Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification

JOONWON CHOI*, MURALIDARAN VIJAYARAGHAVAN*, BENJAMIN SHERMAN, ADAM CHLIPALA, and ARVIND, MIT CSAIL, USA

It has become fairly standard in programming-languages research to verify functional programs in proof assistants using induction, algebraic simplification, and rewriting. In this paper, we introduce Kami, a Coq library that uses labeled transition systems to enable similar expressive and modular reasoning for hardware designs expressed in the style of the Bluespec language. We can specify, implement, and verify realistic designs entirely within Coq, ending with automatic extraction into a pipeline that bottoms out in FPGAs. Our methodology has been evaluated in a case study verifying an infinite family of multicore systems, with cache-coherent shared memory and pipelined cores implementing (the base integer subset of) the RISC-V instruction set.

CCS Concepts: • **Hardware → Theorem proving and SAT solving**; *Hardware description languages and compilation*; *High-level and register-transfer level synthesis*;

Additional Key Words and Phrases: formal verification, hardware, proof assistants

ACM Reference Format:
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 (September 2017), 30 pages.
https://doi.org/10.1145/3110268

1 INTRODUCTION

In the face of skepticism about the practical potential of formal methods, the standard response of the specialist is that formal methods already play a widespread and essential role in quality control for the computer-hardware industry. However, researchers with experience *only* on the software side might be surprised at some pervasive limitations of the kinds of formal methods that industry applies to hardware, typically based on model checking and satisfiability solving, where analysis reduces to explicit state-space exploration. For instance:

(1) Verification effort is focused on (relatively) small components of full systems, for instance on the floating-point units of processors. While complete verification of the algorithms implementing floating-point arithmetic requires heroic effort by itself, these individual results are *not composed into full-system theorems.*
(2) Relatively *weak properties* are proved, with considerable abstraction gaps from the natural correctness conditions, even for limited components of full systems. For instance, it is common

*Choi and Vijayaraghavan are primary authors with equal contributions, and their two names are listed alphabetically.

2475-1421/2017/9-ART24
https://doi.org/10.1145/3110268

for engineers to think about specifications and manually come up with invariants to be asserted about circuits, with no formal connection back to specs.

(3) The *scope* of verifiable components is fundamentally limited by the *state-space-explosion problem*. For example, in order to verify cache-coherence protocols with Murphi [Dill et al. 1992], a widely used model checker designed for that exact task, at one point the biggest system that could be verified had only 3 cores interacting with a single-address memory having two potential values. While improvements to algorithms and hardware (for running the analysis) continually increase the feasible system size, there are fundamental limitations of such tools that explore finite state spaces, in the face of exponential growth in state spaces as we add processor cores or memory addresses. One mitigation, rarely applied in industry, is verification of *parameterized* systems, with variables standing for e.g. the number of cores. But adoption of this strategy is mostly limited to *models* of real hardware systems, not connected to synthesis pipelines that generate real silicon.

Consider instead the standard procedure used in the programming-languages research community to verify software programs with proof assistants.

(1) Implement the program in the functional programming language built into the proof assistant.
(2) In a rich higher-order logic, state the most natural correctness theorem for the program.
(3) Prove the theorem using scripts of *tactics* for proof steps at different levels of granularity, saving the user from tedious details while still giving an opportunity to spell out the key insights manually.
(4) Use *extraction* to translate the program to a language like OCaml automatically, and from here use standard development tools to compile and run it.

In this paper, we introduce the *Kami* framework for the Coq proof assistant, which brings this style of development and verification to the world of computer architecture, simultaneously reversing all of the weaknesses enumerated above for most hardware-verification tools. Kami makes it possible to code and verify a fairly realistic processor within Coq and then extract it to run on FPGAs (and almost certainly in silicon, though we have not yet had the opportunity to fabricate a "real" chip). We also emphasize that our use of "verified" is more in the tradition of CompCert [Leroy 2006] or seL4 [Klein et al. 2009] than of the day-to-day practices of verification engineers in the hardware industry: we prove that a system *with any number of shared-memory cores with associated coherent caches* correctly implements the *functional-correctness specification of our instruction set*, with respect to sequential consistency.

We must reconsider each of the steps of the recipe above, so that they apply properly to hardware designs. On one level, the central new challenge is the fundamental *limitations* of hardware circuits, where every variable must have a finite domain and where loops and recursion are not supported directly. On another level, the challenge is to support all of the *opportunities* that circuits provide to realize levels of parallelism far beyond what is feasible in software, where in some sense all the gates on a circuit board are constantly running in parallel. Just as most uses of proof assistants verify functional programs rather than assembly programs, to help the proof author focus on the essential problems, in Kami we code hardware designs at a higher abstraction level than circuits: we formalize a subset of the Bluespec hardware description language, whose fundamental structuring principle is sets of state-change rules that execute atomically, with a scheduler automatically choosing how to interleave them.

Readers may be more familiar with Verilog or other hardware descriptions at the level of so-called register-transfer languages (RTL). RTL programs describe circuits as networks of gates, but without physical-placement details. Concerns of *timing* complicate programming at this level. The designer is often required to know how many clock cycles go by between when a signal flows into a module on one wire and when a related signal comes out on another wire. Bluespec abstracts these issues, much like how a C compiler abstracts details of a call stack. It is also the case that RTL designs make parallelism explicit, which is convenient for maximizing performance, but which creates challenges for correctness reasoning: proofs must consider explicit simultaneous execution of components. Bluespec instead exposes a transaction-based model with the fiction that fragments of code run independently and in sequence, and the commercial Bluespec compiler deals automatically with the challenges (timing and parallelization) that Verilog forces upon the programmer. As we progress through introducing the modular proof techniques at the heart of Kami, it should become clear that essentially none of them would apply in the traditional RTL model. It naturally follows that instead of verifying the circuits produced after synthesis from Bluespec, we verify that a high-level implementation refines a high-level specification, both written in Kami. If we trust (or later verify) the Bluespec compiler to preserve the semantics while translating from Kami programs into RTL circuits, then we are guaranteed that the RTL circuits of the implementation refine the high-level specification[*].

The next challenge is choosing the right kind of correctness theorem to ascribe to the components of a system, so that the individual theorems can be composed in a black-box way into full-system results. Here we follow our previous work [Vijayaraghavan et al. 2015] in adapting ideas from process algebra. We formalize hardware components as *labeled transition systems*, with an interaction-centric operational semantics that helps us reason about each component individually while abstracting over its environment. Our previous work [Vijayaraghavan et al. 2015] suffers from one of the problems we mentioned earlier: it applies only to *models* of real hardware designs, with no path toward extraction-style generation of synthesizable circuits. We previously applied a manual and error-prone process to example hardware designs, formalizing each one directly as an inductive relation in Coq. With Kami, we work instead with explicit program syntax designed to be trivially translatable to real Bluespec syntax, to which we can apply Bluespec's commercial synthesis tools to produce hardware circuits.

The last challenge is to figure out the hardware analogues of the workhorse proof tactics of software verification, like induction, execution-based simplification, and rewriting[†], whose closest equivalents turn out to be simulation arguments, inlining of method definitions, and replacement of one hardware module with another proved to be compatible, respectively.

The rest of the paper proceeds through a running example (Section 2); the syntax, semantics, and correctness notions of Kami (Section 3); more background on Bluespec and its practicality, including the approach to automatic RTL synthesis (Section 4); the key framework lemmas used in modular proofs (Section 5); our case-study multicore system (Section 6); results synthesizing and running it on FPGAs (Section 7); related work (Section 8); and remaining barriers to industrial adoption (Section 9).

Our implementation and benchmarks are available as open source:

https://github.com/mit-plv/kami

2 KAMI BY EXAMPLE

We will introduce the main concepts of the Kami infrastructure via a simple example. Figure 1 represents a Kami module that continuously increments a counter and outputs the value of the counter. It consists of a register `counterReg`, representing the state of the counter, and a *rule* `incrementAndOutput` describing an allowable state transition in a single step of the system, incrementing the counter. All registers take values in finite sets (since they represent hardware state).

[*]Our future-work plans include verifying a suitable compiler for core Bluespec.

[†]For Coq experts: `induction`, `simpl`, and `rewrite`

```
Definition counter cSz := MODULE {
  Register "counterReg" : Bit cSz <- Default
  with Rule "incrementAndOutput" :=
    Read val <- "counterReg";
    Write "counterReg" <- #val + $1;
    Call "output"(#val);
    Retv
}.
```

Fig. 1. A simple counter component in Kami

The code we show, in this figure and later ones, is literally embedded in standard Coq source files, thanks to Coq's extensible parser.

A rule consists of a sequence of *actions*, which are executed *atomically*. During execution of a rule, when an action reads a register, it receives the *current state* of the register, *i.e.,* the value of the register before the rule starts executing; when it writes to a register, the state update is collected (a register can be written only once in a rule); and when it calls a *method*, it sends the method call's argument to the external environment via wires in the hardware implementation corresponding to this module. The external environment could either be other Kami modules or an already-existing hardware design written in a traditional HDL like Verilog/VHDL, but its interaction with a Kami module remains the same, via methods. Once a rule is executed, the state updates collected during the execution are applied en-masse to the state. The `incrementAndOutput` rule keeps executing forever, incrementing the counter and outputting the value. During the execution of one rule, semantically, no other rule in the whole system executes, including those in the external environment. When these modules are compiled into hardware circuits, however, a scheduler is also generated (in hardware) that schedules multiple such rules simultaneously, as long as the concurrent execution of these rules is serializable.

Since Kami is embedded inside Coq, we can produce Kami modules with functions in Gallina, the functional programming language in Coq. In other words, we "for free" get parameterized Kami modules, meaning that they may depend on formal parameters that are not instantiated immediately with concrete values. For instance, the counter module is parameterized on `cSz`, which represents the bit width of `counterReg`. Parameters can be more complex. For instance, we could parameterize the counter over a Gallina *function* to implement the single-step evolution of the counter value, calling it in place of the "+ $1" operation in Figure 1. Parameters may range over expressions, actions, or even other Kami modules.

Pipelining is a widely used optimization for improving performance in hardware systems. We can refine the previous example using a two-stage pipeline containing a producer, a consumer, and a queue in between. The producer and consumer can be scheduled to operate concurrently, potentially increasing throughput, though the real value of pipelining will be more apparent in our later full-scale examples with more intensive computation.

Figure 2 shows the Kami code for the producer-consumer optimization. The queue module defines two methods: 1) enq, for enqueuing the supplied argument into its `elts` buffer, returning nothing, and 2) deq, for dequeuing the oldest entry from the buffer and returning it. These methods are called by `producer` and `consumer` respectively. Datatypes in Kami are not restricted to bit-vectors; for example, register `elts` in queue holds a vector of `dataType` entries, of size $2^{\mathtt{qSz}}$, with `dataType` and `qSz` parameters for the module queue.

An action in a rule or method can be an `Assert`, which ensures that the rule or method is executed only when the assertion holds. See, for example, method deq in queue, which ensures that the queue is nonempty whenever deq is called. Note that these are *not* the usual assertions

```
Definition producer cSz := MODULE {
  Register "counterReg" : Bit cSz <- Default
  with Rule "produce" :=
    Read val <- "counterReg";
    Call "enq"(#val);
    Write "counterReg" <- #val + $1;
    Retv
}.
Definition consumer cSz := MODULE {
  Rule "consume" :=
    Call val: Bit cSz <- "deq"();
    Call "output"(#val);
    Retv
}.
Definition queue dataType qSz := MODULE {
  Register "elts" : Vector dataType qSz <- Default
  with Register "head": Bit (qSz+1) <- Default
  with Register "tail": Bit (qSz+1) <- Default
  with Method "enq"(d : dataType) : Unit :=
    Read elts <- "elts";
    Read head <- "head";
    Read tail <- "tail";
    Assert (#tail + $(1<<qSz) != #head);
    Write "elts" <- #elts@[#head <- #d];
    Write "head" <- #head + $1;
    Retv
  with Method "deq"() : dType :=
    Read elts <- "elts";
    Read head <- "head";
    Read tail <- "tail";
    Assert (#tail != #head);
    Write "tail" <- #tail + $1;
    Return #elts@[#tail]
}.
Definition prodQCons cSz qSz :=
  producer cSz + queue (Bit cSz) qSz + consumer cSz.
```

Fig. 2. A producer-consumer example

of, say, C programs, where it is a program error if an assertion could ever fail at runtime. Rather, following Bluespec, assertions are *guards* to control which rules are enabled when.

Composing modules in Kami semantically results in a new Kami module that consists of the union of the registers and rules, plus the union of noninteracting methods, *i.e.,* those that are not defined by one and called by the other; the definitions of the interacting methods are semantically inlined at the places of their calls. In our example, when we compose the producer, queue, and consumer, the resulting module has registers `counterReg`, `elts`, `head`, and `tail`, rules `produce` and `consume`, and no resulting methods, with the definitions of `enq` and `deq` semantically inlined in the `produce` and `consume` rules, respectively.

