rule relations} := \{\mathbb{r}_i | \mathbb{r}_i = [[r_i]]\} \cup \bigcup_l \text{lift}(\mathbb{m}_l.rules); \\ \text{Action-method relations} := \{\mathbb{am}_i | \mathbb{am}_i = [[am_i]]\}; \\ \text{Value-method relations} := \{\mathbb{vm}_i | \mathbb{vm}_i = [[vm_i]]\}\}\end{gathered}$$

The lift operation flattens the rules of submodules and promotes them to rules of the parent module. We found that this encoding worked better than retaining hierarchy semantically.

Note that we do not directly use the *syntax* of the submodules (which might not even exist if the submodules are primitive modules) to define the semantics of the parent module.

$$\frac{}{(\Gamma, v) \dashv[\text{arg}]\vdash_{\text{value}} (\Gamma, v, \Gamma[arg])}\textsc{Arg} \qquad \frac{}{(\Gamma, \beta) \dashv[\text{v}]\vdash_{\text{value}} (\Gamma, \beta, \Gamma[\text{v}])}\textsc{Var}$$

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', r)}{(\Gamma, \beta) \dashv[(\text{set x } e)]\vdash_{\text{value}} (\Gamma'[\text{x } := r], \beta', r)}\textsc{SetVar}$$

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', 1) \qquad (\Gamma', \beta') \dashv[t]\vdash_{\text{value}} (\Gamma'', \beta'', r)}{(\Gamma, \beta) \dashv[(\text{if } e\ t\ f)]\vdash_{\text{value}} (\Gamma'', \beta'', r)}\textsc{IfVT}$$

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', 0) \qquad (\Gamma', \beta') \dashv[f]\vdash_{\text{value}} (\Gamma'', \beta'', r)}{(\Gamma, \beta) \dashv[(\text{if } e\ t\ f)]\vdash_{\text{value}} (\Gamma'', \beta'', r)}\textsc{IfVF}$$

$$\frac{(\Gamma, \beta) \dashv[e_1]\vdash_{\text{value}} (\Gamma', \beta', r_1) \qquad (\Gamma', \beta') \dashv[e_2]\vdash_{\text{value}} (\Gamma'', \beta'', r_2)}{(\Gamma, \beta) \dashv[(f\ e_1\ e_2)]\vdash_{\text{value}} (\Gamma'', \beta'', f(r_1, r_2))}\textsc{PureF}$$

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', v_{arg}) \qquad \text{inst.vm} \notin \beta' \qquad (st[\text{inst}], v_{arg}, r) \in [[\text{inst}]].\text{vmethods}[\text{vm}]}{(\Gamma, \beta) \dashv[\{\text{vm inst } e\}]\vdash_{\text{value}} (\Gamma', \beta' \cup \{\text{inst.vm}\}, r)}\textsc{Call}$$

Fig. 2. Judgment rules for $* \dashv[*]\vdash_{\text{value}} *$. The relation tracks the effects of value expressions on the environment, on the data structure tracking the value methods used by the expression, and the returned value.

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', \_)}{(\alpha, \Gamma, \beta) \dashv[e]\vdash (\alpha, \Gamma', \beta')}\textsc{Expr} \qquad \frac{}{(\alpha, \Gamma, \beta) \dashv[\text{pass}]\vdash (\alpha, \Gamma, \beta)}\textsc{pass}$$

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', 1) \qquad (\alpha, \Gamma', \beta') \dashv[t]\vdash (\alpha', \Gamma'', \beta'')}{(\alpha, \Gamma, \beta) \dashv[(\text{if } e\ t\ f)]\vdash (\alpha', \Gamma'', \beta'')}\textsc{IfT}$$

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', 0) \qquad (\alpha, \Gamma', \beta') \dashv[f]\vdash (\alpha', \Gamma'', \beta'')}{(\alpha, \Gamma, \beta) \dashv[(\text{if } e\ t\ f)]\vdash (\alpha', \Gamma'', \beta'')}\textsc{IfF}$$

$$\frac{(\Gamma, \beta) \dashv[e]\vdash_{\text{value}} (\Gamma', \beta', r) \qquad \text{inst} \notin Dom(\alpha) \qquad (st[\text{inst}], r, \_) \in [[\text{inst}]].\text{amethods}[\text{am}]}{(\alpha, \Gamma, \beta) \dashv[\{\text{am inst } e\}]\vdash (\alpha[\text{inst} \mapsto \text{am}], \Gamma', \beta')}\textsc{CallAct}$$

Fig. 3. Judgment rules for $* \dashv[*]\vdash *$ tracking the effect of action expressions on the environment, the data structure tracking the value methods used, and the one tracking the action methods requested.

6 Simulations as Refinement

Now we are ready to adapt classic correctness notions and proof principles for transition systems. The different limitations and hierarchical structure that we have woven into the language design will pay off in greater simplicity, compared to how these details worked out in RTL-level reasoning or in past work on verifying Bluespec-style designs [Choi et al. 2017].

State One-Step-Simulation. Let M_I and M_S be two Fjfj modules which expose the same *interface*, *i.e.*, they define the same value methods and action methods. Let their corresponding semantics be $[[M_I]]$, $[[M_S]]$, respectively.

