PEak: A Single Source of Truth for Hardware Design and Verification

CALEB DONOVICK, Computer Science, Stanford University, Stanford, United States
JACKSON MELCHERT, Electrical Engineering, Stanford University, Stanford, United States
ROSS DALY, Computer Science, Stanford University, Stanford, United States
LENNY TRUONG, Computer Science, Stanford University, Stanford, United States
PRIYANKA RAINA, Electrical Engineering, Stanford University, Stanford, United States
PAT HANRAHAN, Computer Science, Stanford University, Stanford, United States
CLARK BARRETT, Computer Science, Stanford University, Stanford, United States

Domain-specific languages for hardware can significantly enhance designer productivity, but sometimes at the cost of ease of verification. On the other hand, ISA specification languages are too static to be used during early stage design space exploration. We present PEak, an open-source hardware design and specification language, which aims to improve both design productivity and verification capability. PEak does this by providing a single source of truth for functional models, formal specifications, and RTL. PEak has been used in several academic projects, and PEak-generated RTL has been included in three fabricated hardware accelerators. In these projects, the formal capabilities of PEak were crucial for enabling both novel design space exploration techniques and automated compiler synthesis.

CCS Concepts: • **Hardware** → **Functional verification**; **Hardware description languages and compilation**.

Additional Key Words and Phrases: PEak, domain-specific languages, hardware design, formal methods

1 Introduction

Domain-specific languages (DSLs) for hardware allow designers to build generators that are impossible to express using traditional hardware description languages such as SystemVerilog and VHDL [3, 40]. Such generators are of increasing importance as specialized chips become the norm in a post-Dennard-scaling world [24, 39]. DSLs can also provide better correctness guarantees through type safety (a well-known pain point in Verilog). These factors have led to an explosion of new DSLs for hardware design over the last decade [3, 16, 26, 31, 40].

Unfortunately, the design of most hardware DSLs has not sufficiently taken into account the impact on verification [30]. For example, using a Verilog simulator to debug DSL-generated designs is notoriously difficult, as information is lost or obscured during the compilation process. A first step towards addressing this challenge is to include support for writing properties that can be translated to SystemVerilog assertions (SVAs), and indeed several languages provide this (e.g., Chisel [15] and Magma [41]). More ambitious efforts aim to enable source-level

Authors' Contact Information: Caleb Donovick, Computer Science, Stanford University, Stanford, California, United States; e-mail: donovick@cs.stanford.edu; Jackson Melchert, Electrical Engineering, Stanford University, Stanford, California, United States; e-mail: melchert@stanford.edu; Ross Daly, Computer Science, Stanford University, Stanford, California, United States; e-mail: rdaly525@cs.stanford.edu; Lenny Truong, Computer Science, Stanford University, Stanford, California, United States; e-mail: lenny@cs.stanford.edu; Priyanka Raina, Electrical Engineering, Stanford University, Stanford, California, United States; e-mail: praina@stanford.edu; Pat Hanrahan, Computer Science, Stanford University, Stanford, California, United States; e-mail: hanrahan@cs.stanford.edu; Clark Barrett, Computer Science, Stanford University, Stanford, California, United States; e-mail: barrett@cs.stanford.edu.

ACM 1558-3465/2024/11-ART
https://doi.org/10.1145/3703456

debugging [42], which will likely be crucial for effective debugging of generated RTL, especially at later design stages.

On the other hand, DSL models are well-positioned to dramatically improve the *early-stage* verification experience. In particular, they can be leveraged to greatly improve debugging and verification during design space exploration (DSE). Traditionally, separate functional models play a key role during this phase, but a promising alternative supported by some DSLs (e.g., pyMTL [31]) is to automatically extract a high-performance executable functional model from a DSL description. Moreover, with the right semantics and support, the user can even be provided with direct access to an automatically-generated *formal* model for the design, enabling novel and early uses of formal methods during the design exploration process. Current DSLs provide very limited support for such features.

In this paper, we introduce PEak, a Python-embedded DSL, with an accompanying set of open-source tools, including a compiler. PEak provides a *single source of truth* for compilation to RTL, functional simulation, and formal modeling. Designers who use PEak do not need to implement the same thing multiple times, and the different implementations are guaranteed to be consistent with each other. Furthermore, these capabilities directly enable novel formal-in-the-loop design methodologies.

PEak is partly motivated by work being done at the Stanford Agile Hardware center [4, 11, 17, 27],[1] where it has been used to generate coarse-grained reconfigurable array (CGRA) architectures[2] for three generations of chips, two of which were fabricated. Section 4 explains how the formal model generated by PEak was used to synthesize compiler components for different candidate architectures, thereby enabling a systematic and automatic exploration of the design space.

The rest of this paper is organized into the following sections: Section 2 describe `hwtypes` and `ast_tools` which PEak is built on; Section 3 describes the PEak language and how it can be extended; and Section 4 evaluates PEak as a tool for DSE, showing it can generate both high performance RTL as well as SMT models which are usable in a formal-in-the-loop design flow. We discuss related work and conclude in Sections 5 and 6, respectively.

2 Hardware Types and AST-Tools

We first introduce two libraries we developed which serve as the foundation of PEak: `hwtypes` and `ast_tools`. `hwtypes`[3] serves as both the type system and compilation target for PEak. `ast_tools`[4] is used for Python abstract syntax tree (AST) analysis and rewriting, which is used both to build the PEak compiler and to extend PEak's meta-programming facilities. These libraries are independent of PEak, and may be of interest on their own.

The interplay between PEak, `ast_tools`, and `hwtypes` is illustrated in Figure 1. A PEak specification is the input to the PEak compiler. The PEak specification uses the `hwtypes` type system for things like `Bit`, `BitVector`, and algebraic data types. The PEak compiler uses `ast_tools`, first to transform the Python AST of the PEak specification, and then again to generate the final compiled specification in the `hwtypes` expression language. In the following subsections, we describe `hwtypes` and `ast_tools` in detail.