It is easy to see informally that the `prodQCons` module indeed behaves just like the `counter` module in any external context that implements the `output` method. The producer keeps incrementing the counter and enqueuing these values, while the consumer module keeps dequeuing these values and outputting them, so the *trace* produced by `prodQCons`, consisting of `output` method calls, matches that produced by `counter`, denoted as (`prodQCons cSz qSz`) $\sqsubseteq$ (`counter cSz`). We will use the task of verifying that `prodQCons` *implements* `counter` as a running example throughout the paper.

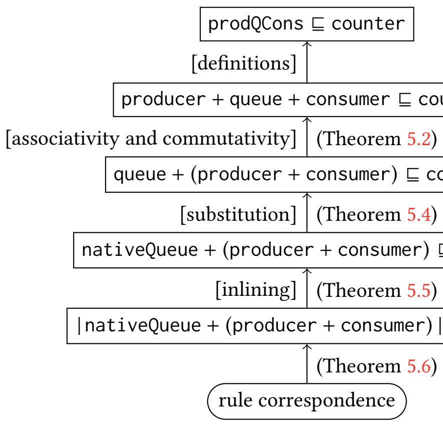
prodQCons ⊑ counter
[definitions]
producer + queue + consumer ⊑ counter
[associativity and commutativity]
(Theorem 5.2)
queue + (producer + consumer) ⊑ counter
[substitution]
(Theorem 5.4)
nativeQueue + (producer + consumer) ⊑ counter
[inlining]
(Theorem 5.5)
|nativeQueue + (producer + consumer)|⊑ counter
(Theorem 5.6)
rule correspondence

Fig. 3. Steps to prove that `prodQCons` implements `counter` (parameters are omitted for brevity)

```
Definition nativeQueue dType (def: dType) := MODULE {
  Register "elts" :< list dType <- nil
  with Method "enq"(d : dType) : Unit :=
    Read elt :< list dType <- "elts";
    Write "elts" <- #<(elt ++ [d]);
    Retv
  with Method "deq"() : dType :=
    Read elt :< list dType <- "elts";
    Assert $$@(match elt with nil => false | _ => true end);
    Write "elts" <- #<(tl elt);
    Ret (match elt with nil => $$def | h :: t => #h end)
}.
```

Fig. 4. A Gallina list-based queue specification

Figure 3 shows a high-level overview of the tasks involved in the verification; we elaborate on each of these tasks later in the paper. The hardware queue module is first substituted with a list-based specification of an unbounded queue, *i.e.*, the buffer inside this specification is represented by a Gallina list. The method calls `enq` and `deq` are then inlined at call sites. Next, one has to prove a simulation relation between the states in the inlined version of `prodQCons` and `counter` via *rule correspondence*. This proof uses the invariant that every element in the queue is the successor of the previous element, and the youngest element has the same value as the counter in `producer`. The value of `counterReg` in `counter` must match the value of `counterReg` in `prodQCons` whenever the queue is empty, and if not, must match the head of the queue in `prodQCons`. The rule correspondence is then used to produce a proof of trace inclusion. In Section 5, we decompose these verification tasks and detail the general proof ingredients behind them.

We will elaborate on replacing the queue with a Gallina list-based queue, given in Figure 4. First note that the native queue contains a register `elts` that stores a Gallina list of Kami elements of Kami type `dType`. Since the Gallina list does not have a capacity, the queue is unbounded; we can always append an element to the tail of the list. Reading the register returns a Gallina list that can be manipulated using Gallina functions (here `elt ++ [d]`). Note that in normal Kami code (without Gallina terms), Kami variables act as the variables we are used to from functional programming, but here, we take advantage of a pun enabled by the encoding we chose for variable binding, parametric higher-order abstract syntax [Chlipala 2008]. In brief, this encoding style represents terms as polymorphic functions, parameterized over representation types for variables, one per object-language type. For variables of embedded Coq types, however, we hardcode their representation types to match Coq's "native" types. In this way, wherever Kami's grammar asks for a variable, we are free to embed normal Gallina terms. For instance, in Figure 4, we give Gallina expression `#<(elt ++ [d])` in a variable position, written into register `elts`.

Whenever an element is dequeued, we check that the list is not empty using a Gallina `match` expression, update the `elts` register to contain the tail of the original list, and return the head of the original list, again using the Gallina `match` expression. The last step requires a default value `def`, which is never returned by this method in any call since the list is asserted to be nonempty.

The reader may at this point be feeling tricked by our earlier descriptions, as clearly here we are mixing in conventional high-level functional-programming code in what is purportedly a "hardware design." We do not depend on a general way of compiling functional programs to hardware. Instead, designs that use such features are *for specification purposes only*: they cannot be synthesized to circuits. Still, we found it convenient to define one Kami language with embedding of normal Coq programs as an optional feature, by convention to be used only for specifications, not "real" designs. As a result, our implementations and specifications are programs in the same language, which economizes on proof machinery.

3 KAMI LANGUAGE: SYNTAX AND SEMANTICS

3.1 Syntax

Figure 5 gives the syntax of the Kami language. Kami types (τ) include Booleans (Bool), bit-vectors ($\mathsf{Bit}\ n$), homogeneous vectors of length 2^n ($\mathsf{Vector}\ \tau\ n$), and records ($\{\overrightarrow{k :: \tau}\}$). In addition, one can inject Gallina terms into Kami modules as mentioned in Section 2, where the specification of the queue module is given using Gallina lists. We allow any Gallina type (t) to be lifted implicitly as a type (σ) in the Kami language. This feature should only be used in specifications, since in general Gallina types are infinite and cannot be represented uniformly in hardware circuits.

Expressions consist of constants (c), variables (x), operations on expressions ($\mathsf{op}(\vec{e})$), vector constructors ($[\vec{e}]$), record constructors ($\{\overrightarrow{k = e}\}$), vector-index reads ($e[e]$), and record-field reads ($e.k$). Actions consist of register reads ($\mathsf{let}\ x = r\ \mathsf{in}\ a$), register writes ($r := e\ ;\ a$), method calls ($\mathsf{let}\ x = f(e)\ \mathsf{in}\ a$), let bindings ($\mathsf{let}\ x = e\ \mathsf{in}\ a$), conditional actions ($\mathsf{if}\ e\ \mathsf{then}\ a\ \mathsf{else}\ a\ ;\ a$), assertions ($\mathsf{assert}\ e\ ;\ a$), and returns ($\mathsf{return}\ e$). Actions may include method calls both to their own modules and to different ones.

A Kami module is either a basic module containing a set of registers with initial values ($\langle r, c\rangle$), a set of rules ($\langle s, a\rangle$), and a set of methods ($\langle f, \lambda x : \tau.\ a\rangle$); or a composition of modules ($m + m$). The identifiers for register names and method definition names must be unique across all the basic modules taking part in a composition. The registers in a basic module can be initialized with syntactic constants, for syntactic Kami types; or with arbitrary Gallina terms, for lifted Gallina types.

Expressions and actions are typed in a straightforward manner, and a Kami module is well-typed if all its rules and methods are well-typed.

3.2 Semantics

A transition in a Kami module corresponds exactly to the execution of one rule in the system. This rule may lead to a chain of method calls, thereby allowing multiple methods to participate simultaneously in a transition. This chain of method calls can go out of a module (by calling a method in the external environment) and later come back into the same module (when the external environment responds by calling a method in the module). Thus, a priori, we do not know the set of methods that can participate in any transition. We therefore allow any combination of methods to be part of a transition orchestrated by a rule in a module. In the same vein, a set of methods of a module may participate in a transition orchestrated by a rule in the external environment.

$$
\begin{array}{rl}
\mathbb{N} & n \\
\text{Identifier} & r, f, s, k, \ldots \\
\text{Constant} & c, \mathsf{op}, \ldots \\
\text{Variable} & x \\
\text{Kami type} & \tau ::= \mathsf{Bool} \\
 & \quad \mid \ \mathsf{Bit}\ n \\
 & \quad \mid \ \mathsf{Vector}\ \tau\ n \\
 & \quad \mid \ \{\overrightarrow{k :: \tau}\} \\
\text{Gallina type} & t \\
\text{Type} & \sigma ::= \tau \mid t
\end{array}
\qquad
\begin{array}{rrl}
\text{Expression} & e & ::= c \mid x \mid \mathsf{op}(\vec{e}) \mid [\vec{e}] \mid \{\overrightarrow{k = e}\} \mid e[e] \mid e.k \\
\text{Action} & a & ::= \mathsf{let}\ x = r\ \mathsf{in}\ a \\
 & & \ \mid \ r := e\ ;\ a \\
 & & \ \mid \ \mathsf{let}\ x = f(e)\ \mathsf{in}\ a \\
 & & \ \mid \ \mathsf{let}\ x = e\ \mathsf{in}\ a \\
 & & \ \mid \ \mathsf{if}\ e\ \mathsf{then}\ a\ \mathsf{else}\ a\ ;\ a \\
 & & \ \mid \ \mathsf{assert}\ e\ ;\ a \\
 & & \ \mid \ \mathsf{return}\ e \\
\text{Module} & m & ::= \langle[\overrightarrow{\langle r, c\rangle}], [\overrightarrow{\langle s, a\rangle}], [\overrightarrow{\langle f, \lambda x : \tau.\ a\rangle}]\rangle \\
 & & \ \mid \ m + m
\end{array}
$$

Fig. 5. Kami language syntax

The semantics below formalizes these notions of rules and methods participating in transitions. They can be understood in terms of 5 main ideas: 1) expressions are read-only operations, 2) actions are the smallest unit of a transition that can be composed with other actions, 3) a *substep* is a composition of several actions, each of them corresponding to a rule or method, with some additional constraints, 4) a *step* is a substep that corresponds to a single atomic transition in the module, and 5) a *behavior* is a sequence of steps starting from the initial state of the module, *i.e.*, it is the transition sequence of a module.

Expressions. Figure 6a and Figure 6b give the denotational semantics for Kami types and expressions, respectively. Variables are replaced with the corresponding values before they are evaluated (by the semantics of actions, which create the variables), and hence their denotations are omitted.

Actions. Figure 6c gives the semantics for actions. The semantics for an action a is given by a judgment $o \xrightarrow[a]{\ell} (u, v)$, where o is the register mapping, denoting the current state; u is the map of register updates during execution of a; and v is the value returned after executing action a (which is relevant only if the action is the body of a method definition). Most interestingly, ℓ is a *label*, in the tradition of labeled transition systems from process algebra. It summarizes all externally visible interactions of the step. While in process algebra such interactions are sends and receives of messages on channels, in Kami the analogous interactions are making and receiving method calls. Actions themselves can only call methods, not define them, so the action rules only show examples of extending a label with a call $f(a) = b$. While the arguments are computed locally within the action, the return value is *assumed* and cannot be computed, in the same way that channel-receive operations are modeled in traditional process algebra.

We use the operator $\uplus$ for the disjoint union of register maps and labels. Such a union is only well-defined when the operands deal with disjoint registers or method names, respectively. For each use of $\uplus$ in a rule's conclusion, we assume disjointness of the arguments, as an implicit extra premise.