Let i and s be states for the implementation and specification modules respectively. We say that the module M_I in state i *one-step-refines* the module M_S in state s (or that s *one-step-simulates* i), when:

(1) For every value method vm,

$$\forall\ arg, ret.\ (i, arg, ret) \in [[M_I]].\text{vmethods}[vm] \Rightarrow (s, arg, ret) \in [[M_S]].\text{vmethods}[vm]$$

(2) For every action method am,

$$\forall\ arg, i'.\ (i, arg, i') \in [[M_I]].\text{amethods}[am] \Rightarrow \exists s'.(s, arg, s') \in [[M_S]].\text{amethods}[am]$$

In such a situation, we write $i\ {}_{M_I}\!\prec_{M_S} s$. Intuitively it says that if a module M_I is in state i, nothing that it can do immediately cannot be done by a module M_S in state s. We will often omit M_I and M_S in the notation and write $i \prec s$ when they are obvious from the context.

Module Simulation. We say that M_S simulates M_I (or M_I refines M_S) along the relation $\phi \subset State_{M_I} \times State_{M_S}$, noted $M_I \sqsubseteq_\phi M_S$, when:

(1) For every initial state i of the implementation, there exists an initial state s of the specification, such that $i \prec s \wedge \phi\ i\ s$.
(2) For every i_1 and s_1 such that $i_1 \prec s_1 \wedge \phi\ i_1\ s_1$, and for every r and i_2 such that $(i_1, i_2) \in [[M_I]].\text{rules}[r]$, there exists a sequence of rules (and corresponding intermediate states) $\mathrm{r}_1, \ldots, \mathrm{r}_k,\ s_2 \ldots s_k$, s.t.

$$(\forall j \in [1, k).\ (s_j, s_{j+1}) \in [[M_S]].\text{rules}[r_j]) \wedge i_2 \prec s_k \wedge \phi\ i_2\ s_k$$

(3) For every i_1 and s_1 such that $i_1 \prec s_1 \wedge \phi\ i_1\ s_1$, and for every am, arg and i_2 such that $(i_1, arg, i_2) \in [[M_I]].\text{amethods}[am]$, there is s_2 such that $(s_1, arg, s_2) \in [[M_S]].\text{amethods}[am]$, and there exists a sequence of rules (and corresponding intermediate states) $\mathrm{r}_2, \ldots, \mathrm{r}_k,\ s_2 \ldots s_k$, such that,

$$(\forall j \in [2, k).\ (s_j, s_{j+1}) \in [[M_S]].\text{rules}[r_j]) \wedge i_2 \prec s_k \wedge \phi\ i_2\ s_k$$

Finally we note $M_I \sqsubseteq M_S$ whenever $\exists \phi.\ M_I \sqsubseteq_\phi M_S$.

This last sequence of definitions is notable for which complexities it *does not* require the proof author to confront. Recall that Kami [Choi et al. 2017] forced module specifications in terms of all sets of incoming method calls that a module might face in a single atomic step, which could in general require handling exponentially many cases, w.r.t. a module's method count. Hence, a module in Kami could not be characterized by giving a transition relation for each method individually. In contrast, thanks to the restrictions in our language, each method may be characterized and proved individually, as we are used to from e.g. frameworks for verifying object-oriented programs. This simplification allows us to scale our design and verification methodology to significantly larger systems (see subsection 8.1 and subsection 8.2).

6.1 Example: Simple Queue Simulation

We now verify a one-element queue implementation against the idealized spec given earlier, sketching the simulation relation ϕ. The full proof, which might be the simplest useful proof of a refinement, is also mechanized in Coq.

Theorem 6.1 (Queue refinement). *mkFifo1* $\sqsubseteq$ *mkFifoSpec*.

Proof. In this case we use the following natural ϕ.

$$\phi\ i\ s := \exists\ (v\ d : N)(l_a : list\ N).\ i = \#(v, d) \wedge s = \#l_a \wedge v \in \{0, 1\} \wedge$$
$$(\forall a.\ l_a = [a] \iff v = 1 \wedge d = a) \wedge (l_a = [] \iff v = 0).$$

□

Remark – the implementation queue is a strict refinement of the specification queue: The two queues are not *bisimilar* (or equivalent). Indeed, some behaviors exhibited by the specification queue will never occur in the 1-element implementation queue. An example is the behavior $[enq(1); enq(2)]$, which would require the queue to be able store at least two elements.

The usefulness of this refinement notion comes in sound verification of a module where we consider its child modules *only in terms of their specifications* (proved with the same notion of refinement). The fact that once a refinement is proven that such a substitution becomes valid is justified by the following *refinement theorem.*

6.2 Refinement Theorem

Let $(\mathbb{m}_i | i \in [1, k])$ be a family of k modules, and let $\mathbb{m}_0$ and $\mathbb{m}'_0$ be two modules defining the same value methods and action methods and such that $\mathbb{m}_0 \sqsubseteq \mathbb{m}'_0$.