2.1 Hardware Types

The core of PEak is the Python-embedded expression language of `hwtypes`. `hwtypes` provides a uniform interface for: functional simulation, via direct execution in Python; formal analysis, via automatic translation to formulas in the language of satisfiability modulo theories (SMT) [6]; and RTL generation, via a compiler to Magma [41]. By unifying these types we ensure the equivalence of the generated functional, formal, and RTL models.

[1] aha.stanford.edu

[2] CGRAs [22, 32, 35] are a spatial architecture similar to FPGAs and are composed of processing element (PE) and memory tiles, and a configurable routing network.

[3] https://github.com/leonardt/hwtypes

[4] https://github.com/leonardt/ast_tools

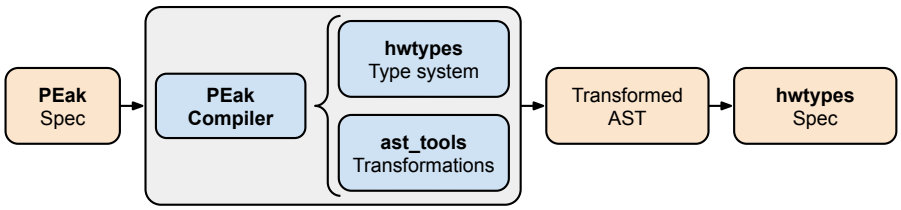

Fig. 1. PEak, `ast_tools`, and `hwtypes` transform a PEak specification into a compiled `hwtypes` program.

`hwtypes` defines abstract interfaces and type constructors for a number of types and kinds. This includes a `Bit` (Boolean) type, fixed-width `BitVector` types (signed and unsigned), arbitrary-precision floating-point types, and algebraic data types (ADTs). We first focus on the `Bit` and `BitVector` types (we discuss the use of ADTs in Section 3.2). `Bit` type provides the usual Boolean operators: and &, or `|`, xor `^`, and not `~`; equality operators: equals ==, and not equals `!=`; and an `ite` (if-then-else) method.

The SMT-LIB standard [5] defines a large set of arithmetic and bitwise functions on bitvectors. The `hwtypes` `BitVector` interface defines a method for each of these functions. For instance, the equivalent of the SMT-LIB term `(bvadd x y)` (bitvector addition), where `x` and `y` are of sort `(_ BitVec 16)`, or 16-bit bitvectors, is the `hwtypes` expression `x.bvadd(y)`, where `x` and `y` are of the type `BitVector[16]`. More generally, if `f` is a function over bitvectors defined by SMT-LIB, then there is an equivalent method named `f` on the `hwtypes` `BitVector` type. As a convenience, these methods are also defined by overloading Python operators when appropriate. For example: `x.bvadd(y)` can be invoked with `x + y`. The semantics of sign-dependent operators are defined by their type. For example, `x < y` invokes `x.bvslt(y)` (signed less than) for signed `x` and `x.bvult(y)` (unsigned less than) for unsigned `x`.

There are three implementations of the `BitVector` and `Bit` types. The first implementation is a pure Python functional model over constant values. The second wraps pySMT [20] to generate SMT terms. Finally, Magma provides a third implementation which allows for the definition of circuits. This uniform interface allows for the same `hwtypes` program to be interpreted in multiple ways. The pure Python implementation is used to simulate a circuit, the SMT implementation is used to generate a formal model, and the Magma implementation is used to generate actual RTL.

The real power of `hwtypes` comes from its embedding in Python which facilitates the generation of complex formulas. For example, we can generate an adder tree over any number of inputs with the use of a recursive function as shown in Example 2.1. This can be easily generalized to perform reduction over any function as shown in Example 2.2.

It is important to note that `hwtypes` is an expression language only; all statements are executed in pure Python following typical Python semantics. This is in contrast to PEak (see Section 3, below), which breaks away from the semantics of pure Python and reinterprets the meaning of if statements as `ite`s using AST rewriting.

2.2 AST Tools

In order to be able to reinterpret Python code, we developed the `ast_tools` library, which provides a generic infrastructure for composing passes that analyze and transform the Python abstract syntax tree (AST). The design is the result of our experience developing ad hoc AST rewrites for various DSLs (including PEak) and recognizing the need for a common infrastructure to serve these languages.

2.2.1 Pass Architecture. The entry point to the `ast_tools` library is the `apply_passes` function, which takes a list of passes to run and returns a decorator that is used to transform a function or class. The `apply_passes`

```
def tadd(*args):
  n = len(args)
  if n == 0:
    return 0
  elif n == 1:
    return args[0]
  else:
    left = tadd(*args[:n//2])
    right = tadd(*args[n//2:])
    return left + right
```

Example 2.1: hwtypes adder tree generator.

```
def treduce(f, ident, *args):
  n = len(args)
  if n == 0:
    return ident
  elif n == 1:
    return args[0]
  else:
    largs = args[:n//2]
    rargs = args[n//2:]
    l = treduce(f,ident,*largs)
    r = treduce(f,ident,*rargs)
    return f(l, r)
```

Example 2.2: hwtypes reduction tree generator.

```
@apply_passes([loop_unroll()])
def foo():
  for i in unroll([1,3,9]):
    print(i)

def foo():
  print(1)
  print(3)
  print(9)
```

Example 2.3: Code with loop unrolling applied.

function provides a generic prologue and epilogue, which handles logic common to most code transformers. The prologue parses the marked code into an AST and captures a closure of the environment. The epilogue serializes the transformed AST into code and executes it using the captured environment. Passes use a generic interface that consumes as arguments the current AST, the current environment, and a metadata dictionary. A pass may modify any or all of these and return them as results to be used for the next pass or for the epilogue.