In the `producer` example in Figure 2, the action corresponding to rule `produce` results in the following transition (where `Retv` is shorthand for returning an empty value).

$$\{\mathsf{counterReg} \mapsto v\} \xrightarrow{\quad\{\mathsf{enq}(v)=()\}\quad} (\{\mathsf{counterReg} \mapsto v + 1\}, ())^{‡}$$

```
Read val <- "counterReg";
Call "enq"(#val);
Write "counterReg" <- #val + $1;
Retv
```

‡() represents a `word(0)` value, a dummy "unit" placeholder.

$$\llbracket \text{Bool} \rrbracket = \texttt{bool}$$
$$\llbracket \text{Bit } n \rrbracket = \texttt{word}(n)^{\S}$$
$$\llbracket \text{Vector } \tau \ n \rrbracket = \texttt{word}(n) \to \llbracket \tau \rrbracket$$
$$\llbracket \{k_1 :: \tau_{k_1}, \ldots, k_n :: \tau_{k_n}\} \rrbracket = \forall (k : \{k_1, \ldots, k_n\}).\ \llbracket \tau_k \rrbracket$$

(a) Denotations for Kami types (into Gallina types)

§a bit-vector of size n, implemented in Coq

$$\llbracket c \rrbracket = c$$
$$\llbracket \text{op}(\vec{e}) \rrbracket = \llbracket \text{op} \rrbracket_{\text{op}}(\overrightarrow{\llbracket e \rrbracket})$$
$$\llbracket [e_0, \ldots, e_{2^n-1}] \rrbracket = \lambda i : \texttt{word}(n).\ \llbracket e_i \rrbracket$$
$$\llbracket \{k_1 = e_{k_1}, \ldots, k_n = e_{k_n}\} \rrbracket = \lambda k : \{k_1, \ldots, k_n\}.\ \llbracket e_k \rrbracket$$
$$\llbracket e_v[e_i] \rrbracket = \llbracket e_v \rrbracket(\llbracket e_i \rrbracket)$$
$$\llbracket e.k \rrbracket = \llbracket e \rrbracket(k)$$

(b) Denotations for expressions

$$\text{ActionReadReg:}\ \frac{o \xrightarrow[{[o(r)/x]a}]{\ell} (u, v)}{o \xrightarrow[\texttt{let } x{=}r \texttt{ in } a]{\ell} (u, v)}$$

$$\text{ActionWriteReg:}\ \frac{\llbracket e \rrbracket = v_r \qquad r \notin u \qquad o \xrightarrow[a]{\ell} (u, v)}{o \xrightarrow[r{:=}e\,;\,a]{\ell} (u[r \leftarrow v_r], v)}$$

$$\text{ActionCall:}\ \frac{\llbracket e \rrbracket = v_a \qquad (f(\_) = \_) \notin \ell \qquad o \xrightarrow[{[v_r/x]a}]{\ell} (u, v)}{o \xrightarrow[\texttt{let } x{=}f(e) \texttt{ in } a]{\{f(v_a)=v_r\} \cup \ell} (u, v)}$$

$$\text{ActionLet:}\ \frac{\llbracket e \rrbracket = v_l \qquad o \xrightarrow[{[v_l/x]a}]{\ell} (u, v)}{o \xrightarrow[\texttt{let } x{=}e \texttt{ in } a]{\ell} (u, v)}$$

$$\text{ActionIfElseT:}\ \frac{\llbracket e \rrbracket = \text{true} \qquad o \xrightarrow[a_t]{\ell_t} (u_t, v_t) \qquad o \xrightarrow[a]{\ell} (u, v)}{o \xrightarrow[\texttt{if } e \texttt{ then } a_t \texttt{ else } a_f\,;\,a]{\ell_t \uplus \ell} (u_t \uplus u, v)}$$

$$\text{ActionIfElseF:}\ \frac{\llbracket e \rrbracket = \text{false} \qquad o \xrightarrow[a_f]{\ell_f} (u_f, v_f) \qquad o \xrightarrow[a]{\ell} (u, v)}{o \xrightarrow[\texttt{if } e \texttt{ then } a_t \texttt{ else } a_f\,;\,a]{\ell_f \uplus \ell} (u_f \uplus u, v)}$$

$$\text{ActionAssert:}\ \frac{\llbracket e \rrbracket = \text{true} \qquad o \xrightarrow[a]{\ell} (u, v)}{o \xrightarrow[\texttt{assert } e\,;\,a]{\ell} (u, v)}$$

$$\text{ActionReturn:}\ \frac{\llbracket e \rrbracket = v}{o \xrightarrow[\texttt{return } e]{\varnothing} ([], v)}$$

(c) Semantics for actions

$$\text{EmptyRule:}\ \frac{}{o \xrightarrow[m]{\{\bullet\}} []}$$

$$\text{EmptyMeth:}\ \frac{}{o \xrightarrow[m]{\varnothing} []}$$

$$\text{Rule:}\ \frac{\langle \_, a \rangle \in \text{rulesOf}(m) \qquad o \xrightarrow[a]{\ell} (u, \_)}{o \xrightarrow[m]{\{\bullet\} \uplus \ell} u}$$

$$\text{Meth:}\ \frac{\langle f, \lambda x.\ a \rangle \in \text{methsOf}(m) \qquad o \xrightarrow[{[v_a/x]a}]{\ell} (u, v_r)}{o \xrightarrow[m]{\{\overline{f}(v_a)=v_r\} \uplus \ell} u}$$

$$\text{SubstepConcat:}\ \frac{o \xrightarrow[m]{\ell_1} u_1 \qquad o \xrightarrow[m]{\ell_2} u_2}{o \xrightarrow[m]{\ell_1 \uplus \ell_2} u_1 \uplus u_2}$$

(d) Substep semantics

$$\text{StepIntro:}\ \frac{o \xrightarrow[m]{\ell} u \qquad m \odot \ell}{o \xRightarrow[m]{\ell \setminus m} u}$$

(e) Step semantics

$$\text{InitBehavior:}\ \frac{}{m \Downarrow \langle \text{initRegs}(m), [] \rangle}$$

$$\text{SequenceBehavior:}\ \frac{m \Downarrow \langle o, \alpha \rangle \qquad o \xRightarrow[m]{\ell} u}{m \Downarrow \langle o[u], \alpha; \ell \rangle^{\|}}$$

(f) Behavior semantics

‖$(\alpha; \ell)$ denotes the concatenation of ℓ to list α.

Fig. 6. The Kami language semantics

Similarly, in the queue example in Figure 2, the action corresponding to method deq runs as follows. The precondition ($t \neq h \Rightarrow \ldots$) for values t and h records the guard condition corresponding to any execution of this action, arising from an `Assert` in the code.

$$t \neq h \Rightarrow \{\texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow{\{\}} (\{\texttt{tail} \mapsto t+1\}, e(t))$$

```
Read elts <- "elts";
Read head <- "head";
Read tail <- "tail";
Assert (#tail != #head);
Write "tail" <- #tail + $1;
Return #elts@[#tail]
```

Substeps. A *substep* defines the execution of a collection of at most one rule and any number of methods. It is derived from the semantics of the actions comprising the rule and the methods as shown in Figure 6d. A set of one rule and many methods can be combined into a substep only if they have disjoint register updates and disjoint method calls. The methods participating in a substep, as well as all the called methods in all the actions forming the substep, form a *label* (as alluded to, when describing the semantics of actions). A label ℓ is a set formed from the following elements:

$$\text{Label element} \quad \omega \quad ::= \quad \bullet \mid f(a) = b \mid \overline{f}(a) = b$$

The label elements denote the presence of a rule ($\bullet$), a called method ($f(a) = b$), or its dual, an executed method ($\overline{f}(a) = b$) with method name f, argument a, and return value b. Two label elements can be combined into a label only when they overlap neither by calling nor defining the same method. This convention prevents a method from taking part twice in a transition and also prevents two actions that call the same method from participating in a substep. The latter restriction is needed because method calls are translated into wires in hardware; there is only one set of wires for a method, so a method can be called only once in a transition.

The semantics for a substep is given by a judgment $o \xrightarrow[m]{\ell} u$, where m is the target module, o is the old state, and u is the map containing register updates (Figure 6d). Rule EmptyRule generates a rule-like substep, but without any participating rules. This rule is typically used in specification modules to allow mapping transitions in the implementation to empty transitions in the specification, when the implementation's transition only affects low-level state that does not map directly to specification-level state. Rule EmptyMeth is defined for convenience and serves the purpose of the *nil* constructor in a list. Rules Rule and Meth describe the cases where one rule or one method is executed, respectively, and the resulting substep is called a *singleton*. rulesOf(m) and methsOf(m) collect all rules and methods in the module, respectively. Two substeps with disjoint effects may merge (SubstepConcat).

The nonempty substeps for the module `prodQCons` in Figure 2 are as follows (parameters of the module definition are omitted for brevity, and all arithmetic is on constant-sized bit-vectors):

- For rule consume:

$$\{\texttt{counterReg} \mapsto c, \texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow[\texttt{prodQCons}]{\{\bullet, \texttt{deq}(())=v, \texttt{output}(v)=()\}} \{\}$$

- For method deq:

$$t \neq h \Rightarrow \{\texttt{counterReg} \mapsto c, \texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow[\texttt{prodQCons}]{\{\overline{\texttt{deq}}(())=e(t)\}} \{\texttt{tail} \mapsto t+1\}$$

- For the combination of rule consume and method deq (using SubstepConcat):

$$t \neq h \Rightarrow \{\texttt{counterReg} \mapsto c, \texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow[\texttt{prodQCons}]{\{\bullet, \mathsf{deq}(())=v, \mathsf{output}(v)=(), \overline{\mathsf{deq}}(())=e(t)\}} \{\texttt{tail} \mapsto t+1\}$$

For the combination of rule consume, method enq, and method deq (using SubstepConcat):

$$t + 2^{\mathsf{qSz}} \neq h \wedge t \neq h \Rightarrow$$

$$\{\texttt{counterReg} \mapsto c, \texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow[\texttt{prodQCons}]{\{\bullet, \mathsf{deq}(())=v, \mathsf{output}(v)=(), \overline{\mathsf{enq}}(d)=(), \overline{\mathsf{deq}}(())=e(t)\}} \{\texttt{elts} \mapsto e[h \mapsto d], \texttt{head} \mapsto h+1, \texttt{tail} \mapsto t+1\}$$

Seven other rule/method combinations are possible, but we will not list them here, as they work out quite similarly to the ones we have shown. Note, though, that the rules produce and consume cannot be combined using SubstepConcat since the labels of the combining substeps should be disjoint and hence can have at most one • element.

Steps. A step is a single atomic state transition of a module, where all internal communications are hidden. It also ensures that internal calls are executed correctly, *i.e.,* if a $f(a) = b$ label came about by a call to method f defined in the same module, then $\overline{f}(a) = b$ is also present, and vice versa.

Figure 6e gives the details. The semantics of a step is given by the judgment ($o \overset{\ell}{\underset{m}{\Rightarrow}} u$), which is similar to the one for substeps. $\ell \setminus m$ removes all mentions of methods in ℓ that are both defined and called in m, and $m \odot \ell$ enforces the constraint on labels mentioned before, *i.e.,* $f(a) = b$ and $\overline{f}(a) = b$ are either both present or both absent for a method called and defined in m.

In the module prodQCons in Figure 2, the only possible steps are as follows:

$$t + 2^{\mathsf{qSz}} \neq h \Rightarrow \{\texttt{counterReg} \mapsto c, \texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow[\texttt{prodQCons}]{\{\bullet\}} \{\texttt{counterReg} \mapsto c+1, \texttt{elts} \mapsto e[h \mapsto c], \texttt{head} \mapsto h+1\}$$

Note that the counterReg's value c is inserted at head position h of the vector map e of elements elts. That is, the value enqueued matches the value that the queue uses to update its internal registers. Moreover, the labels corresponding to the call and execution of enq are removed.

$$t \neq h \Rightarrow \{\texttt{counterReg} \mapsto c, \texttt{elts} \mapsto e, \texttt{head} \mapsto h, \texttt{tail} \mapsto t\} \xrightarrow[\texttt{prodQCons}]{\{\bullet, \mathsf{output}(e(t))=()\}} \{\texttt{tail} \mapsto t+1\}$$

As in the previous case, the argument passed to the output call is $e(t)$ where e is the vector map denoting the elements elts and t is the value of tail, and the label contains only the *external* call output.

No other combination of substeps can produce a step. The methods enq and deq are called within the module prodQCons and hence must not be present in the step's label after removing all the methods that appear in the called and executed positions in the label. Thus, in order to eliminate these methods in the label, the rule that calls these methods must also execute. Since both the rules, produce and consume, cannot execute simultaneously, i.e. there cannot be a substep with the combined effects of execution of both the rules, we are left only with the above two combinations.

Behaviors. A behavior is given by a sequence of steps starting from the initial state of a module. Its semantics are given by the judgment $m \Downarrow \langle n, \alpha \rangle$, where m is the module that has reached state n producing a label sequence α. In Figure 6f, $\mathsf{initRegs}(m)$ creates a state by mapping a register name to its initial value, and $o[u]$ applies the updates u to override values of affected registers in o. A behavior is therefore given by a serialized sequence of rule executions; people accustomed to Bluespec call this the *one-rule-at-a-time semantics.*

3.3 Implementation Relation

We now give the formal definition for the implementation relation between two modules. A module m is said to implement a module m' (or m is said to refine m'), written $m \sqsubseteq m'$, iff any trace sequence produced by m can also be produced by m'. That is,

$$m \sqsubseteq m' \triangleq \forall n\ \alpha.\ m \Downarrow \langle n, \alpha \rangle \Rightarrow \exists n'.\ m' \Downarrow \langle n', \alpha \rangle$$

We write $m \cong m'$ to indicate that refinement holds in both directions.

4 COMPILATION INTO HARDWARE CIRCUITS

Kami is inspired by the Bluespec high-level hardware description language, many aspects of which should feel familiar to a functional-programming audience. However, this approach to hardware programming is far from mainstream. Most hardware engineers find it quite foreign. The default reaction is that RTL designs (as in Verilog) are somehow "real" hardware, while higher-level code as in Bluespec is more like software and counts as a completely separate enterprise. Our view is that both Bluespec and RTL are significant abstractions above real hardware, e.g. because translating RTL to physical circuits requires a highly nontrivial process of placement in space. Thankfully, hardware designers can rely on mature toolsuites to do automated compilation from either starting point, and we spend a few pages here justifying the practicality of the toolchain that starts from Bluespec code (which is itself easily produced by desugaring of Kami code).