Let (S_I, R, A, V) be an Fjfj module, composed of submodules $S_I = (\mathbb{m}_0, \mathbb{m}_1, \ldots, \mathbb{m}_k)$ and pieces of syntax for rules $R = \{r_i | i \in [0, n_{rules})\}$, action methods $A = \{am_i | i \in [0, n_{amethods})\}$ and value methods $V = \{vm_i | i \in [0, n_{vmethods})\}$. Let (S_S, R, A, V) be the same module when we substitute the first submodule by its specification: $S_S = (\mathbb{m}'_0, \mathbb{m}_1, \ldots, \mathbb{m}_k)$. The refinement theorem guarantees:

$$[\![(S_I, R, A, V)]\!] \sqsubseteq [\![(S_S, R, A, V)]\!]$$

Note that here, without loss of generality, we refined the first submodule of the system. We could similarly have refined a module in any position. The proof of this refinement theorem has also been mechanized in Coq and constitutes one of the main workhorses of our framework.

6.3 From Refinement to Trace Inclusion

We sometimes want a top-level theorem statement about inclusion of traces of behaviors instead of existence of simulation relations. We now adapt and present a well-known result from Baier and Katoen [2008] to Fjfj, proving that simulation implies the inclusion of behaviors.

We define a *method interaction* with a module M as either: (1) an action-method step indicating we succesfully called a given action method: am(arg), or (2) a value-method step coupled with the return value observed: vm(arg) $\longrightarrow$ ret.

We define inductively the predicate behavior $M\ l\ i\ e$ declaring that a list l of method interactions is a behavior starting from the initial state i of the module M, ending in state e, if there exists a choice of rules interleaved with these method calls, which support this trace. For example one can easily prove:

$$\text{behavior } mkFifoSpec\ [enq(1); enq(2); first() \longrightarrow 1; deq()]\ []\ [2]$$

Fjfj provides the following theorem (also mechanized in Coq) to get from refinement to trace inclusion:

Theorem 6.2 (Trace inclusion). *If $M_I \sqsubseteq M_S$ then:*

$$\forall l, e.\ behavior\ M_I\ l\ init_i\ e \Rightarrow \exists e'.\ behavior\ M_S\ l\ init_s\ e'$$

6.4 Typical Usage of Refinement

We will now illustrate some typical uses and advantages of refinement. Our usual pattern is to introduce intermediate specifications in our proofs, to replace low-level, synthesizable modules by clean data structures on which it is easy to write invariants to connect with the top-level specification.

Canonicity of representation. Another aspect in which picking the right data structure for our intermediate specification matters is what we call *canonicity*. Consider two implementations of a 4-element queue. One is built with 4 registers, an enqueue pointer, a dequeue pointer, and a register indicating an empty queue. We can call this version the *circular-buffer version*. The other version is simply using a list of length at most 4, which we can call the *list version*.

The list version has the property of *canonicity*: reasoning about an empty queue, we only need to consider a single case: the empty list. In contrast, the circular buffer has 4 different states that represent the empty case (equal enqueue pointer and dequeue pointer, and empty register is set).

Thus, the proof will either need to cover more cases explicitly or work modulo a custom equivalence relation. Those issues disappear when we do a proof in the list version: there are no multiple state representations of the same conceptual state. We say the data structure is canonical.

Generally, we say that a module M has the canonicity property if there exists no nontrivial simulation of M by itself, i.e. if ϕ witnesses a simulation of M by itself, then $\phi\ i\ s := i = s$.

We think of the refinement theorem as the systematic tool to work modulo the right notion of substitution, with a foundational justification for the substitution. And we often use the refinement theorem to replace a noncanonical and synthesizable implementation with a canonical specification, which requires fewer cases to consider in proofs.

We will always favor data structures that avoid requiring that we state invariants about symmetric cases; preferring to bake those equivalent states directly into a custom data structure will make proofs easier and smaller. Our use of canonical data structures to replace submodules can be seen as an alternative to the more implicit partial-order-reduction techniques used in traditional model-checking, like Flanagan and Godefroid [2005] and Peled [1993].

Simulation vs. Hoare logic. This style of proof has much in common with the earliest methods of verification for data abstraction [Hoare 1975]. A simulation relation is very similar to an abstraction relation used to explain why a program module implements an abstract data type. The difference is that classic Hoare-logic methods of this type effectively force the "reference implementation" to be written in math/logic within a specification. The refinement style can support similar affordances while offering greater convenience for proofs by stepwise refinement, such that both "implementation" and "specification" can be coded in the same compiler-ready language, though with the chance to intersperse purely logical constructs where convenient.

7 Vectors of Submodules

One common design pattern in hardware is including several clones of particular submodules, for example a collection of processing elements or a collection of queues. In this section we elaborate on our strategy to describe and verify this kind of design pattern.

Consider a parametrized butterfly crossbar, whose job is to route messages on a number of input ports to the appropriate output ports. This crossbar will serve as our running example of verifying parametrized code containing a variable number of subunits: the inputs and outputs are each collectively seen as vectors.

One natural approach would be to consider that submodule duplication is a construction in the metalanguage Coq, where a functional "macro" runs to generate a more verbose, redundant hardware design. However, we found that it streamlined proofs to axiomatize vectors as a kind of higher-order primitive module.

We observe that, in hardware, we often do one of two things with vectors:

(1) Call one method of *one* of the submodules of the vector (the *select* style).
(2) Call one method of *all* the submodules of the vector (the *map* style).