In addition to the pass infrastructure, `ast_tools` provides several useful utilities such as the ability to generate a *free* name in the environment, which allows new variables to be introduced without clobbering existing mappings. It also includes a collection of generic transformation and visitor passes that perform common operations.

2.2.2 Macros. The macro sub-package provides a simple mechanism for performing syntactic rewrites of the Python AST. When an explicit macro identifier is encountered, such as `unroll` in Example 2.3, the corresponding transformation is invoked (`loop_unroll`). PEak employs the macro pattern to allow staged expansion of the specification. For example, `if` statements marked as macros will be evaluated before they are compiled, allowing the user to distinguish between conditional logic intended to describe the generation of the specification versus conditional logic intended to be part of the specification.

3 PEak

The high-level aim of PEak is to provide a natural object-oriented view of hardware, in which a circuit is defined as a Python class. PEak circuits declare sub-components in their `__init__` method[5] and define their behavior in their `__call__` method.[6] A circuit's inputs are the arguments to its call method and its outputs are the return values of the method. Sub-components are included simply by calling them as functions.

The underlying semantics of PEak is a synchronous hardware model that uses an implicit clock and implicit wiring. In a PEak program with state, a single call to the `__call__` method represents one clock cycle, updating any state variables that have been declared in the `__init__` method. The goal of PEak is to make writing hardware easier through a natural object-oriented view of the hardware. Therefore, PEak works best for specifying hardware that can be encapsulated into well-defined modules with instructions, inputs, and outputs. Due to the implicit clocking and wiring in PEak, designs with multiple clocks or combinational loops cannot be expressed.

In Example 3.1, we show a small example of PEak code. Code points of interest have been annotated with `# n`. We start by explaining `ALU` (`# 4`) and `RegALU` (`# 7`); then, in Section 3.3 we discuss the remaining code points. The `ALU` class performs either an add or a multiply on two data inputs (`in_0`, `in_1`) and is controlled by a single bit `op`. We show the results of compiling this ALU to Verilog using the MLIR [29] backend to Magma in Example 3.2.

The `RegALU` class instantiates an `ALU` and two `Registers`. `RegALU` is controlled by a two-bit signal `instr`, where bit 0 is the `ALU` op and bit 1 is an `acc` flag. `RegALU` passes the contents of its registers to the `ALU` and outputs the `ALU`'s output. When the `acc` flag is set, it stores its output in `reg_0`; otherwise, it stores the first input. The observant reader will note that the registers in Example 3.1 are not called as functions. Instead, they are simply read and written as instance attributes. We provide this syntax to allow registers' next state to be dependent on current state (which is impossible with the `__call__` syntax).

3.1 PEak Normal Form

The `ast_tools` library is used to convert a PEak program to a `hwtypes` program. This is achieved by first performing a typical single static assignment (SSA) transformation [38], i.e., introducing unique variables for every assignment and replacing control flow with `phi` statements. Next, all `return` statements are replaced with assignments to fresh identifiers. Next, the bodies of `if` blocks are inlined into their enclosing blocks, and phi nodes are replaced with `ite` calls (a method on the primitive type `Bit`). Finally, we construct the return value by reconstructing the condition structure in a nested `ite`. In this form, the program is a pure `hwtypes` program. The transformed PEak code for `ALU.__call__` in Example 3.1 is shown in Example 3.3.

Special care is needed to handle attribute writes (e.g., registers) as they do not behave like other names. At a high level, the compiler simply generates a fresh name for each written attribute which is initialized at the top of

[5] `__init__` is the standard initializer method in Python, which is similar to but not quite equivalent to a constructor in C++. A more thorough explanation can be found in the Python reference manual [19].

[6] `__call__` overloads the function call syntax, i.e., `foo(args)` ≡ `foo.__call__(args)`.

```
@family_closure(family_group) # 1
def gen(family): # 2
  BV = family.BitVector
  T = BV[8]
  Bit = family.Bit
  Register = family.gen_register(T, 0)

  @family.compile(locals(), globals()) # 3
  class ALU(Peak): # 4
    def __call__(self,
        op: Bit, in_0: T, in_1: T) -> T: # 5
      if op:
        return in_0 + in_1
      else:
        return in_0 * in_1

  @family.compile(locals(), globals()) # 6
  class RegALU(Peak): # 7
    def __init__(self):
      self.alu = ALU()
      self.reg_0 = Register()
      self.reg_1 = Register()

    def __call__(self,
        instr: BV[2], in_0: T, in_1: T) -> T:
      op = instr[0]
      acc = instr[1]
      out = self.alu(
        op, self.reg_0, self.reg_1
      )
      if acc:
        self.reg_0 = out
      else:
        self.reg_0 = in_0
      self.reg_1 = in_1
      return out
  return RegALU
```

Example 3.1: PEak code for ALU.

the program. Next, it replaces all references to the attribute with references to the fresh name. Finally, it writes the generated name back to the attribute at the end of the program.

The existence of multiple returns complicates this basic scheme, as there are multiple "ends" of the program. Hence, at each return location, the state of each attribute (i.e., the value held in the attribute's associated name) is stored in a "final" name, so that the proper value may be written to the attribute at end of the program. Then, at the

```
module ALU(
  input op,
  input [7:0] in_0, in_1,
  input CLK, ASYNCRESET,
  output [7:0] O
  );

  wire [1:0][7:0] _GEN = {
    {in_0 + in_1}, {in_0 * in_1}
  };
  assign O = _GEN[op];
endmodule
```

Example 3.2: ALU compiled to Verilog using the MLIR backend of Magma.

```
class ALU(Peak):
  def __call__(self,
      op: Bit, in_0: T, in_1: T) -> T:
    cond_0 = op
    r_val_0 = in_0 + in_1
    r_val_1 = in_0 * in_1
    r_val_f = cond_0.ite(r_val_0, r_val_1)
    return r_val_f
```

Example 3.3: ALU in PEak normal form as generated by the compiler modulo a slight simplification of generated names.

end of the program the final names are multiplexed, in a similar matter to the rebuilding of return values, before being written back. We show the transformation of the simple counter shown in Example 3.4 in Example 3.5.