We will describe the compilation of Kami modules into hardware circuits using the example shown in Figure 7 (where we use `Unit` as shorthand for a zero-width bit-vector type). We sketch the workings of the commercial (unverified) Bluespec compiler; for a complete description of the algorithm, including the proposed optimizations, refer to [Dave et al. 2007; Esposito et al. 2010; Hoe and Arvind 2000; Karczmarek et al. 2014; Rosenband and Arvind 2004; Vijayaraghavan et al. 2013].

Each rule compiles into a combinational circuit that generates output signals to update the registers that the rule writes, starting from the input signals that feed from the registers that the rule reads. In addition, the rule also produces a single Boolean *guard* input signal, g, that indicates that the assertions in the rule are all true, and a Boolean *enable* output signal, *en*, that indicates that the rule has been selected to execute (and this "enable" signal ultimately determines if the registers written by the rule are updated or not). Similarly, a method definition generates a combinational circuit that takes as input the signal corresponding to the argument of the method (*arg*), producing as output the signal corresponding to the return value of the method (*ret*), plus a guard output signal, g, for the assertions. In addition, a method can be called externally, so it has a Boolean input *enable* signal, *en*, to determine if the method is being called or not in the current cycle.

While the semantics of Kami programs executes the rules one by one, the compiler tries to schedule multiple rules for simultaneous execution (i.e., in one hardware clock cycle) without violating the one-rule-at-a-time semantics. In this example, rules `r1` and `r2` can never execute simultaneously without violating one-rule-at-a-time semantics. Similarly, if method `f1` is called externally (from another rule), then rule `r1` cannot execute simultaneously. During a clock cycle, in any hardware circuit, all the reads of registers happen at the beginning of the clock cycle, and all

```
Definition example := MODULE {
  Register "x1" : Bit 32 <- 0
  Register "x2" : Bit 32 <- 1
  Register "x3" : Bit 32 <- 2
  Register "x4" : Bit 32 <- 3
  with Rule "r1" :=
    Read v <- "x2";
    Assert (#v mod 2 == 0);
    Write "x1" <- #v + $1;
    Retv
  with Rule "r2" :=
    Read v <- "x1";
    Assert (#v mod 2 == 1);
    Write "x2" <- #v + $1;
    Retv
  with Rule "r3" :=
    Read v <- "x1";
    Assert (#v mod 2 == 0);
    Write "x3" <- #v + $1;
    Retv
  with Rule "r4" :=
    Read v <- "x4";
    Assert (#v mod 2 == 1);
    Write "x4" <- #v + $1;
    Retv
  with Method "f1"(a: Bit 32): Unit :=
    Read v <- "x1";
    Assert (#v mod 2 == 1);
    Write "x1" <- #a;
    Retv
  with Method "f2"(_: Unit): Bit 32 :=
    Read v <- "x1";
    Return #v
}.
```

Fig. 7. Kami example to illustrate the compilation procedure

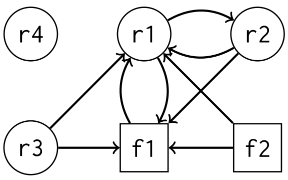

Fig. 8. Graph showing the binary relation (<) between atomic actions, for the example of Figure 7. $x < y$ is represented by an edge from x to y. Circles represent rules, and rectangles represent methods.

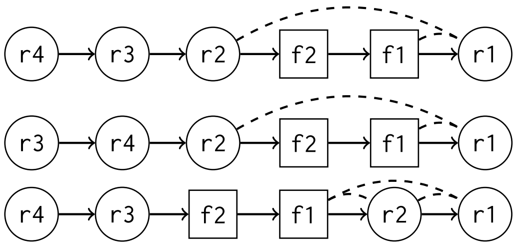

Fig. 9. Nonexhaustive list of schedules that satisfy the conditions that preserve one-rule-at-a-time semantics. Dashed lines represent a rule x being disabled by an earlier atomic action y that is enabled, because the total order for the schedule violates the constraint $x < y$.

the writes happen at the end of the clock cycle. Thus, rules `r1` and `r3` can execute simultaneously without violating the semantics: it appears as if rule `r3` fires, followed by rule `r1`.

The read and write sets for the rules and the method definitions determine a binary relation (<) on the set of rules and methods of a Kami module. We use the phrase *atomic action* to stand for either a rule or method definition in a Kami module. If an atomic action x has a read or a write to a register, and another atomic action y has a write to the same register, then $x < y$. In our example, the < relation is given by Figure 8. Whenever there is a pair $x < y$ and $y < x$, we say that x and y have a *conflict*.

From the < relation, the Bluespec compiler creates a scheduler circuit, which determines which rules and methods are executed every hardware clock cycle, and the sequence in which they are executed. The scheduler statically determines a total order of execution of atomic actions such that whenever an atomic action is enabled, then all the rules following it that have a conflict with the former are disabled from firing, even if their guards are true. This ensures that any simultaneous rule executions are consistent with some serial schedule. The exact constraints on the total order are as follows:

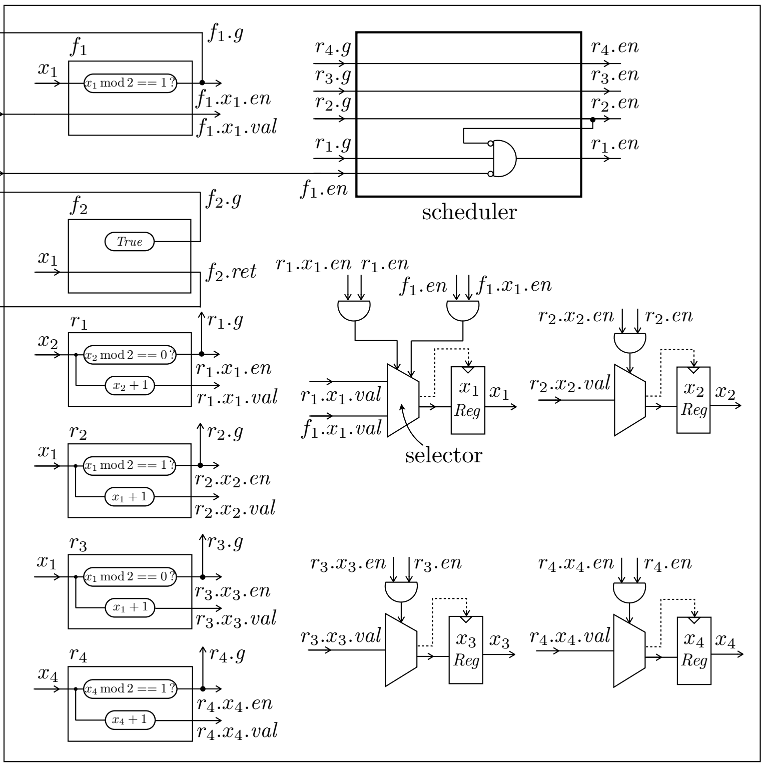

Fig. 10. Hardware circuit generated from Kami module of Figure 7. Wires with identical names are meant to be connected. Complex combinational circuits are abstracted using ovals labeled with the logic they implement.

(1) Whenever an atomic action x appears before rule y in the total order, if $y < x$ in the original binary relation then y is disabled from firing if x is enabled, even if the guard for y is true.
(2) If $f < r$ where f is a method and r is a rule, then f is before r in the total order.
(3) There are no rules between any two methods in the total order.
(4) A rule is chosen to fire only if its guard is true.

The first condition ensures that the rules and methods firing in the current cycle are serializable. The second condition ensures that no method can potentially be disabled by a rule, while the third condition ensures that if two methods are called by the same external rule, then a rule of the current module cannot fire in the middle of the atomic execution of the external rule calling the two methods. The fourth condition ensures that there are no wasted scheduling slots.

The presence of calls to methods of other modules complicates this analysis [Vijayaraghavan et al. 2013]. The called methods create another set of inputs and outputs to the circuit corresponding to the module: inputs being the value returned by the call and the enable signal of the method, and outputs being the argument for the call and the guard signal of the method. And, because of these method calls, the read/write-set analysis for registers is incomplete – one also has to determine the $<$ relation between called methods. It can be defined recursively in the obvious manner, with the base case being the $<$ relation between registers.

Using the above constraints for generating the scheduler results in the schedules shown in Figure 9 for our example in Figure 7. A solid edge represents the total order, and a dashed edge represents the condition when a rule is disabled because another rule or method that is earlier in the total order is enabled. Note that Figure 9 is not the exhaustive list of all valid schedules. The Bluespec compiler has some heuristics to try to pick a schedule that enables the maximum number of rules in a given hardware clock cycle [Esposito et al. 2010]. Figure 10 shows the final circuit containing the scheduler (using the first schedule of Figure 9), with the combinational circuits for rules and method definitions.

Figure 10 is missing the reset/initialization logic and the hardware clock, to avoid clutter. Every synthesized register is clocked by an input "clock" signal and reset by an input "reset" signal. The value to update a register is multiplexed between the initial value, whenever the reset signal is enabled, and the value generated by atomic actions, otherwise. The "selector" in Figure 10 in front of every register ensures that the update value that is selected corresponds to the enable signal that is active. Since the scheduler's constraint to avoid conflicts ensures that at most one atomic action writes any given register at a time, only one enable signal will be active if there are multiple atomic actions updating the same register.

In addition to circuit synthesis, the Bluespec compiler also performs Boolean optimizations aggressively. For instance, in Figure 10, since the values of signals `r4.en`, `r4.g`, and `r4.x4.en` are always the same, `r4.x4.en` can be fed directly into the selector, eliminating the "and" gate between signals `r4.x4.en` and `r4.en`.

5 KAMI VERIFICATION FLOW

Verifying a Kami module is synonymous to proving that the module refines its specification module, via repeated application of the three key high-level procedures: substituting one (sub)module by another, inlining method definitions at call sites, and proving the implementation relation using rule and method correspondences. We will now discuss these procedures in detail using the examples from Section 2. For brevity, we will omit writing the parameters for `prodQCons`, `queue`, and `nativeQueue`.

5.1 Substitution

Say we want to prove `prodQCons` $\sqsubseteq$ `counter`. Reasoning about the `queue` module makes this proof more complicated than it needs to be, so we would like to pretend that `queue` is really `nativeQueue`. In order to allow substituting any submodule in a module composition, we need properties like associativity and commutativity, among others.

Theorem 5.1. *Relation $\sqsubseteq$ is reflexive and transitive.*

Theorem 5.2. *Composition of modules is associative and commutative w.r.t. relation $\cong$.*

The idea of modular refinement captures proof patterns where we substitute one module for another and automatically derive full-system correctness via proving that the substituted module implements the one it replaces. This kind of modularity is crucial to established reasoning techniques for software systems, and it is just as valuable in the computer-architecture domain. It enables developing optimized off-the-shelf modules, with clear and simple specifications, that can be composed together to form large systems.

When proving one composite module refines another, it is useful to consider a relaxed notion of modular refinement to relate their submodules. For instance, consider the verification problem of proving that a cache hierarchy is an implementation of an atomic memory specification (see Section 6.3). When a request is present in the request queue for a cache, the cache must first read the request without dequeuing (by calling a new method `peek` on the queue, not present in the queue of Figure 2). If the address is not present in the cache, it sends a request to the main memory to fetch it, going into a wait state. Finally, when the main memory responds, the cache can dequeue the original request. On the other hand, the atomic memory specification simply dequeues the request instead of reading it first, as it always has the address. Therefore, to compare the cache hierarchy with the atomic memory, the peek method has to be filtered out.

We formalize this notion using a function p : (Label element $\rightarrow$ `option`(Label element)), which can either remove a label element (None) or modify its argument and return values. Such transformations must handle the called-method and executed-method label elements symmetrically, *i.e.*, $f(a) = b$ and $\overline{f}(a) = b$ together are both mapped to None or mapped to some $f(x) = y$ and $\overline{f}(x) = y$, respectively. Modification can be lifted to an operation $\hat{p}$ that modifies entire labels by applying p to each element in the label, using which we extend our notion of refinement.

Definition 5.3. If p : (Label element $\rightarrow$ `option`(Label element)) treats label elements symmetrically as explained above, then $m \sqsubseteq_p m' \triangleq \forall n\ \alpha.\ m \Downarrow \langle n, \alpha \rangle \Rightarrow \exists n'.\ m' \Downarrow \langle n', \mathsf{map}\ \hat{p}\ \alpha \rangle$.

Theorem 5.4 (modular refinement).