From this observation, in Fjfj, a Vector is a construction to build *one primitive module* – the vector – which contains four methods: the *map* value and action methods, and the *select* value and action methods. The former two methods take an input in the form of a vector of bits, with each corresponding to one element in the vector of modules. The latter two methods have a variadic number of parameters: a first parameter indicates which method of the submodules is actually called, and the other parameter is passed to the submethod.

This construction allows us to represent the full vector as a single module, without requiring special new syntax support for it. While it does not support arbitrary usage patterns of vectors of modules, we found it to be expressive enough for now, while also promoting patterns that map properly to modularly compiled RTL. It also can easily be extended, to support more vector-level methods, for example calling the same method for a subset of the modules, or if we wanted to be even more general, a general fold expressing for each submodule of the vector which method we want to call (if any) with which parameters.

8 Evaluation

We evaluated Fjfj on three hardware designs that we verified: a parametrized butterfly crossbar, a processor, and a network switch.

First, a word is called for on the toolchain used in these studies and the project overall. Unlike other projects that have produced verified compilers for Bluespec-style languages (e.g. Koika [Bourgeat et al. 2020]), our focus is on modular proof of functional correctness. Hence, it is convenient to reuse the open-source Bluespec compiler, which called for writing all case studies first in the established Bluespec language. After sufficient analysis and testing of the FPGA-compatible RTL code that resulted, we then hand-translated the Bluespec files to Fjfj designs (embedded in Coq source files).

This flow represents an expedient evaluation strategy, not a suggested long-term workflow. Though the translation from Bluespec to Fjfj is extremely direct, we might also automate it, then viewing Fjfj as an intermediate representation for verification. Indeed, Fjfj today involves modeling choices for simplicity, like passing all method arguments and return values as natural numbers, forcing insertion of modulus operations at the right points; hence marking Fjfj as unsynthesizable. Another approach is to create a *verified* translation into such an intermediate language from a surface language with its own verified compiler (a parallel path to RTL rather than functional-correctness proof). In any case, these alternatives are orthogonal to the questions of this project.

8.1 Pipelined Processor

While several previous projects have demonstrated proofs of processors in verification systems, to our knowledge each proof has tackled a processor with a bounded (and small) number of in-flight instructions. In this section, we present the first proof of a pipelined processor valid for an arbitrary number of in-flight instructions. This choice showcases a machine that would be especially hard to model-check using standard methodology and so where a proof assistant can shine.

We focus on architectural transformation and do not look at Boolean equivalences. For the latter, we reuse a standard uninterpreted-functions technique where paired implementations and specifications will use the same uninterpreted functions for ALU execution, decoding, etc. (axiomatized in Coq). As a result, our proof can focus on the pipelining and other architectural features of interest without intermingling orthogonal proof challenges.

One reason proofs of correctness of hardware are difficult and time-consuming is because they are usually not modular; we often need global invariants of the system that mention all the different submodules of the design simultaneously and ensure constraints between their states. The invariants need to be rewritten significantly whenever the design changes. As an example, one might be interested in replacing the execute-to-writeback (e2w) queue by a two-element queue

in the processor verified in previous work [Choi et al. 2017]. This design change might be to hide a two-cycle latency to data memory for example. With previous proof techniques, the invariant explicitly listed all possible cases of in-flight instructions: there is one instruction in the execute stage, or there is none. Hence, this change to the design would require significantly extending the invariant: it would lead to an invariant having about double the size the original one, requiring significant manual work. At the same time, architecturally speaking, such a change should be inconsequential. In contrast to this kind of proof, we want our processor designs to have submodules independently specified and verified, allowing variations in components (e.g., frontend, backend, queues, register files, scoreboard, memories) without affecting most of the proof.

The lack of modularity in previous proofs likely stemmed primarily from the lack of correct modular specifications. Indeed, we will show that the intuitive specification of a standard processor system (core + memory) must be *generalized* to encompass behaviors seen in implementations. Generalization, however, is sometimes dangerous because it admits behaviors that are not seen in the original intended specification, and thus it can make the whole design wrong. Therefore, the generalized specification of a submodule must be shown to be safe in its context of use.

Processor, Memory, and System Specification

Before we dive in, remember that the words *refinement* and *correct* now have been given precise formal meanings. *Correctness* refers to the existence of a simulation relation (a refinement) between the specification design and the implementation design, which only talks about the behaviors of the two systems at their interface.

Let us show the need for generalizing specifications using a machine composed of a processor and a memory. The machine interacts with the environment using memory-mapped IO (MMIO), and as such MMIO is the only interface to the system. One may implement such a machine using many microarchitectures such as unpipelined, in-order pipelined, or out-of-order microarchitecture for the processor; and different cache hierarchies, replacement policies, and reordering policies for the memory. Typically, a machine will have multiple simultaneously outstanding memory loads. In contrast, the MMIO requests and responses are typically handled sequentially and nonspeculatively, as both are interactions with the outside world that can have arbitrary side effects, even for MMIO loads. We focus on the processor side of the system and assume that the memory module is kept constant as a first-in-first-out memory specification.