3.2 Algebraic Data Types

PEak also supports algebraic data types (ADTs). As ADTs in PEak must be realizable in hardware, we limit them to finite (non-recursive) types. Beyond the usual benefits of abstraction and type safety, ADTs provide a natural abstraction for ISAs: a sum type can be used to specify categories of instructions with different layouts; and product types can used to define the fields of each layout. Example 3.1 uses a single bit to control its operation. However, by doing so we fix the encoding of the ISA. Instead, designers can define ISAs as ADTs as shown in Example 3.6.

Using ADTs to represent ISAs has two main benefits. First, it allows the decode logic to be modified without modifying the functional specification (i.e., the `__call__` method). For instance, to change the encoding of an ADD instruction from `op == 1` to `op == 0` in the original example (3.1), we would need to update the line `if op:` to `if ~op:`. In contrast, in Example 3.6, we just need to change the definition of `AluOp`. While these two edits are of similar complexity for the toy examples shown here, the ADT-based specification is much more maintainable for more complex examples, as most of the complexity tends to lie in the `__call__` method. The second main benefit of using ADTs to describe ISAs is type safety. In the original example, it would be possible for a designer to accidentally use bit 0 as the `acc` flag and bit 1 as the `op`. In contrast, comparing a member of `AluOp` to a member of `RegCtrl` would lead to a type error.

```
@family.compile(locals(), globals())
class Counter(Peak):
  def __init__(self):
    self.reg = Register()

  def __call__(self, en: Bit, rst: Bit) -> T:
    if rst:
      self.reg = T(0)
      return T(0)

    if en:
      state = self.reg
      if state < MAX_COUNT - 1:
        next_state = state + 1
      else:
        next_state = T(0)
      self.reg = next_state
      return state
    else:
      return self.reg
```

Example 3.4: A counter with a reset and enable.

When ADTs are compiled to hardware, they must be encoded as bitvectors. While PEak provides reasonable defaults for the encoding (e.g., `Product` types encoded as the concatenation of their fields), a designer may desire a specific bit-level encoding. PEak provides a simple interface to allow this.

3.3 PEak Internals and Extensions

We now explain the remaining code points in Example 3.1. We highlight a few simple requirements: PEak classes must inherit from the `Peak` class (`# 4` and `# 7`), and the type annotations in the `__call__` method (`# 5`) are *not* optional, as they are needed to generate ports in a Magma context.

Code point `# 2` constructs a closure around the `ALU` and `RegALU` classes. It takes a single argument, which is a *family* object. The family mechanism is the means by which the different interpretations (Python, SMT, Magma) for the same PEak code are provided. Each family object contains one set of implementations for the primitives used by the constructed module (minimally: `Bit`, `BitVector`, ADTs, registers). Note how all types are accessed through the `family` object. `family.compile` (`# 3` and `# 6`) invokes the PEak compiler, passing the current symbols to the compiler with `locals(), globals()`. Each family can define its own compilation flow. For example, the SMT and Magma families rewrite `__call__` code into the PEak normal form.

Finally, the `family_closure` decorator (`# 1`) takes a single parameter, which associates the decorated closure with a specific *family group*, an object (typically a module) with attributes `PyFamily`, `SMTFamily`, and `MagmaFamily`, providing families with the Python, SMT, and Magma interpretations, respectively. Default implementations for each family can be obtained by using a specific family group that is included with PEak. The purpose of an explicit family group parameter is to allow extensions beyond this default implementation. For example, an extended family group could include a floating point type which wraps verilog IP in a Magma context,

```
def __call__(self, en: Bit, rst: Bit) -> T:
  self_reg_0 = self.reg
  cond_0 = rst
  self_reg_1 = T(0)
  self_reg_f_0 = self_reg_1
  r_val_0 = T(0)
  cond_2 = en
  state_0 = self_reg_0
  cond_1 = state_0 < MAX_COUNT - 1
  next_state_0 = state_0 + 1
  next_state_1 = T(0)
  next_state_2 = cond_1.ite(
    next_state_0, next_state_1
  )
  self_reg_2 = next_state_2
  self_reg_f_1 = self_reg_2
  r_val_1 = state_0
  self_reg_f_2 = self_reg_0
  r_val_2 = self_reg_0
  self_reg_f = cond_0.ite(
    self_reg_f_0,
    cond_2.ite(self_reg_f_1, self_reg_f_2)
  )
  self.reg = self_reg_f
  r_val_f = cond_0.ite(
    r_val_0,
    cond_2.ite(r_val_1, r_val_2)
  )
  return r_val_f
```

Example 3.5: A counter in PEak normal form as generated by the compiler. The names have been simplified and additional line breaks have been inserted to increase readability.

uses the hwtypes floating point type in a Python context, and constructs an uninterpreted function in an SMT context.