(1) *If no methods called in m_1 are defined in m_2 and vice versa, and similarly for m'_1 and m'_2, then* $m_1 \sqsubseteq_p m'_1 \wedge m_2 \sqsubseteq_p m'_2 \Rightarrow m_1 + m_2 \sqsubseteq_p m'_1 + m'_2$

(2) *If p changes the label elements only for methods called in m_1 and defined in m_2 or vice versa, then* $m_1 \sqsubseteq_p m'_1 \wedge m_2 \sqsubseteq_p m'_2 \Rightarrow m_1 + m_2 \sqsubseteq m'_1 + m'_2$

In the second case in Theorem 5.4, since label modification only affects the communicating methods, which are erased in the composition, the effect of modification is not visible externally. Thus, we can conclude refinement holds without any label modification.

We will now show how we can prove our example in Section 2 using the above theorems. By applying commutativity and associativity appropriately, we can change the goal from `prodQCons` $\sqsubseteq$ `counter` to `queue` + (`producer` + `consumer`) $\sqsubseteq$ `counter`. Applying the modular refinement theorem (interacting case, with the identity mapping and with the instantiations m_1 = `queue`, m'_1 = `nativeQueue`, $m_2 = m'_2$ = `producer` + `consumer`) and transitivity, we get three new goals `nativeQueue` + (`producer` + `consumer`) $\sqsubseteq$ `counter`, `queue` $\sqsubseteq$ `nativeQueue` and `producer` + `consumer` $\sqsubseteq$ `producer` + `consumer`. The third goal can be proved using reflexivity (and hence we call replacing `queue` with `nativeQueue` in `prodQCons` a *substitution* because it appears as if we are substituting one term with another while keeping the rest of the terms the same). Proving the rest of the goals requires inlining and application of rule and method correspondences. We discuss those next.

5.2 Inlining Method Definitions

We mentioned earlier that the semantics of method execution is similar to having the body of the method inlined at call sites. The semantics discussed in Figure 6 does not, however, take this idea literally. Still, we can show that static inlining is compatible with the base semantics. We will use $|m|$ to denote the module where all the locally defined methods of m are inlined and removed, whenever appropriate. Then,

Theorem 5.5. $m \sqsubseteq |m|$

Static inlining is very helpful for automatic verification, especially for modules without externally visible defined methods (as composing two modules hides the methods used for communicating between the two modules). A Kami step is comprised of several methods and (at most) one rule. If there are no externally visible defined methods in a module, then steps are essentially one-to-one with rule executions. When analyzing a step representing the rule during verification, it is both

```
|nativeQueue + (producer + consumer)| ≜ MODULE {
  Register "counterReg" : Bit cSz <- Default
  Register "elts" :< list dType <- nil
  with Rule "produce" :=
    Read val <- "counterReg";
    Read elt :< list dType <- "elts";
    Write "elts" <- #<(elt ++ [val]);
    Write "counterReg" <- #val + $1;
    Retv
  with Rule "consume" :=
    Read elt :< list dType <- "elts";
    Assert $$@(match elt with nil => false | _ => true end);
    Write "elts" <- #<(tl elt);
    Let val: dType = (match elt with nil => $$c | h :: t => #h end);
    Call "output"(#val);
    Retv
}.
```

Fig. 11. Result of inlining the `nativeQueue` + (`producer` + `consumer`) module

faster and conceptually easier to inline all the methods instead of evaluating the call chain of the rule and substituting the method bodies during analysis.

For instance, in our example in Section 2, after substitution, we are left with proving that `nativeQueue` + (`producer` + `consumer`) ⊑ `counter`. Inlining the left side results in Figure 11.

Inlining automatically discards the method definitions enq and deq, leaving us only the two rules to analyze, which, as mentioned earlier, directly correspond to Kami steps. If we were to work with the original module `nativeQueue` + (`producer` + `consumer`), then we would have two rules and two methods, requiring us to analyze every combination of these that form Kami steps. Instead, applying inlining leaves the goals |`nativeQueue` + (`producer` + `consumer`)| ⊑ `counter` and `nativeQueue` ⊑ queue.

A subtlety of inlining arises in connection to parameterized modules. We have seen that parameterization is accomplished by writing modules as Gallina functions from parameters to the syntax of modules. The module code itself may be considered to include free variables bound by these functions. In general, when we ask Coq to compute in code with free variables, execution may get stuck pattern matching on those variables. Inlining, implemented as a recursive function in Coq, is no exception, so we are not able to perform inlining on parameterized modules solely by computation. Instead, we need to interleave steps of computation and explicit deduction via rewrite rules establishing facts like n + "`foo`" $\neq$ n + "`bar`" (using string concatenation), no matter what value n takes. Another point to note is that while we almost always completely inline all methods in their respective call sites, the inlining procedure is implemented on a method-by-method basis, with a wrapper function to perform the complete inlining. This staging is crucial in allowing the interleaving of computation steps and explicit deduction via rewrite rules.

5.3 Leveraging Rule and Method Correspondences

In order to verify that a Kami step in the implementation corresponds to a Kami step in the specification, one has to reason about all the possible combinations of methods and rules of the implementation (with the constraint that there can be only one rule taking part in a Kami step). The next theorem describes a useful special case that lets us skip analyzing all such combinations.

Theorem 5.6. *Let m and m' be two modules such that either m defines no methods, or every method and every rule writes to at least one common register. Let p be a label map and R be a relation between the states of m and m'. If*

(1) *The initial states of m and m' are related,* i.e., $R(\mathit{initRegs}(m), \mathit{initRegs}(m'))$, *and*

(2) *If states o and o' of the implementation and specification are related,* i.e., $R(o, o')$, *and for every singleton substep in the implementation with updates u and label ℓ,* i.e., $o \xrightarrow[m]{\ell} u$, *there is a corresponding substep of the specification with updates u' and label $\hat{p}(\ell)$,* i.e., $o \xrightarrow[m']{\hat{p}(\ell)} u'$ *such that the states of the implementation and specification post-updates are related,* i.e., $R(o[u], o'[u'])$,

then $m \sqsubseteq_p m'$.

This theorem is similar to its classic LTS counterpart that proves trace inclusion by showing a simulation relation between the states of the two transition systems. If there are no methods defined in m, or if every method and rule write to the same register, then we cannot have Kami substeps that combine multiple methods and rules, giving a simple one-to-one correspondence.

Using the above theorem, one can prove that |`nativeQueue`+(`producer`+`consumer`)| $\sqsubseteq$ `counter` by mapping the rules `produce` and `consume` in |`nativeQueue` + (`producer` + `consumer`)| to an empty rule and the `incrementAndOutput` rule in `counter`, respectively. Similarly, we can use this theorem to prove `queue` $\sqsubseteq$ `nativeQueue` because both the methods of `queue` write to register `elts`.

5.4 Generator Templates in Kami

While Figure 5 represents the core syntax of the Kami language, the Kami framework also allows cloning of n copies of a set of rules, methods, and/or modules. We write code *templates* that include loops over parameter values. The identifiers representing register and method names (both in the calls and in the definitions) defined in the template are explicitly annotated to indicate whether they also have to be renamed (by appropriately concatenating the iterator and the name in the template) or whether they should be the same across all members of the list. A list of modules created this way results in the composition of the corresponding modules, and in this case, the identifiers for register names and names of method definitions are always renamed appropriately.

The most common use for the above procedure is for creating n replicas of modules, with the called methods also renamed appropriately. We represent such a composition of n renamed copies of Kami module m with $n \cdot m$. For example,

```
2 · (counter cSz) ≜
  MODULE {
      Register "counterReg_0" : Bit cSz <- Default
      with Rule "incrementAndOutput_0" :=
        Read val <- "counterReg_0";
        Write "counterReg_0" <- #val + $1;
        Call "output_0"(#val);
        Retv
    } +
  MODULE {
      Register "counterReg_1" : Bit cSz <- Default
      with Rule "incrementAndOutput_1" :=
        Read val <- "counterReg_1";
        Write "counterReg_1" <- #val + $1;
        Call "output_1"(#val);
        Retv
    }
```

In the case of two generated lists of modules $n \cdot a$ and $n \cdot b$, where a and b are ordinary modules, if the implementation relation is verified for a and b, then because of Theorems 5.1 to 5.4, we get $n \cdot a \sqsubseteq_{p^n} n \cdot b$, where p^n denotes the function formed by lifting p to operate on the label generated by the composition of the list of modules (by dealing appropriately with iteration-specialized identifiers). We will be using this property, informally called *replication*, in verifying our multiprocessor example later.

Corollary 5.7 (replication). $a \sqsubseteq_p b \Rightarrow n \cdot a \sqsubseteq_{p^n} n \cdot b$

Along with proving these theorems in Coq, we have also developed *tactics* for 1) substituting modules with other modules, 2) inlining, and 3) applying the rule and method correspondence theorems. These tactics automatically discharge trivial goals, such as those requiring commutativity, associativity, transitivity, reflexivity, etc. The final steps, *i.e.,* applying the rule and method correspondence theorems, require reasoning about complex invariants and simulation relations between states of the implementation and specification. We have developed several proof-searching tactics to help automate this reasoning.

6 CASE STUDY: VERIFYING A MULTIPROCESSOR

We used Kami to specify, implement, and verify a realistic multiprocessor system, or rather an infinite family of multiprocessor systems. The underlying specification is a simple ISA semantics with Lamport's sequential consistency (SC) [Lamport 1979] as the memory model[§]. We formalize SC as a Kami module and prove a trace refinement from the implementation to the spec. The processor is pipelined in order to support simultaneous execution of multiple instructions at different stages of their lifecycles, and the memory system contains coherent caches. We adapt pipelining techniques and cache-coherence protocols used widely in real processors, and we structure our proof as a sequence of refinements that take us from spec to implementation.

While the processors and the memory subsystem employ many parameters (like queue sizes, cache sizes, line sizes, *etc.*), the most important parameter is the number of processors in the multicore system; we write $m^{(n)}$ to emphasize that module m is parameterized on n, the number of processors in the multicore system.

6.1 Sequential Consistency as a Specification

Following Lamport's definition of sequential consistency, we designed a specification module to represent SC. Figure 12 presents the module structure of SC. It consists of multiple *instantaneous processors* ($\mathsf{P}_{\mathsf{inst}}$) and an *instantaneous memory* ($\mathsf{M}_{\mathsf{inst}}$), *i.e.,* $\mathsf{SC} \triangleq n \cdot \mathsf{P}_{\mathsf{inst}} + \mathsf{M}_{\mathsf{inst}}$.

The instantaneous processor ($\mathsf{P}_{\mathsf{inst}}$) and memory ($\mathsf{M}_{\mathsf{inst}}$) handle each instruction within a single rule (cycle). The processor fetches, decodes, and executes an instruction at once. For memory instructions, $\mathsf{P}_{\mathsf{inst}}$ gets the instantaneous response from $\mathsf{M}_{\mathsf{inst}}$, since requests to memory are also handled within a single method. It is easy to see that this specification faithfully encodes the usual definition of SC. In addition, we intentionally added a feature for $\mathsf{P}_{\mathsf{inst}}$ to call `tohost`, which is the only external behavior $\mathsf{P}_{\mathsf{inst}}$ can generate, via a synthetic instruction for sending a message to the world. Without it, SC would have no externally observable behaviors, which would make for underwhelming final correctness theorems.

SC is abstracted over the ISA using function parameters. For example, SC is parameterized over a function argument getOptype, which takes an instruction and returns its opcode.

6.2 Pipelined Processor Implementation

We designed and implemented a pipelined processor ($\mathsf{P}_{\mathsf{4st}}$) and proved it implements SC. Here we have 4 stages (Fetch, Decode, Execute, and Memory/Write-back), each drawn from common

[§] We just chose sequential consistency as an example. We can verify other memory models just as easily, provided we have their specifications.

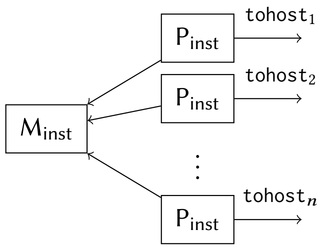

Fig. 12. The Sequential Consistency module (SC)

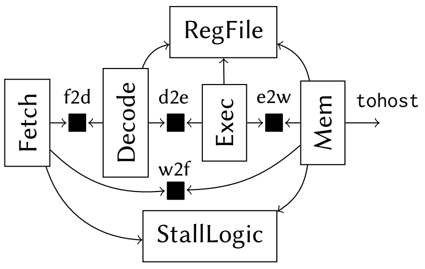

Fig. 13. P_{4st} module

processor-implementation strategies. The processor also abstracts over ISA details (using the same function parameters as the spec), and each pipeline stage interprets instructions using ISA-specific parameters. Figure 13 presents the structure of P_{4st}. A one-element queue (■) connects each two adjacent stages. A register file is configured for sharing across pipeline stages. The Mem module requests memory operations and handles the responses asynchronously, polling the memory for data availability.