We diagram the specification and implementation systems in Figure 4. While the internal processor-memory interaction varies between implementations, all must exhibit identical I/O behavior for any given program. One might expect to prove refinement modularly: show the pipelined processor implements the one-instruction-at-a-time specification, prove the cached memory implements the memory specification, and combine these proofs using our refinement theorem from subsection 6.2. However, this approach fails because the subsystem-level lemmas generally do not hold. For instance, the pipelined processor does not refine the atomic specification processor: it may emit speculative loads and reorder operations, making it easy to prove that $mkPipelinedProcessor \not\sqsubseteq mkAtomicSpecProcessor$. Memory systems face similar issues. Counterintuitively, while the full-system specification comprises separate processor and memory components, these components are not valid specifications for their respective implementations in isolation. The specification is only valid when considering the complete system as a whole. Loosely speaking, a system of memory plus processor will most of the time verify the following:

$$FullSystem(Memory, ProcessorImpl) \sqsubseteq FullSystem(Memory, SimpleProcessorSpec)$$

while at the same time:

$$ProcessorImpl \not\sqsubseteq SimpleProcessorSpec$$

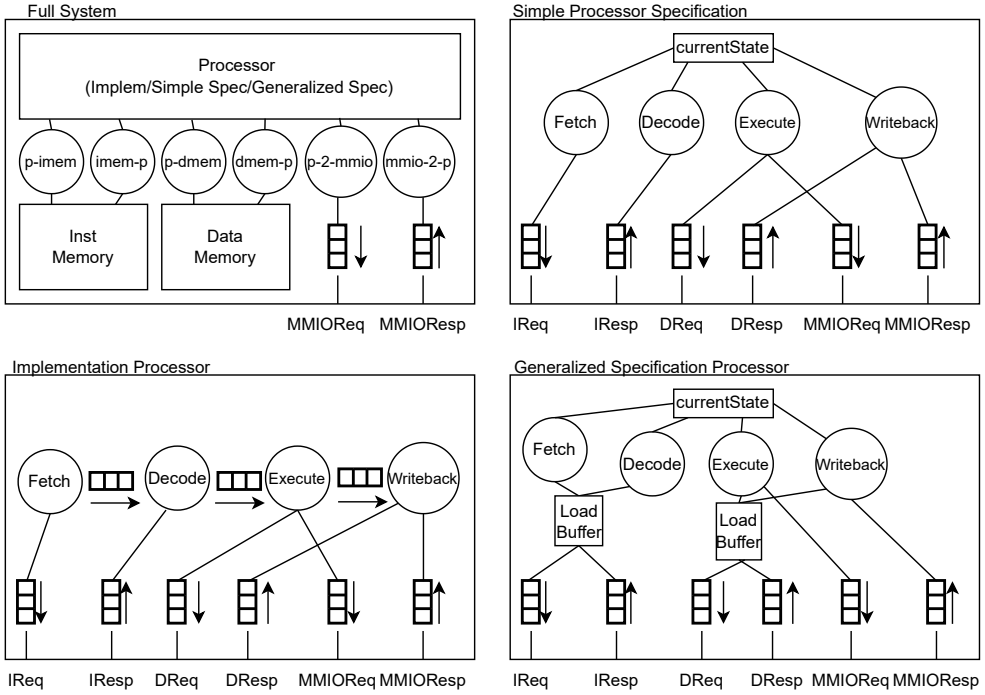


Fig. 4. Architectural sketch of the full-system and processor implementations and specifications. Rectangles delimit module boundaries, circles represent rules, and a plain line always goes from a rule or a method to a method of a submodule indicating a method call. The top-level module simply connects the processor to the memories (instruction and data) and connects the MMIO requests-and-responses queues to the public-facing interface. The processor specification (top right) is a very simple *multicycle* specification: there is exactly one instruction in-flight at any time in this machine, with a register keeping track of which state of execution the instruction is in. The processor implementation (bottom left) is a simple *pipelined* machine: newer instructions are fetched while previous ones are being decoded, executed, etc.

At first glance, we have hit a significant challenge for modular proof.

Generalizing Specifications: Recovering Modularity

We now present a system with a processor and a memory specification compatible with modular refinement, enabling verification of a large family of processors and memory implementations.

Consider the more general specification for our system, sketched in the bottom right of Figure 4. This new specification introduces two *nondeterministic load machines*: the *Data-Load Buffer* and the *Instruction-Load Buffer*. Those two modules are simply machines that emit loads for arbitrary addresses, at arbitrary times, and store the results into local load buffers, to be served if requested from the simple multicycle part of the machine. An intuitive view of this specification is that an actual processor emits both program-generated memory operations and additional loads from speculative execution. Instead of characterizing precisely the shape of loads that can be emitted, this specification indicates that we should *conservatively* think of a processor as a machine that can emit loads to any addresses at any time.

The *Generalized Processor Specification*, augmented with the nondeterministic load machines, is able to fake the speculative loads that made it impossible to use the ISA processor specification alone as a specification for the processor. We can now conclude:

$$ProcessorImpl \sqsubseteq GeneralizedProcessorSpec$$

The refinement map used to support this proof states that flushing the pipelined and speculative machine yields a state directly relatable to the specification. Since the proof accommodates unbounded in-flight instructions, the invariant (on the order of 100 LoC) is expressed in higher-order logic but *remains completely local to the processor*. The main challenge stems from handling unbounded memory requests in a non-instantaneous request-response system where multiple requests may interleave before receiving a desired response. A key challenge with this generalized specification is also ensuring additional loads do not cause system-wide (MMIO-level) issues.