3.4 Verification and Testing of PEak Circuits

Verification is a complex task, and thorough verification of a hardware design often takes more time and resources than are required to design it in the first place. One of the goals of PEak is to simplify functional testing and democratize formal verification by making the experience nearly equivalent to writing functional tests. Functional testing is made easier by raising the level of abstraction compared to Verilog testbenches and by providing several useful helper functions to easily generate test vectors. Writing a functional testbench in PEak is as straightforward as instantiating a PEak class, calling the PEak object with some instruction and inputs, and checking that the outputs are correct. These features make PEak testbenches much simpler and easier to write than a conventional Verilog testbench. Formal verification is also much easier thanks to the formal interpretation feature of PEak. A

```
class AluOp(Enum):
  ADD = 1
  MUL = 0

class RegCtrl(Enum):
  ACC = 1
  BYPASS = 0

class Inst(Product):
  op = AluOp
  ctrl = RegCtrl

...
@family.compile(locals(), globals())
class ALU(Peak):
  def __call__(self,
      op: AluOp, in_0: T, in_1: T) -> T:
    if op == AluOp.Add:
      return in_0 + in_1
    else:
      return in_0 * in_1

@family.compile(locals(), globals())
class RegALU(Peak):
  def __init__(self):
    self.alu = ALU()
    self.reg_0 = Register()
    self.reg_1 = Register()

  def __call__(self,
      instr: Inst, in_0: T, in_1: T) -> T:
    out = self.alu(
      instr.op, self.reg_0, self.reg_1
    )
    if instr.ctrl == RegCtrl.ACC:
      self.reg_0 = out
    else:
      self.reg_0 = in_0
    self.reg_1 = in_1
    return out
```

Example 3.6: Defining an ISA as an ADT.

```
py_alu = gen.Py()
# iterate over all possible instructions
for alu_op in (AluOp.ADD, AluOp.MUL):
  for reg_mode in RegCtrl.field_dict.values():
    # set initial state to random
    py_alu.reg_0 = random_bv(8)
    py_alu.reg_1 = random_bv(8)
    # use random input variables
    i0 = random_bv(8)
    i1 = random_bv(8)
    instr = Inst(alu_op, reg_mode)
    out = py_alu(instr, i0, i1)
    post_condition = py_alu.reg_1 == i1
    assert post_condition
```

Example 3.7: Random testing of a PEak circuit.

functional testbench can be converted into a formal verification check simply by using the formal interpretation and using SMT `Bit` and `BitVector` types.

As an example of both functional and formal verification, we check whether the code in Example 3.6 always writes its second input to `reg_1`, first using random testing then using formal verification. In Example 3.7, a Python instance of the `ALU` is instantiated. Next, all possible instructions are exhaustively generated by iterating over all values of `AluOp` and `RegCtrl`.[7] Then, the registers are set to random initial states, and random inputs are passed to the `ALU`. Finally, we assert the postcondition that `reg_1` contains the value of `i1`.

In Example 3.8 we show the formal verification of this property which is similar to the random test. First, free SMT variables for the initial state, inputs, and instruction are constructed. Then, we set the initial state and execute the circuit. Finally, we use CVC4 [7] via pySMT to formally verify that `reg_1` contains the value of `i1` by asserting the negation of the property.

The design of PEak makes it extremely natural to specify and verify hardware. The choice to embed PEak in Python means that hardware designers familiar with Python can start writing PEak almost immediately. The choice to use implicit wiring and clocking means that the designer no longer needs to worry about low-level details and raises the level of abstraction to an appropriate level for the types of applications targeted by PEak. Enabling an object-oriented view of hardware design makes reuse of common sub-components and smaller circuit building blocks simple. The strong support for ADT types lends itself very well to the specifications of instructions for hardware-like processors, simplifying the specification and thereby reducing the risk of introducing bugs. The access to the AST enables designers to extend PEak with powerful transformations, which enable higher design productivity. Finally, the multiple interpretations of each PEak specification not only make designing and verifying circuits easier, but also enable powerful techniques like rewrite rule synthesis, which we discuss in Section 4.2.

[7]The inner-loop uses the `field_dict` attribute of the `RegCtrl` type which returns a `dict` (mapping type) of names to enum members allowing programmatic generation of such tests.

```
initial_reg_0 = SMTBitVector[8]()
initial_reg_1 = SMTBitVector[8]()
i0 = SMTBitVector[8]()
i1 = SMTBitVector[8]()
instr = make_symbolic(Inst)

smt_alu = gen.SMT()
# set the initial state to be symbolic
smt_alu.reg_0 = initial_reg_0
smt_alu.reg_1 = initial_reg_1
# symoblically execute the circuit
out = smt_alu(instr, i0, i1)
post_condition = to_pysmt(smt_alu.reg_1 == i1)

# pysmt code
with Solver("cvc4") as s:
  s.add_assertion(Not(post_condition))
  if s.solve():
    print("Counter example found")
  else:
    print("Verified")
```

Example 3.8: Verification of a PEak circuit using the CVC4 backend of pySMT.

4 Evaluation

PEak has been used in the design of three generations of CGRA-based programmable hardware accelerators: Garnet [4], Amber [18], and Onyx [28]. Amber and Onyx were fabricated in 16 nm and 12 nm commercial CMOS technologies respectively, and were verified in silicon.

CGRAs are a class of programmable accelerators composed of an array of tiles: processing element (PE) tiles, memory (MEM) tiles, and input/output (IO) tiles. PE tiles perform the arithmetic computation in the application, MEM tiles buffer data, and IO tiles send data to and from the array. These tiles communicate through a reconfigurable interconnect. PEak was used to specify the PE tiles for all three generations of CGRAs.

A CGRA PE operates at the word level and contains arithmetic operations found in a variety of applications. A typical PE contains an ALU with a variety of operations like add, multiply, shift, etc. It includes registers for integer operands, bit registers for bitwise operands, and a lookup table (LUT) for bitwise operations.