Whenever an instruction reads a register being written by an earlier instruction still being processed in the pipeline stages, the former instruction must wait until the earlier instruction commits its updates. We use a Boolean array to keep track of which registers are being written by in-flight instructions, *i.e.,* those in the pipeline. Thus, we have to make an instruction wait even if it writes a register that is being written by an earlier in-flight instruction, to avoid premature untracking of a register being written by two in-flight instructions.

Since an instruction has to be executed fully before determining which instruction to fetch next, the pipeline can never process instructions concurrently unless the next instruction is fetched speculatively. A branch predictor predicts the address of the next instruction, which is then fetched. At the end of the memory stage, when the actual direction and address of the branch is known, if the prediction for the next in-flight instruction was incorrect, the program counter (PC) used to fetch instructions is corrected, and the existing in-flight instructions are marked as no-ops.

6.2.1 Proof Overview. Just as in verification of concurrent programs, the biggest hurdle for proving the pipelined processor comes in dealing with interleaving among instruction executions. Interleaving is inevitable in pipelined systems; any two stages can (and are expected to) operate simultaneously, which manifests in our Bluespec-style atomic semantics by allowing n different stages to be scheduled in any of $n!$ possible orders. In our 4-stage processor P_{4st}, for instance, the next instruction can be fetched while a current instruction is executed. The strawman approach would be to do brute-force enumeration of all possible interleavings, proving each schedule separately. However, there are several techniques to avoid brute force. We demonstrate that the Kami framework's *modular refinement* principles help us employ one such technique, namely "stage merging."

The refinement from P_{inst} to P_{4st} is proven through a number of steps. Firstly, we prove $P_{4st} \sqsubseteq P_{3st}$, in which Fetch and Decode are merged from P_{4st}. A merged module (FD) fetches and decodes an instruction within a single rule. One can then prove (Fetch + f2d + Decode) $\sqsubseteq$ FD, in the style of the producer-consumer example in Section 2, and thus substitute FD in the original goal and continue the proof from that point.

The decoupled processor (P_{dec}) acts as a next intermediate refinement step from P_{inst} to P_{3st}. It has a similar design to P_{inst}, except that memory requests and responses are not instantaneous. Like P_{4st} or P_{3st}, it waits for the response after each request. We use rule correspondence to give a refinement proof from P_{dec} to P_{3st}. The refinement from P_{inst} to P_{dec} requires a way to translate asynchronous memory requests/responses to instantaneous ones. As a solution, (MRqRs + MWrap) is composed with $\mathsf{P}_{\mathsf{dec}}$ to relate it to $\mathsf{P}_{\mathsf{inst}}$. MRqRs consists of two queues, one for requests and the other for responses. MWrap contains just a single rule that pulls a request, gets the result instantaneously, and pushes the result as a response. MRqRs and MWrap are structures borrowed from the memory subsystem (see Section 6.4).

6.3 Coherent Caches + Memory

The memory subsystem ($\mathsf{M}^{(n)}_{\mathsf{cache}}$) for the multicore processor consists of coherent caches connected by a crossbar to memory. The sizes of the caches and their cache lines (*i.e.,* the granularity of access in a cache, which is usually 64 bytes, as opposed to the granularity of access in a processor, which is usually 4 or 8 bytes) are parameters.

The cache-coherence protocol is a directory-based MSI protocol [Dave et al. 2005; Sorin et al. 2011; Vijayaraghavan et al. 2015], which scales well with the number of cores. The protocol ensures that whenever a cache has permission to write an address, no other cache has permission to read or write that address. If a cache has read or write permission for an address, and another cache wants to write that address, then the former cache gives up all its permission for the address. On the other hand, if a cache has write permission for an address, and another cache wants to read that address, then the former downgrades itself to having only read permission. The memory, in addition to serving as a backing store for the caches, also contains a *directory*, which records permissions for the different caches.

The cache-coherence protocol, in effect, is a distributed algorithm for correct synchronization of permissions (and data) between the memory and the caches, simulating an atomic memory, which is just an instantaneous memory ($\mathsf{M}^{(n)}_{\mathsf{inst}}$), with the rules for accessing the instantaneous memory (MWrap). Formally, $\mathsf{M}^{(n)}_{\mathsf{atomic}} \triangleq n \cdot \mathsf{MWrap} + \mathsf{M}^{(n)}_{\mathsf{inst}}$.

Now we outline how the Kami rules for the cache work. When a cache gets a request from its processor, it checks to see if the cache permission is high enough to process the request. If so, it sends back a response, performs appropriate updates, and dequeues the request.

On receiving a request that *misses* in the cache and hence cannot be processed immediately, the cache first obtains a slot to keep the address (if the address is not already in the cache). This process may involve evicting another address from the cache. After obtaining a slot for the request, an upgrade request is sent to the memory.

When the memory gets this upgrade request, it checks to see if the other children's cache permissions are compatible with the requesting child's upgrade. It does so by consulting its directory. The invariant that we maintain in this protocol is that the directory contains a conservative estimate of the children's cache permissions. If the other children's cache permissions are compatible, it dequeues the request, upgrades the requesting child's directory information, and sends back the response.

On receiving a request that cannot be processed immediately because the other children's cache permissions are incompatible (as indicated by the directory), the memory sends downgrade requests to the incompatible children. These children, if they have more permission than what is requested by the parent, must downgrade and respond to the parent. A child with write permission must also send back the data for this memory address. As the corresponding responses are obtained, the directory is updated to reflect the downgrades, and the data is updated if the response was obtained from a child with write permissions. Finally, when all the other children's cache permissions have become compatible, the original child's request is dequeued and responded to – the data is sent by the parent if the original child had no read or write permissions (as indicated by the directory).

Once the response from the parent is received back at the original cache for its upgrade request, then the permission is upgraded, and the data is updated if the cache had no permissions. Finally, the request from the processor is dequeued (and the data again updated in case of a store), and a response is sent back to the processor.

6.3.1 Proof Overview. We want to prove the implementation relation between our memory system and the atomic memory:

Theorem 6.1. *$M^{(n)}_{cache} \sqsubseteq_{dropPeek^n} (n \cdot MWrap + M^{(n)}_{inst})$*

We use the notion of the extended implementation relation (Definition 5.3) in order to drop label elements calling the peek method of the queue. The reason was explained in Section 5.1 – the atomic memory specification always dequeues any incoming request, while the cache has to get the required data and permissions after looking at the requests, before dequeuing them.

The initial step in this proof is to inline $\mathsf{M}^{(n)}_{\mathsf{cache}}$. The heart of the proof is an invariant over the combined states of the cache-coherence system and the specification-level atomic memory, as we run the two in lock-step through a simulation argument. The elements of the invariant are:

(1) The directory's notion of each of the caches' permissions for any address is conservative, *i.e.,* the cache can never have more permission than what the directory indicates.
(2) Whenever a cache has read or write permission for an address, the value it has for that address matches that in the atomic memory.
(3) Whenever a cache has write permission for an address, no other cache has read or write permission for that address.
(4) Whenever a message is in flight from a cache to the memory for an address, and the directory indicates that the cache has write permission for that address, then the data present in the message matches that in the atomic memory for the address.
(5) Whenever a message is in flight from the memory to the cache for an address, and the cache has no permission for that address, then the data present in the message matches that in the atomic memory for the address.
(6) Whenever the directory indicates that no cache has write permission, then the data present in the memory matches that in the atomic memory.

In order to prove these main invariants, we needed 30 additional low-level invariants that must be proven mutually inductively. The Ltac-based automation we developed was extremely useful in discharging most of the easy proofs, and the rest of the cases were discharged manually, with a bit of context-specific Ltac automation.

6.4 Modular Composition of Processor and Memory

The last two subsections introduced the several refinement proofs that we carried out on the individual components of our case study. We finish by composing these proofs together into a full-system theorem.

Theorem 6.2 (the final refinement). *$n \cdot P_{4st} + n \cdot MRqRsEx + M^{(n)}_{cache} \sqsubseteq SC^{(n)}$.*

The proof of this theorem is shown in Figure 14. One point to note is that MRqRsEx consists of two queues like MRqRs, but the queue for requests has an additional method called peek for reading the head of the queue (if nonempty). Just as queue $\sqsubseteq$ `nativeQueue` was proved using Theorem 5.6, we can also prove the following:

Theorem 6.3. *$MRqRsEx \sqsubseteq_{dropPeek} MRqRs$.*

6.5 Comments and Experiences from Our Case Study

The overall Kami infrastructure includes about 25,000 lines of code. We had to augment the standard Coq library with our own (to define bit-vectors, etc.), adding up to about 7,000 lines. The whole

$$\begin{tikzcd}
 & \boxed{n \cdot \mathsf{P}_{\mathsf{4st}} + n \cdot \mathsf{MRqRsEx} + \mathsf{M}^{(n)}_{\mathsf{cache}} \sqsubseteq \mathsf{SC}^{(n)}} & \\
(1) \arrow[ur, bend left] & \text{[transitivity]} & (2) \arrow[ul, bend right]
\end{tikzcd}$$

$$\begin{tikzcd}[column sep=small]
 & (1) \arrow[d, dash] & & \\
 & \boxed{n \cdot \mathsf{P}_{\mathsf{4st}} + n \cdot \mathsf{MRqRsEx} + \mathsf{M}^{(n)}_{\mathsf{cache}} \sqsubseteq n \cdot \mathsf{P}_{\mathsf{dec}} + n \cdot \mathsf{MRqRs} + \mathsf{M}^{(n)}_{\mathsf{atomic}}} & & \\
 & \boxed{\begin{array}{l} n \cdot \mathsf{P}_{\mathsf{4st}} + n \cdot \mathsf{MRqRsEx} + \mathsf{M}^{(n)}_{\mathsf{cache}} \\ \sqsubseteq n \cdot \mathsf{P}_{\mathsf{dec}} + n \cdot \mathsf{MRqRs} + n \cdot \mathsf{MWrap} + \mathsf{M}^{(n)}_{\mathsf{inst}} \end{array}} \arrow[u, "\text{[definitions]}"'] & & \\
\boxed{n \cdot \mathsf{P}_{\mathsf{4st}} \sqsubseteq n \cdot \mathsf{P}_{\mathsf{dec}}} \arrow[ur, bend left, "\text{[modular refinement]}"] & & \boxed{\begin{array}{l} n \cdot \mathsf{MRqRsEx} + \mathsf{M}^{(n)}_{\mathsf{cache}} \\ \sqsubseteq n \cdot \mathsf{MRqRs} + n \cdot \mathsf{MWrap} + \mathsf{M}^{(n)}_{\mathsf{inst}} \end{array}} \arrow[ul, bend right] & \\
\boxed{\mathsf{P}_{\mathsf{4st}} \sqsubseteq \mathsf{P}_{\mathsf{dec}}} \arrow[u, "\text{[replication]}"] & & & \\
\boxed{\mathsf{P}_{\mathsf{4st}} \sqsubseteq \mathsf{P}_{\mathsf{3st}}} \arrow[u, bend left] & \boxed{\mathsf{P}_{\mathsf{3st}} \sqsubseteq \mathsf{P}_{\mathsf{dec}}} \arrow[ul, bend right, "\text{[transitivity]}"'] & & \boxed{\begin{array}{l} \mathsf{M}^{(n)}_{\mathsf{cache}} \sqsubseteq_{\mathsf{dropPeek}^n} \\ n \cdot \mathsf{MWrap} + \mathsf{M}^{(n)}_{\mathsf{inst}} \end{array}} \arrow[uul, bend right, "\text{[modular refinement]}"'] \\
\boxed{\begin{array}{l} \mathsf{Fetch} + \mathsf{f2d} + \mathsf{Decode} \\ +\mathsf{Exec/Mem} \\ \sqsubseteq \mathsf{FD} + \mathsf{Exec/Mem} \end{array}} \arrow[u, bend right, "\text{[definitions]}"] & \text{[by Section 6.2.1]} & \boxed{\begin{array}{l} n \cdot \mathsf{MRqRsEx} \\ \sqsubseteq_{\mathsf{dropPeek}^n} n \cdot \mathsf{MRqRs} \end{array}} \arrow[uuu, bend left] & \text{[by Theorem 6.1]} \\
\boxed{\mathsf{Fetch} + \mathsf{f2d} + \mathsf{Decode} \sqsubseteq \mathsf{FD}} \arrow[u, "\text{[left substitution]}"] & & \boxed{\mathsf{MRqRsEx} \sqsubseteq_{\mathsf{dropPeek}} \mathsf{MRqRs}} \arrow[u, bend left, "\text{[replication]}"'] & \\
\text{[by Section 6.2.1]} & & \text{[by Theorem 6.3]} &
\end{tikzcd}$$

$$\begin{tikzcd}
(2) \arrow[d, dash] \\
\boxed{n \cdot \mathsf{P}_{\mathsf{dec}} + n \cdot \mathsf{MRqRs} + \mathsf{M}^{(n)}_{\mathsf{atomic}} \sqsubseteq \mathsf{SC}^{(n)}} \\
\boxed{n \cdot \mathsf{P}_{\mathsf{dec}} + n \cdot \mathsf{MRqRs} + n \cdot \mathsf{MWrap} + \mathsf{M}^{(n)}_{\mathsf{inst}} \sqsubseteq n \cdot \mathsf{P}_{\mathsf{inst}} + \mathsf{M}^{(n)}_{\mathsf{inst}}} \arrow[u, "\text{[definitions]}"] \\
\boxed{n \cdot \mathsf{P}_{\mathsf{dec}} + n \cdot \mathsf{MRqRs} + n \cdot \mathsf{MWrap} \sqsubseteq n \cdot \mathsf{P}_{\mathsf{inst}}} \arrow[u, "\text{[left substitution]}"] \\
\boxed{n \cdot (\mathsf{P}_{\mathsf{dec}} + \mathsf{MRqRs} + \mathsf{MWrap}) \sqsubseteq n \cdot \mathsf{P}_{\mathsf{inst}}} \arrow[u, "\text{[associativity and commutativity]}"] \\
\boxed{\mathsf{P}_{\mathsf{dec}} + \mathsf{MRqRs} + \mathsf{MWrap} \sqsubseteq \mathsf{P}_{\mathsf{inst}}} \arrow[u, "\text{[replication]}"] \\
\text{[by Section 6.2.1]}
\end{tikzcd}$$

Fig. 14. Proving the final refinement

multiprocessor + memory example takes about 15,000 lines, including the design of the processor and the memory along with the supporting design library (like queues, etc.) as well as their proofs.