What is a valid generalized specification? Now that we have defined a generalized specification, we have two definitions of full-system specifications: the more-involved one using a generalized specification for the processor and the simpler one using the simple specification for the processor. We say that a generalized specification is valid in its context if the full system using the generalized specification is refined by the full system using the simple specification, i.e.,

$$FullSystem(GeneralizedProcessorSpec, Memory) \sqsubseteq FullSystem(SimpleProcessorSpec, Memory)$$

While both execute instructions sequentially, the proof of this refinement demonstrates that the extra loads in the generalized specification do not affect the MMIO trace. The refinement map maintains identical internal states of the two processors; only memory-related structures differ. For instruction memory, both the response queue and load buffer must reflect instruction-memory contents. The data-memory relation is more complex: flushing outstanding requests in the implementation should yield a memory state matching the specification's memory.

Putting everything together: full-system decomposition. To summarize, we can prove our target full-system refinement, using two proofs of refinement and an application of the refinement theorem:

$$ProcessorImpl \sqsubseteq GeneralizedProcessorSpec$$

By application of the refinement theorem it is lifted to obtain:

$$System(ProcessorImpl, Memory) \sqsubseteq System(GeneralizedProcessorSpec, Memory)$$

And then transitively composed with the proof of validity of the generalized specification:

$$System(GeneralizedProcessorSpec, Memory) \sqsubseteq System(SimpleProcessorSpec, Memory)$$

One could remark that in Choi et al. [2017], the processor was also proven as a module independently of its memory, and so one could wonder how this previous work did not run into the problem of having to generalize the processor specification. The reason is that the verified processor serializes the memory accesses and waits for a response before it sends a new request, i.e. it does not pipeline the memory accesses. With these restrictions, the processor traces are indeed similar to those emitted by a multicycle specification without requiring generalizing the specification. The restriction comes at the cost of forbidding architectural optimizations like pipelining memory accesses, which we were interested in studying. At the same time, they proved a processor in a form that was connected to a sofware stack for an end-to-end proof [Erbsen et al. 2021]. Such formal linking is possible when the processor's specification can be read as an instruction-set-architecture (ISA) formal semantics. In contrast, our proof is parameterized over ISA details (including combinational logic for e.g. arithmetic operations, shared between implementation and specification). Once instantiated, the top-level specification would be in the form needed to link with software proofs.

8.2 Network Switch

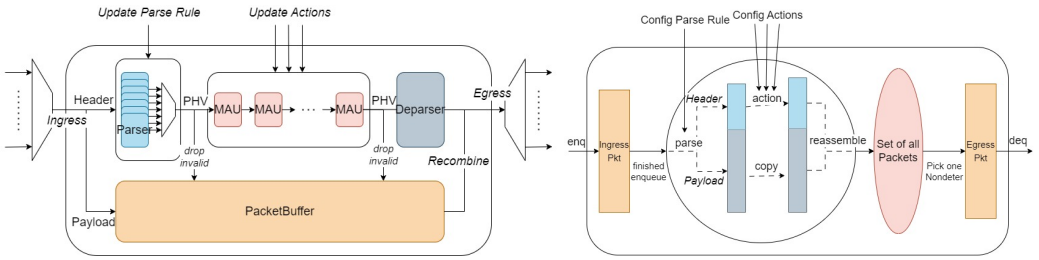


Fig. 5. Network-switch case study: implementation (left) and specification (right).

The most comprehensive case study with Fjfj is VeriSwitch, a verified programmable network switch based on Intel's Tofino PISA architecture [Bosshart et al. 2013]. Though programmable, a Tofino-like architecture differs fundamentally from a standard processor, making it a good target to explore nonprocessor verification. The switch is structured around three main units: a variable-length parallel parsing unit, a match-action unit, and a deparsing unit.

The system makes packet-handling decisions (dropping, rewriting, forwarding) based on configurable policies. Implemented in Bluespec (about 1000 LoC) and translated to Fjfj (about 1000 LoC as well), it achieves 100 Gb/s throughput on FPGAs. Figure 5 shows both VeriSwit's implementation as concurrent Fjfj modules and the specification as a single non-concurrent module that processes each packet atomically. The proof is composed of a hierarchy of refinements through 20 different modules, from high-level specification to implementation.

8.3 Implementation, Specification, and Verification Effort

For each of the three designs, we report the following lines-of-code counts: implementation, specification, intermediate specifications, description of the mappings between different layers of the system, and actual proofs.

Table 2. Lines of code per example

Example	Impl.	Spec.	Interm. Spec.	Ref. Map	Proof	Person-Months
CrossBar	148	71	N/A	70	2956	0.5
Processor	304	205	497	299	7213	3
Network Switch	1157	173	3083	807	16230	9

9 Related Work

Verification of combinational circuits and sequential machines. Combinational circuits can be seen as Boolean functions. Research has produced impressive techniques to solve practical Boolean equivalence problems [Bryant 1986; Moskewicz et al. 2001]. This paper makes no attempt at tackling that problem. Those tools and techniques, effective for combinational circuits, are also at the root of most mainstream formal-verification techniques in use today for verification of sequential machines [Bradley 2011; Burch et al. 1992; McMillan 2003]. While they can search impressively large spaces, these techniques face combinatorial explosion and falter when tackling processors with more than a few in-flight instructions.