In each generation of CGRA, we extended the previous PEak PE to include more complex operations. For example, in the Garnet PE, the instruction set included only individual simple operations such as multiplication and addition. In the Amber PE, we wanted to include complex floating point operations like division, exponentiation, multiplication, natural log, and sine. The hardware for these operations was large and expensive, so we split each operation into smaller parts (e.g., get mantissa, subtract exponents, float to int, etc.). Then, we implemented these smaller operations within every PE, with the idea that when these expensive operations were required, we could use several PEs to implement one complex operation. This kept area overhead low while extending the capability of the CGRA. PEak made experimenting and implementing these complex operations easy, as the functional model written in Python could be used directly for the implementation.

In the Onyx chip, we extended the PEak PE instruction set to include larger operations such as multiply-add, min-max, and multiply-shift. These operations made accelerating applications in the image processing and machine learning domains much more efficient and performant. Implementing and experimenting with these operations in PEak was simple, and leveraging the formal model of each PE made verification easy and fast.

As an indication that PEak is easy to use and to learn, the PEs for Garnet, Amber, and Onyx were developed by 13 students, 8 of whom did not participate in the development of PEak. For students who were familiar with Python, the operation of PEak PEs were understood within minutes, and improvements could be made and designs could be tested within hours. The design productivity that PEak enabled was instrumental in the fast development of each of these accelerators.

PEak's unique capabilities have also enabled a number of research projects. Here, we present a summary of results from two of these projects. First, we discuss the CGRA specialization framework APEX [33], which uses PEak to generate high-performance RTL. Second, we describe our work on compiler rewrite rule synthesis [13], which uses PEak's formal model to synthesize instruction selection rewrite rules efficiently. Finally, we compare a simple ALU specified in PEak, PyRTL, and Chisel to highlight the advantages of PEak.

4.1 APEX

APEX aims to automatically specialize a CGRA's processing element (PE) architecture to an application or a class of applications. First, it uses frequent subgraph mining and analysis techniques to find common computational patterns in applications of interest. After finding frequent subgraphs, APEX merges these graphs into a new graph. This new graph acts as a specification of a specialized PE architecture capable of accelerating the applications.

APEX considers three axes while specializing PEs: number and type of operations within the PE, intraconnect within each PE, and number of inputs and outputs to and from the PE. Each has a direct effect on the complexity and capability of the PE and resulting CGRA.

After performing this analysis, APEX automatically converts the graph specification of each PE into a PEak program. At this point, APEX automatically inserts pipeline registers into the design to ensure high performance. The meta-programming utilities in PEak, including loop unrolling and if-statement inlining, make this conversion possible.

Figure 2 shows the results of evaluating APEX on four image-processing applications: camera pipeline, harris corner detection, unsharp, and gaussian blur. For each application, we compare an APEX-specialized PE (CGRA-IP) to results obtained using an FPGA, an unspecialized CGRA, and an ASIC. We compare both the energy consumed and the application runtime. The specialized CGRA-IP consumes 18% to 47% less energy than a generic CGRA with no specialization, while providing comparable performance.

The metaprogramming capability of PEak and the ability to easily generate parameterized designs that explored the design space were crucial enablers for this project. The APEX application analysis framework consumes application dataflow graphs and produces a dataflow graph representation of the PE specialized to those applications. Translating this dataflow graph, which can contain a variety of different operations, can have any number of inputs/outputs, and can include various means of interconnection between the subcomponents, into a hardware description is not straightforward.

For example, the parameter space of inputs and outputs in a PE is beyond the expressive capabilities of Verilog, which cannot parameterize the number of ports on a module. In Verilog, a new specification generator would need to be created for every PE that requires a unique number of inputs and outputs. In PEak, such parameterization is trivial, as the input ADT of each PE can be constructed with one line of Python code.

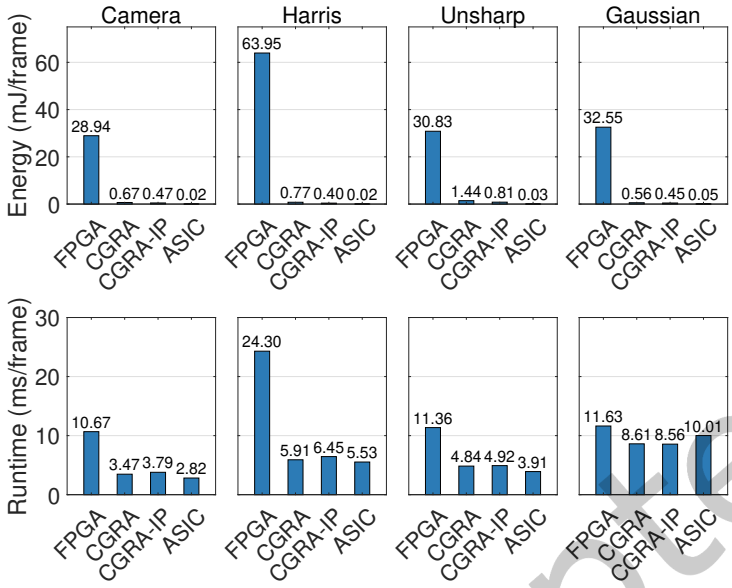

Fig. 2. Energy and runtime comparison between an FPGA, an unspecialized CGRA, an APEX-specialized CGRA, and an ASIC. Figure courtesy of Melchert et al. [33].

4.2 Rewrite Rule Synthesis

A working application compiler for each generated PE is required to perform realistic benchmarking of PEs during design space exploration. In this context, design space exploration means the systematic exploration and evaluation of many PE designs in order to optimize an objective such as power, performance, or area. During the instruction selection phase of code generation, *rewrite rules* are used to map computations described in an intermediate representation (IR) to concrete inputs, outputs, and instructions on the PE. Each distinct PE requires its own set of rewrite rules. Creating these rules manually is both labor-intensive and error-prone. Furthermore, manual construction would make automatic design space exploration impossible. In a recent work [13], we show how these rewrite rules can be efficiently and automatically synthesized, given a formal SMT model of the IR and the target PE. In that work, we conveniently use PEak to describe both, making it easy to extract the SMT models.