The cache-coherence proof was less automated than the processor proof, because the cache invariants were complex – and it took less time to discharge individual goals manually than wait for the proof automation to complete.

We discovered several bugs in our design while proving it (in both cache coherence and processor). The way the bugs manifested is by creating a proof scenario (by case analysis on states, either manually for cache coherence or automatically for the processor) where the invariants do not hold.

Another aspect of any verification infrastructure is how it deals with missing invariants. In other words, what feedback does the system give when the invariants are weak? Again, the feedback is via creating scenarios that cannot be proven with existing hypotheses. For the cache proof, we worked out all the details before starting the Coq proof, so we did not run into the issue of needing to strengthen invariants.

7 SYNTHESIS AND EVALUATION

We instantiate all the parameters of a Kami module and extract an OCaml program that, when run, computes a Kami abstract syntax tree. We pretty-print that tree as a Bluespec program. The Bluespec program, when passed through a Bluespec compiler, produces a Verilog netlist that can be synthesized into FPGAs or fabricated into chips.

Currently, the entire synthesis process is part of Kami's trusted base. The compilation from Kami to Bluespec is not verified, but it is very straightforward. We also, unsurprisingly, have not verified the commercial tools at the bottom of our compilation pipeline, the Bluespec compiler and the Xilinx toolkit. Braibant and Chlipala [2013] have verified a synthesizer from a stripped-down Bluespec language that does not support modules and method calls and which synthesizes a simple schedule, but for now we use the (unverified) Bluespec compiler because of its superior code generation.

7.1 Synthesizing a RISC-V Multiprocessor

In order to confirm that our processor and memory design is indeed executable, we concretized the abstract ISA into a commonly used subset of RV32I, the 32-bit integer portion of the open-source RISC-V ISA[¶]. The subset consists of all instructions except the ones performing subword memory accesses. We added a new `TOHOST` instruction, which makes the `tohost` call. Concretely, choosing RISC-V requires writing all of the parameter functions (e.g., instruction decoding from binary) that our implementation refers to.

We successfully synthesized and ran the multiprocessor system containing RISC-V cores, each connected to a coherent L1 cache, on Xilinx's Virtex-7 VC707 FPGA. While we verified the multicore system for arbitrary core counts and arbitrary sizes of (direct-mapped) caches and cache lines, on the FPGA, we synthesized a 4-core processor with 32-kilobyte direct-mapped L1 caches, where each line holds 64 bytes.

We compared the results of synthesis of a single processor written in Kami against an independent Bluespec implementation [Wright et al. 2016] of the full RV32I RISC-V ISA (Figure 15). The Bluespec processor does not do any speculative execution and instead fetches and executes nonmemory instructions atomically and executes memory instructions one cycle later, bypassing the results of the load to the next instruction when it is executed. While the execution unit implemented by the Bluespec processor indeed executes a bigger subset of instructions, the resource consumption is almost entirely due to the variable-shift-left-instruction implementation (using a "barrel shifter"). We also implement this instruction, making comparison of synthesis results reasonable. Their

[¶]https://riscv.org/, see [Waterman et al. 2016]

	Kami	Bluespec
$\text{IPC}_{\text{average}}$	0.21	1.00
Clock Frequency (MHz)	142.86	31.25
#LUTs per core	3,061	2,184
#FFs per core	1,545	3,440

Fig. 15. RISC-V cores in Kami and Bluespec

Single-threaded	IPC
Euclid's GCD	0.23
Factorial	0.22
Bubble Sort	0.25
Towers of Hanoi	0.18
Multi-threaded	**IPC**
Shared Counter (Dekker's)	0.19
Shared Counter (Peterson's)	0.20
Parallel Matrix Multiply	0.20
Concurrent Bank Transactions	0.18

Fig. 16. Performance on Kami-based multiprocessor

processor runs at a much lower frequency than ours since they perform both the fetch and execute in the same hardware clock cycle, while ours is a 4-stage pipeline. The resources utilized are comparable ("LUTs" stands for lookup tables, a primitive for combinational circuits in an FPGA, and "FFs" stands for flip-flops, a primitive for state in an FPGA). The synthesis results exclude the L1 caches and the memory from both the processors (the Bluespec processor does not implement cache coherence, as it is a single-core implementation).

We tested our multiprocessor with real RISC-V programs. The benchmark programs were compiled from actual (multithreaded) C programs, using the RISC-V GCC, and the instruction memory is initialized with the executable produced by GCC. We also set the initial stack-pointer value for each processor, since these programs are directly run on our processor, with no intervening OS. Our processor executes an average of 0.21 instructions every clock cycle per processor (IPC). Figure 16 shows the IPC for each benchmark. Note that we instantiated the branch predictor to always predict a branch to be not taken and have not implemented aggressive pipelining to forward results from the execution units and memory as soon as they have been computed. Comparing the instructions executed per second against the Bluespec processor, we are lower only by 4% ($\frac{0.21\text{ IPC } \times 142.86\text{ MHz}}{1\text{ IPC } \times 31.25\text{ MHz}} \approx 96\%$). If we had designed the same processor in both Kami and Bluespec, we would have gotten the same synthesis results.

8 RELATED WORK

Vijayaraghavan's thesis [Vijayaraghavan 2016] forms the theoretical basis for our work.

Hardware verification is dominated by model checking: for instance, processor verification [Burch and Dill 1994; McMillan 1998] and, more recently, Intel's execution cluster verification [Kaivola et al. 2009]. The work of Burch and Dill [1994] is notable for using symbolic execution for automatic computation of an abstraction function of the kind that we so far write manually when applying our Theorem 5.6. Most of model-checking-based verification is done on concrete systems as opposed to parameterized systems. Cache-coherence protocol verification has mostly treated systems with concrete topologies, involving particular finite numbers of caches and processors. For instance, explicit-state model checking tools like Murphi [Dill et al. 1992] or TLC [Joshi et al. 2003; Lamport 2002] can handle only tens of addresses and CPUs, as opposed to the billions of addresses in a real system, or the ever-growing number of CPUs. Symbolic model checking by itself does not do any better: a 2-level MSI protocol in the Gigamax distributed multiprocessor having two clusters with six processors each has been verified using SMV [McMillan and Schwalbe 1992]. SMV had been developed further into Cadence SMV [Jhala and McMillan 2001], which allows compositional model checking. Several processor-design aspects have been verified using Cadence SMV [Lungu and Sorin 2007], though verifications of complete processors were not reported.

There are many abstraction techniques to reduce parameterized designs to finite state spaces, which can be explored exhaustively. Optimizations on symbolic model checking (*e.g.*, partial order reduction [Bhattacharya et al. 2005], symmetry reduction [Bhattacharya et al. 2006; Chen et al. 2010; Chou et al. 2004; Emerson and Kahlon 2003; Ip et al. 1996; Zhang et al. 2014], compositional reasoning [Jhala and McMillan 2001; McMillan 1999, 2001], extended-FSM [Delzanno 2000], *etc.*) further scale the approach. Adopting these techniques still does not fully automate model-checking-based verification of complex systems – all the invariants obeyed by the system must be supplied manually (similarly to the hard problem of deriving loop invariants automatically in software). ARM has published its formal-verification workflow [Reid 2016; Reid et al. 2016], which uses bounded model checking. They also acknowledge (in their CAV'16 paper) that they would need invariants to get proofs of infinite families of designs, while our technique already admits such proofs. Chou et al. [2004]; Matthews et al. [2016]; Talupur and Tuttle [2008]; Zhang et al. [2014, 2010] have all verified cache-coherence protocols using model checking in settings where the number of cores, number of levels in the hierarchy, *etc.* have been parameterized, by relying on paper-and-pencil proofs for properties about compositions and supplying the invariants manually.

Theorem provers have been used to verify hardware designs before, *e.g.,* HOL to verify an academic microprocessor AVI-1 [Windley 1995] or Nqthm to verify the FM9001 microprocessor [Hunt 1989; Hunt and Brock 1992, 1995]. Cache-coherence proofs have also used mechanized theorem provers [Moore 1998; Park and Dill 1996; Vijayaraghavan et al. 2015]. All these approaches, including the ones verified using model checking earlier, verify a model of the actual system, requiring trust in the manual translation from hardware specifications to the model. Moreover, our work is different in providing a general-purpose framework for hardware verification, as opposed to verifying specific types of hardware like processors or cache systems.

Various high-level programming languages have been used to design hardware, where programs are converted into or generate equivalent circuits. One well-known tool in the functional-languages world is Lava [Bjesse et al. 1998], a system to specify, design, and verify hardware in Haskell. It supports recursion but requires the corresponding description to be inlined. Functional programs with recursive definitions have also been successfully converted without inlining [Ghica 2007; Ghica and Smith 2010, 2011; Ghica et al. 2011]. Esterel, a language for reactive systems, has also been used to design hardware and later embedded into HOL [Schneider 2001]. Probably the most recent work in this tradition centers on a HDL and its semantics defined in Agda [Flor and Swierstra 2015], with functional semantics for defining, verifying, and simulating hardware designs. All languages mentioned here, however, are defined based on (or converted to) low-level circuits, which perhaps explains why their case studies were significantly less involved than multicore processors. Our framework is high-level and general enough to prove full functional correctness for realistic multicore processors with cache-coherent memory systems.

9 PRACTICAL ISSUES IN ADOPTION OF KAMI BY INDUSTRY

The hardware design and verification toolflow in industry is based on Verilog/VHDL, whose components are easy to model as external modules that interact with a Kami module through well-defined methods. This factoring allows designs developed and verified using Kami to be used directly in the existing environment catering to RTL-based design and verification. That said, the Bluespec language and its compiler are already very practical for adoption by industry. They are supported by Bluespec, Inc., a commercial company that has existed for more than a decade. With companies looking to protect their competitive advantages, it is tricky to lock down confirmed uses of avant-garde tools in industry. However, from public disclosures, we know that IP blocks in several popular commercial chips have been designed using Bluespec. Examples include a TI display controller and an STM video data mover. (These examples also show the interoperability of Bluespec with other hardware languages.) A variety of other big-name companies including Hitachi, IBM, Intel, Nokia, and Qualcomm have used or continue to use Bluespec for modeling and emulation on FPGAs. Bluespec has also been used for fabricated chips associated with research projects published at the most prestigious venues for circuits [Juvekar et al. 2016; Lis et al. 2013;

Raina et al. 2016]. Since the same design is rarely written in both Bluespec and Verilog, careful studies were performed to show that performance using Bluespec is actually quite competitive and sometimes superior to hand-written Verilog [Arvind et al. 2004]. The scheduler generated by Bluespec exactly embodies the control logic that would have been specified manually in Verilog [Nikhil and Czeck 2010].

Synchronous hardware designs (which is the class of designs obtained by writing in traditional HDLs like Verilog/VHDL, where the whole system performs a huge state transition on every hardware clock tick) are also subsumed in Bluespec/Kami. Such designs can be expressed modularly in Bluespec/Kami by having just methods for communication without rules in each module, with a single top-level rule calling every method of every module. So, designers can mimic their traditional synchronous design styles in Bluespec/Kami, though with more unwieldy proofs.