Meaning of correctness. Correctness is often defined through a *refinement map*: a predicate relating states of implementation and specification machines. This approach makes correctness proofs dependent on the chosen predicate; an incorrect predicate (e.g., relating all states) renders the proof meaningless. This observation prompted research on what constitutes a good refinement map. For processors, Su et al. [1996] introduced the criterion that flushed implementation states should match the specification's architectural states. While variations in flushing definitions create incomparable correctness notions, our approach uses behavioral simulation for correctness, with flushing merely serving as a proof technique where incorrect definitions simply prevent proof completion.

In the vein of verification with custom predicates, we mention the pioneering works [Brock and Hunt 1997; Jr. 1989] that tackled industrial designs, often in the ACL2 theorem-proving system. We also mention verifications [Berezin et al. 1998; Brady et al. 2011, 2010; Bryant 2018; Burch and Dill 1994; McMillan 1998, 2000; Velev 2023] done in other frameworks (for example, in UCLID5 [Seshia and Subramanyan 2018] or in SMV [McMillan 1993]), using various levels of automation and tackling custom models expressed at various levels of abstraction over synthesizable designs, trading off for complexity of the architectural schemes being proven. Finally, we mention Huang et al. [2024, 2023, 2018]; Xing et al. [2022], who propose a conceptual framework to specify and prove the correctness of accelerators. They show that accelerators can be seen as processors with custom ISAs, and as such one can use traditional formal-verification techniques to prove correctness of accelerators at the granularity of single commands/instructions.

High-level-synthesis (HLS) verification. Another approach to hardware design is to transform programs written in software languages like C, Python, etc. into hardware [Canis et al. 2011; Cong et al. 2011; Gajski 2001; Gupta et al. 2004; Josipović et al. 2018; Mentor [n. d.]; Xilinx [n. d.]]. Recent work [Herklotz et al. 2021] leveraged the use of the software-language semantics to tackle verification tasks (a compiler-verification task). As far as we know, those approaches have not yet been applied for complete formal functional correctness of complex sequential machines.

Bluespec verification. Arvind and Shen [1999]; Dave et al. [2011, 2010]; Vijayaraghavan et al. [2015] first looked at using the rule-based formalism to describe and study the correctness of complex microarchitectural schemes, introducing various forms of refinement (on paper) to characterize the correctness of designs. Kami [Choi et al. 2022, 2017; Vijayaraghavan et al. 2015] formally defined a notion of refinement in Coq which we already discussed in the core of this paper, where the main downside we highlighted was inability to verify each method of a module independently. Wright [2021] has explored model checking for rule-based languages. While Wright [2021] defines a notion of refinement similar to ours, it is not used for processor verification.

10 Conclusion

We introduced Fjfj, a framework for modular formal verification of hardware designs that enables software-style independent verification of module methods. We demonstrated through three case studies that we can tackle deep hierarchical designs relying on stepwise refinement from implementation to specification, introducing intermediate generalized specifications.

Acknowledgments

We dedicate this paper to its posthumous coauthor, Arvind. He was the constant through all of our experiences with the Bluespec language and the philosophy behind it. Together we revisited the fundamentals to systematically question unnecessary sources of complexity in the designs and their proofs. Sharing this process with him was a joy. We hope we have properly imbibed that spirit, to carry on the legacy.

This research was supported by the National Science Foundation under grants CNS-2115587, CCF-2217064, and CCF-2421734. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Thomas Bourgeat was partially supported by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips research project.

References

Arvind and Xiaowei Shen. 1999. Using term rewriting systems to design and verify processors. *IEEE Micro* 19, 3 (1999), 36–46.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387.

Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu. 1998. Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification. In *Formal Methods in Computer-Aided Design*, Ganesh Gopalakrishnan and Phillip Windley (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–386.

Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 99–110. https://doi.org/10.1145/2534169.2486011

Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

B. A. Brady, R. E. Bryant, and S. A. Seshia. 2011. Learning conditional abstractions. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 116–124.

B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. 2010. ATLAS: Automatic Term-level abstraction of RTL designs. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 31–40. https://doi.org/10.1109/MEMCOD.2010.5558624

C. Brock and W.A. Hunt. 1997. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. 31–36. https://doi.org/10.1109/ICCD.1997.628846

Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

Randal E Bryant. 2018. *Formal verification of pipelined Y86-64 microprocessors with UCLID5.* Technical Report. Technical Report CMU-CS-18-122.

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1992. Symbolic Model Checking: 10^20 States and Beyond. *Inf. Comput.* 98, 2 (1992), 142–170. https://doi.org/10.1016/0890-5401(92)90017-A

Jerry R. Burch and David L. Dill. 1994. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, David L. Dill (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–80.