As an example, consider the rewrite rule for a 16-bit subtraction targeting the `ALU` described in Example 4.1. The rule specifies that the `invert_0`, `invert_1`, and `op` fields of `Inst` should be set to `InverterCtrl.ident`, `InverterCtrl.invert`, and `AluOp.ADD` respectively. Instead of manually creating this rule, it can be synthesized by solving the following SMT query: $\exists\, inst.\ \forall\, x, y.\ \text{bvsub}(16, x, y) = ALU(inst, x, y)$, where bvsub is the SMT operator for bitvector subtraction and ALU is the result of executing the PEak program with the SMT family interpretation (note that this is a simplified form of the query and does not take into account several complications discussed in [13] such as operand ordering, arity mismatches, and state). We show the construction of this query in Example 4.2.

Another challenge is handling instructions that use compile-time constants such as immediate fields (e.g., add immediate). Using the above formula, we would need a distinct query for each possible compile-time constant. Instead, we can modify the query by finding an instruction that works for every value of the constant, i.e., $\exists\, inst.\ \forall\, x, y, c.\ \text{bvadd}(16, x, c) = ALU(inst(c), x, y)$. To solve this query, we want to treat some fields of the

```
class AluOp(Enum):
  ADD = 0
  AND = 1
  OR  = 2

class InverterCtrl(Enum):
  ident  = 0
  invert = 1

class Inst(Product):
  invert_0 = InverterCtrl
  invert_1 = InverterCtrl
  op = AluOp

@family_closure
def gen(family):
  BV = family.BitVector
  T = BV[8]
  Bit = family.Bit
  @family.compile(locals(), globals())
  class ALU(Peak):
    def __call__(self,
        inst: Inst, in_0: T, in_1: T) -> T:
      if inst.invert_0 == InverterCtrl.invert:
        in_0 = ~in_0

      if inst.invert_1 == InverterCtrl.invert:
        in_1 = ~in_1
        cin = Bit(1)
      else:
        cin = Bit(0)

      if inst.op == AluOp.ADD:
        res, cout = add_with_carry(
          in_0, in_1, cin
        )
        return res
      elif inst.op == AluOp.AND:
        return in_0 & in_1
      else:
        return in_0 | in_1
  return ALU
```

Example 4.1: An ALU supporting 6 operations: Add, Subtract, And, Or, Nand, Nor.

```
i0 = SMTBitVector[8]()
i1 = SMTBitVector[8]()
instr = make_symbolic(Inst)

smt_alu = gen.SMT()
# symbolically execute the circuit
out = smt_alu(instr, i0, i1)

# construct the synthesis query (pysmt code)
spec = to_pysmt(out == i0 - i1)
universal_vars = [to_pysmt(i0), to_pysmt(i1)]
query = ForAll(universal_vars, spec)
with Solver("cvc4") as s:
  s.add_assertion(query)
  if s.solve():
    val = s.get_py_value(to_pysmt(instr))
    print("Rule found using instruction:")
    print(disassemble(val))
  else:
    print("No Rule")
```

Example 4.2: Rewrite rule synthesis query using PEak and pySMT.

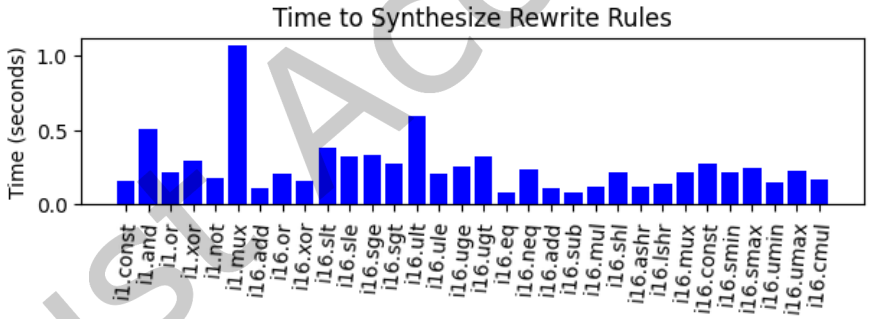

Fig. 3. Rewrite rule synthesis times for various IR instructions.

instruction as universally quantified and others as existentially quantified. PEak's ability to represent instructions as ADTs makes this possible.

Figure 3 shows the results of synthesizing rewrite rules for a set of IR instructions. The maximum time is 1.1 seconds. Synthesizing all of the rules takes less than 30 seconds, fast enough to be used in the loop during design space exploration, and a significant improvement over manual implementation of rules. This approach scales well to larger, more complex processors as well. We implemented a RISC-V processor with the RV32IM instruction set. It took 3 minutes to solve for all of the 37 rewrite rules for this architecture.

PEak's formal interpretation was also critical to the success of this project. If this project were implemented using another HDL, one without a formal interpretation, a separate formal representation of each design would

have to be created. Automatically generating the formal representation not only saves a significant amount of time and effort, but it also ensures that the formal representation matches the behavior of the hardware and functional model.

4.3 Comparison with PyRTL and Chisel

In this subsection, we show an example of a simple PE specified in three different languages: PEak, PyRTL [12], and Chisel [3]. The goal is to illustrate how hardware specified in PEak differs from these other languages, and how a hardware designer familiar with Python would find writing a PEak specification most natural.

PyRTL is a Python-embedded hardware design language intended to provide a more Pythonic method of specifying hardware. Rather than a high-level synthesis approach, in which a design is inferred from a high-level language, PyRTL instead provides a set of primitives in Python for constructing the hardware. Chisel is a popular Scala-based hardware description language which focuses on object-orientation, functional programming, and type safety.