While all the semantically interesting features are the same between Kami and Bluespec, there are a few practical differences: (1) Bluespec supports constructs that violate one-rule-at-a-time semantics (namely "wires," whose behavior depends on the schedule); and (2) Bluespec, being a separate language by itself, contains special-purpose metaprogramming/generic-programming features, which we replace by using Coq as a metalanguage, since Kami is a DSL inside Coq.

While verification at RTL or higher level is an important component of hardware verification, it does not subsume all the lower-level verification tasks that go into translating hardware specifications into real silicon circuits (e.g., hand optimization of physical layout, timing analysis, register retiming, verifying if the integrated-circuit layout corresponds to the original circuit, etc.). These tasks are orthogonal to designing and verifying hardware systems using RTL-based languages and tools, and the same is true when Kami is the source language.

10 CONCLUSION AND FUTURE WORK

We have developed a fairly complete infrastructure to design, verify, and synthesize hardware systems. We have designed and verified a multiprocessor system containing RISC-V cores connected to a fairly complicated memory system with coherent caches. While we have started developing verified RISC-V components, we plan to scale up the realism level by designing more complex processor optimizations (such as superscalar, out-of-order execution) verified to adhere to the simple instantaneous-processor specification. We plan to implement a verified compiler in Coq to translate Kami modules directly into hardware netlists. Another fruitful extension of Kami would be to support proof of liveness properties like deadlock freedom.

ACKNOWLEDGMENTS

We thank David Dill, Andres Erbsen, Lionel Rieg, and Daniel Ziegler for their feedback on drafts. We also thank Thomas Bourgeat, Andrew Wright, and Sizhuo Zhang, for their help in setting up the FPGA environment for our synthesis results; and Benjamin Delaware and Christian J. Bell, for help integrating their library code.

This work was supported in part by National Science Foundation awards CCF-1253229 and CCF-1521584 (the latter for the *Science of Deep Specification* Expedition). Partial support also came from the DARPA BRASS program, a Lincoln Labs grant, Intel's University-Industry Collaboration grants, Samsung GRO grants, and Arvind's discretionary research funds. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. 2004. High-level synthesis: an essential ingredient for designing complex ASICs. In *2004 International Conference on Computer-Aided Design, ICCAD 2004, San Jose, CA, USA, November 7-11, 2004.* IEEE Computer Society / ACM, 775–782. https://doi.org/10.1109/ICCAD.2004.1382681

Ritwik Bhattacharya, Steven German, and Ganesh Gopalakrishnan. 2005. Symbolic Partial Order Reduction for Rule Based Transition Systems. In *Correct Hardware Design and Verification Methods*, Dominique Borrione and Wolfgang Paul (Eds.). Lecture Notes in Computer Science, Vol. 3725. Springer Berlin Heidelberg, 332–335. https://doi.org/10.1007/11560548_25

Ritwik Bhattacharya, Steven M. German, and Ganesh Gopalakrishnan. 2006. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *Antti Valmari, editor, SPIN, volume 3925 of Lecture Notes in Computer Science.* Springer, 252–270.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98).* ACM, New York, NY, USA, 174–184. https://doi.org/10.1145/289423.289440

Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *CAV 2013, 25th International Conference on Computer Aided Verification (Lecture Notes in Computer Science)*, Vol. 8044. Springer, 213–228. http://gallium.inria.fr/~braibant/fe-si/

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

Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. 2010. Efficient Methods for Formally Verifying Safety Properties of Hierarchical Cache Coherence Protocols. *Form. Methods Syst. Des.* 36, 1 (Feb. 2010), 37–64. https://doi.org/10.1007/s10703-010-0092-y

Adam Chlipala. 2008. Parametric Higher-order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08).* ACM, New York, NY, USA, 143–156. https://doi.org/10.1145/1411204.1411226

Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. 2004. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer Aided Design.* Springer, 382–398.

Nirav Dave, Arvind, and Michael Pellauer. 2007. Scheduling as Rule Composition. In *5th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2007), May 30 - June 1st, Nice, France.* IEEE Computer Society, 51–60. https://doi.org/10.1109/MEMCOD.2007.371249

Nirav Dave, Man Cheuk Ng, and Arvind. 2005. Automatic synthesis of cache-coherence protocol processors using Bluespec. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings.* IEEE Computer Society, 25–34. https://doi.org/10.1109/MEMCOD.2005.1487887

Giorgio Delzanno. 2000. Automatic Verification of Parameterized Cache Coherence Protocols. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Lecture Notes in Computer Science, Vol. 1855. Springer Berlin Heidelberg, 53–68. https://doi.org/10.1007/10722167_8

D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. 1992. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on.* 522–525. https://doi.org/10.1109/ICCD.1992.276232

E. Allen Emerson and Vineet Kahlon. 2003. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings.* 247–262. https://doi.org/10.1007/978-3-540-39724-3_22

Thomas M Esposito, Mieszko Lis, Ravi A Nanavati, Joseph E Stoy, and Jacob B Schwartz. 2010. System and method for scheduling TRS rules. (Jan. 12 2010). US Patent 7,647,567.

Joao Paulo Pizani Flor and Wouter Swierstra. 2015. Π-Ware: An Embedded Hardware Description Language using Dependent Types. *TYPES 2015* (2015), 67.

Dan R. Ghica. 2007. Geometry of Synthesis: A Structured Approach to VLSI Design. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07).* ACM, New York, NY, USA, 363–375. https://doi.org/10.1145/1190216.1190269

Dan R. Ghica and Alex Smith. 2010. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits. *Electron. Notes Theor. Comput. Sci.* 265 (Sept. 2010), 301–324. https://doi.org/10.1016/j.entcs.2010.08.018

Dan R. Ghica and Alex Smith. 2011. Geometry of Synthesis III: Resource Management Through Type Inference. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11).* ACM, New York, NY, USA, 345–356. https://doi.org/10.1145/1926385.1926425

Dan R. Ghica, Alex Smith, and Satnam Singh. 2011. Geometry of Synthesis IV: Compiling Affine Recursion into Static Hardware. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11).* ACM, New York, NY, USA, 221–233. https://doi.org/10.1145/2034773.2034805

James C. Hoe and Arvind. 2000. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000*, Ellen Sentovich (Ed.). IEEE Computer Society, 511–518. https://doi.org/10.1109/ICCAD.2000.896524

Warren A. Hunt. 1989. Microprocessor design verification. *Journal of Automated Reasoning* 5, 4 (1989), 429–460. https://doi.org/10.1007/BF00243132

Warren A. Hunt and Bishop C. Brock. 1992. A Formal HDL and its Use in the FM9001 Verification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 339, 1652 (1992), 35–47. https://doi.org/10.1098/rsta.1992.0024 arXiv:http://rsta.royalsocietypublishing.org/content/339/1652/35.full.pdf

W. A. Hunt and B. C. Brock. 1995. The DUAL-EVAL hardware description language and its use in the formal specification and verification of the FM9001 microprocessor. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal.* 637–642. https://doi.org/10.1109/ASPDAC.1995.486381

Chung-Wah Norris Ip, David L. Dill, and John C. Mitchell. 1996. State Reduction Methods For Automatic Formal Verification. (1996).

Ranjit Jhala and Kenneth L. McMillan. 2001. Microarchitecture Verification by Compositional Model Checking. In *Computer Aided Verification: 13th International Conference, Paris, France, July 18–22, 2001 Proceedings*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, Berlin, Heidelberg, 396–410. https://doi.org/10.1007/3-540-44585-4_40

Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. 2003. Checking Cache-Coherence Protocols with TLA$^+$. *Formal Methods in System Design* 22, 2 (2003), 125–131. https://doi.org/10.1023/A:1022969405325

Chiraag Juvekar, Hyung-Min Lee, Joyce Kwong, and Anantha P. Chandrakasan. 2016. A Keccak-based wireless authentication tag with per-query key update and power-glitch attack countermeasures. In *2016 IEEE International Solid-State Circuits Conference, ISSCC 2016, San Francisco, CA, USA, January 31 - February 4, 2016*. IEEE, 290–291. https://doi.org/10.1109/ISSCC.2016.7418021

Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. 2009. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. In *Computer Aided Verification*. Springer, 414–429.

Michal Karczmarek, Arvind, and Muralidaran Vijayaraghavan. 2014. A new synthesis procedure for atomic rules containing multi-cycle function blocks. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*. IEEE, 22–31. https://doi.org/10.1109/MEMCOD.2014.6961840

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. SOSP*. ACM, 207–220.

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on* 100, 9 (1979), 690–691.

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*. ACM, 42–54.

Mieszko Lis, Keun Sup Shim, Brandon Cho, Ilia Lebedev, and Srinivas Devadas. 2013. Hardware-level thread migration in a 110-core shared-memory multiprocessor. In *Hot Chips 25 Symposium (HCS), 2013 IEEE*. IEEE, 1–27.

Anita Lungu and Daniel J. Sorin. 2007. Verification-Aware Microprocessor Design. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, Washington, DC, USA, 83–93. https://doi.org/10.1109/PACT.2007.79

Opeoluwa Matthews, Jesse D. Bingham, and Daniel J. Sorin. 2016. Verifiable hierarchical protocols with network invariants on parametric systems. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 101–108. https://doi.org/10.1109/FMCAD.2016.7886667

K.L. McMillan. 1999. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods*, Laurence Pierre and Thomas Kropf (Eds.). Lecture Notes in Computer Science, Vol. 1703. Springer Berlin Heidelberg, 219–237. https://doi.org/10.1007/3-540-48153-2_17

K.L. McMillan and James Schwalbe. 1992. Formal verification of the Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*. 111–134.

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

K. L. McMillan. 2001. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144*. Springer, 179–195.

J Strother Moore. 1998. An ACL2 Proof of Write Invalidate Cache Coherence. In *Proc. CAV'98, volume 1427 of LNCS*. Springer, 29–38.

Rishiyur S Nikhil and Kathy R Czeck. 2010. BSV by Example. *CreateSpace, Dec* (2010).

Seungjoon Park and David L. Dill. 1996. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, 288–296.

Priyanka Raina, Mehul Tikekar, and Anantha P. Chandrakasan. 2016. An energy-scalable accelerator for blind image deblurring. In *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference, Lausanne, Switzerland, September 12-15, 2016*. IEEE, 113–116. https://doi.org/10.1109/ESSCIRC.2016.7598255

Alastair Reid. 2016. Trustworthy Specifications of ARMR v8-A and v8-M System Level Architecture. In *Formal Methods in Computer-Aided Design, FMCAD*.

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-Formal. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9780. Springer, 42–58. https://doi.org/10.1007/978-3-319-41540-6_3

Daniel L. Rosenband and Arvind. 2004. Modular scheduling of guarded atomic actions. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, Sharad Malik, Limor Fix, and Andrew B. Kahng (Eds.). ACM, 55–60. https://doi.org/10.1145/996566.996583

Klaus Schneider. 2001. *A Verified Hardware Synthesis of Esterel Programs*. Springer US, Boston, MA, 205–214. https://doi.org/10.1007/978-0-387-35409-5_20

Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture* 6, 3 (2011), 1–212. https://doi.org/10.2200/S00346ED1V01Y201104CAC016 arXiv:http://dx.doi.org/10.2200/S00346ED1V01Y201104CAC016

M. Talupur and Mark R. Tuttle. 2008. Going with the Flow: Parameterized Verification Using Message Flows. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08*. 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.14

Muralidaran Vijayaraghavan. 2016. *Modular Verification of Hardware Systems*. Ph.D. Dissertation. http://hdl.handle.net/1721.1/106096

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)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 109–127. https://doi.org/10.1007/978-3-319-21668-3_7

Muralidaran Vijayaraghavan, Nirav Dave, and Arvind. 2013. Modular compilation of guarded atomic actions. In *11th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMCODE 2013, Portland, OR, USA, October 18-20, 2013*. IEEE, 177–188. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6670957

Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2016. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*. Technical Report UCB/EECS-2016-118. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

Phillip J. Windley. 1995. Formal modeling and verification of microprocessors. *Computers, IEEE Transactions on* 44, 1 (1995), 54–72.

Andy Wright, Sizhuo Zhang, Thomas Bourgeat, Muralidaran Vijayaraghavan, Jamey Hicks, and Arvind. 2016. Riscy Processors: A collection of open-sourced RISC-V processors. In *4th RISC-V Workshop*. https://riscv.org/wp-content/uploads/2016/07/Wed830_Riscy_Processors_V1.pdf

Meng Zhang, Jesse D. Bingham, John Erickson, and Daniel J. Sorin. 2014. PVCoherence: Designing flat coherence protocols for scalable verification. In *20th IEEE International Symposium on High Performance Computer Architecture, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 392–403. https://doi.org/10.1109/HPCA.2014.6835949

Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. 2010. Fractal Coherence: Scalably Verifiable Cache Coherence. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 471–482. https://doi.org/10.1109/MICRO.2010.11