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*. 33–36. https://doi.org/10.1145/1950413.1950423

Joonwon Choi, Adam Chlipala, and Arvind. 2022. Hemiola: A DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocols. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13372)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 317–339. https://doi.org/10.1007/978-3-031-13188-2_16

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on CAD of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

https://doi.org/10.1109/TCAD.2011.2110592

Nirav Dave, Michael Katelman, Myron King, Arvind, and José Meseguer. 2011. Verification of microarchitectural refinements in rule-based systems. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*, Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt (Eds.). IEEE, 61–71. https://doi.org/10.1109/MEMCOD.2011.5970511

Nirav Dave, Man Cheuk Ng, Michael Pellauer, and Arvind. 2010. A design flow based on modular refinement. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*. IEEE Computer Society, 11–20. https://doi.org/10.1109/MEMCOD.2010.5558626

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, 604–619. https://doi.org/10.1145/3453483.3454065

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

Daniel D. Gajski. 2001. SpecC Design Environment. *System Design* (2001), 217–235. https://doi.org/10.1007/978-1-4615-1481-7_5

David J. Greaves. 2019. Further sub-cycle and multi-cycle schedulling support for Bluespec Verilog. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2019, La Jolla, CA, USA, October 9-11, 2019*, Partha S. Roop, Naijun Zhan, Sicun Gao, and Pierluigi Nuzzo (Eds.). ACM, 2:1–2:11. https://doi.org/10.1145/3359986.3361199

Sumit Gupta, Nikil D. Dutt, Rajesh Gupta, and Alexandru Nicolau. 2004. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*. 114–121. https://doi.org/10.1109/DATE.2004.1268836

Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 117 (Oct. 2021), 30 pages. https://doi.org/10.1145/3485494

C. A. R. Hoare. 1975. Proof of correctness of data representation. In *Language Hierarchies and Interfaces, International Summer School, Marktoberdorf, Germany, July 23 - August 2, 1975 (Lecture Notes in Computer Science, Vol. 46)*, Friedrich L. Bauer and Klaus Samelson (Eds.). Springer, 183–193. https://doi.org/10.1007/3-540-07994-7_54

Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Gus Henry Smith, Thierry Tambe, Akash Gaonkar, Vishal Canumalla, Andrew Cheung, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. 2024. Application-level Validation of Accelerator Designs Using a Formal Software/Hardware Interface. *ACM Trans. Design Autom. Electr. Syst.* 29, 2 (2024), 35:1–35:25. https://doi.org/10.1145/3639051

Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. 2023. INVITED: Generalizing the ISA to the ILA: A Software/Hardware Interface for Accelerator-rich Platforms. In *60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco, CA, USA, July 9-13, 2023*. IEEE, 1–4. https://doi.org/10.1109/DAC56929.2023.10247894

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 (TODAES)* 24, 1 (2018), 1–24.

Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *Information Processing 83*, Vol. 9. 321–332.

Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/3174243.3174264

Warren A. Hunt Jr. 1989. Microprocessor Design Verification. *J. Autom. Reason.* 5, 4 (1989), 429–460. https://doi.org/10.1007/BF00243132

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *POPL '15* (Mumbai, India). 637–650.

K. M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 06 (Nov. 2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

Kenneth L. McMillan. 1993. *The SMV System*. Springer US, Boston, MA, 61–85. https://doi.org/10.1007/978-1-4615-3190-6_4

Kenneth L McMillan. 1998. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *International Conference on Computer Aided Verification*. Springer, 110–121.

Kenneth L. McMillan. 2000. A methodology for hardware verification using compositional model checking. *Science of Computer Programming* 37, 1-3 (2000), 279–309.

Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

Mentor. [n. d.]. ModelSim. https://www.mentor.com/products/fpga/verification-simulation/modelsim/.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 530–535. https://doi.org/10.1145/378239.379017

R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

Doron A. Peled. 1993. All from One, One for All: on Model Checking Using Representatives. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 697)*, Costas Courcoubetis (Ed.). Springer, 409–423. https://doi.org/10.1007/3-540-56922-7_34

Sanjit A. Seshia and Pramod Subramanyan. 2018. UCLID5: Integrating Modeling, Verification, Synthesis and Learning. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 1–10. https://doi.org/10.1109/MEMCOD.2018.8556946

Jeffrey X Su, David L Dill, and Clark W Barrett. 1996. Automatic generation of invariants in processor verification. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 377–388.

Miroslav N. Velev. 2023. Automatic Formal Verification of RISC-V Pipelined Microprocessors with Fault Tolerance by Spatial Redundancy at a High Level of Abstraction. In *IFM 2023: 18th International Conference, IFM 2023, Leiden, The Netherlands, November 13–15, 2023, Proceedings* (Leiden, The Netherlands). Springer-Verlag, Berlin, Heidelberg, 193–213. https://doi.org/10.1007/978-3-031-47705-8_11

Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 109–127. https://doi.org/10.1007/978-3-319-21668-3_7

Andrew Wright. 2021. *Modular SMT-Based Verification of Rule-Based Hardware Designs.* Ph. D. Dissertation. Massachusetts Institute of Technology, USA. https://hdl.handle.net/1721.1/139491

Xilinx. [n. d.]. Vivado HLS. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

Yue Xing, Huaixi Lu, Aarti Gupta, and Sharad Malik. 2022. Compositional Verification Using a Formal Component and Interface Specification. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2022, San Diego, California, USA, 30 October 2022 - 3 November 2022*, Tulika Mitra, Evangeline F. Y. Young, and Jinjun Xiong (Eds.). ACM, 72:1–72:9. https://doi.org/10.1145/3508352.3549341

Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. 2018. Composable Building Blocks to Open up Processor Design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 68–81. https://doi.org/10.1109/MICRO.2018.00015

Received 2024-11-15; accepted 2025-03-06