In Example 4.3, we show the specification and functional verification code for a simple ALU written in PEak. This ALU takes as input an *instruction* specified as an ADT. The instruction encodes information about how many inputs the ALU is using and whether the ALU is performing an addition or multiplication.

Example 4.4 shows a PyRTL specification of the same ALU. While both the PyRTL and PEak languages are embedded in Python, the PEak code uses fewer non-native Python APIs, and its structure is much more similar to a native Python program. For example, inputs in PEak are specified as inputs to the `__call__` method of the ALU class, while the inputs in the PyRTL specification are declared using pyrtl.Input. Additionally, the PEak ALU can be instantiated and called like a normal Python class, while the PyRTL ALU must be simulated using PyRTL APIs.

Example 4.5 shows a Chisel specification of the same ALU. As Chisel is embedded in Scala instead of Python, the syntax used in this specification is very different. A typical hardware designer is more likely to know Python than Scala, and therefore would have an easier time writing and understanding PEak than Chisel. PEak is less verbose than Chisel and thus results in shorter, more concise code.

For this example, PEak has clear advantages over the other two languages. The PEak specification and verification is more concise, and the support for ADT types simplifies both the process of passing the instruction to submodules and the logic for decoding the instruction. Furthermore, the focus on maintaining a Python-like approach to constructing hardware makes PEak very natural for hardware designers familiar with Python to learn and understand.

5 Related Work

The design of PEak draws inspiration from the classic work of Bell and Newell [8], which similarly separated the logical description of an ISA from its semantics and bit-level representations. However, this idea seems to have been largely lost over time and, to our knowledge, is not used in any modern system. PEak generalizes this idea from ISAs to arbitrary ADTs.

There are many HDLs designed for general-purpose hardware construction, the most popular being Verilog. However, Verilog has extremely limited meta-programming capabilities, weak type systems, and poorly defined semantics. More modern languages with strong type systems like Magma [40] and Chisel [3] ease meta-programming by being embedded in Python and Scala, respectively. These languages define hardware as a graph of modules which is explicitly wired together. In contrast, PEak uses an implicit wiring model to avoid combinational loops. This is a deliberate design decision to keep designs readable and to ensure deterministic behavior.[8] PEak also provides access to a formal model, a feature not available in other HDLs.

[8]This means that certain design patterns that use combinational loops, flip-flops constructed from NAND gates for example, are not expressible in PEak.

```python
from peak import Peak, family_closure, Const, family
from hwtypes.adt import Product, Enum
import random

family = family.PyFamily()
T = family.BitVector[8]

class AluOp(Enum):
  ADD = 0
  MUL = 1

class NumInputsOp(Enum):
  TWO = 0
  THREE = 1

class Inst(Product):
  alu_op = AluOp
  num_inputs = NumInputsOp

@family.compile(locals(), globals())
class AddMul(Peak):
  def __call__(self, alu_op: AluOp, a: T, b: T) -> T:
    if alu_op == AluOp.ADD:
      return a + b
    else:
      return a * b

@family.compile(locals(), globals())
class ALU(Peak):
  def __init__(self):
    self.addmul = AddMul()

  def __call__(self, inst: Inst, a: T, b: T, c: T) -> T:
    if inst.num_inputs == NumInputsOp.TWO:
      c_temp = 0
    else:
      c_temp = c
    return self.addmul(inst.alu_op, a, b) + c_temp
```

```python
py_alu = ALU()
for alu_op in (AluOp.ADD, AluOp.MUL):
  for num_inputs in (NumInputsOp.TWO, NumInputsOp.THREE):
    inst = Inst(alu_op=alu_op, num_inputs=num_inputs)
    a = random.randint(0, 10)
    b = random.randint(0, 10)
    c = random.randint(0, 10)
    out = py_alu(inst, a, b, c)
    if num_inputs == NumInputsOp.TWO:
      assert out == (a+b) if alu_op == AluOp.ADD else out == (a*b)
    else:
      assert out == (a+b+c) if alu_op == AluOp.ADD else out == (a*b+c)
```

Example 4.3: Left: PEak specification of a simple ALU. Right: PEak functional verification code for the simple ALU.

PEak is also inspired by Lava [9], a Haskell-based DSL which supports multiple interpretations similar to PEak. Lava programs, like Magma and Chisel programs, describe hardware structurally. CλaSH [2] is another Haskell-based DSL which is less structural than Lava. It allows the use of case statements and pattern matching, enabling the construction of complex control structures which are difficult to build structurally. However, it does not have direct support for formal analysis like PEak and Lava. Both of these languages have limited type systems. In particular, they lack the ADT capability supported by PEak. Finally, while Haskell is appealing to DSL

```python
import random
import pyrtl
import enum

class AluOp(enum.IntEnum):
  ADD = 0
  MUL = 1

class NumInputsOp(enum.IntEnum):
  TWO = 0
  THREE = 1

def AddMul(a, b, op):
  alu_out = pyrtl.WireVector(bitwidth=8, name='alu_out')
  with pyrtl.conditional_assignment:
    with op == AluOp.ADD:
      alu_out |= a + b
    with op == AluOp.MUL:
      alu_out |= a * b
  return alu_out

def ALU(alu_op, num_inputs, a, b, c):
  c_temp = pyrtl.WireVector(bitwidth=8, name='c_temp')
  with pyrtl.conditional_assignment:
    with num_inputs == NumInputsOp.TWO:
      c_temp |= 0
    with num_inputs == NumInputsOp.THREE:
      c_temp |= c
  out = AddMul(a, b, alu_op) + c_temp
  return out

a = pyrtl.Input(8, 'a')
b = pyrtl.Input(8, 'b')
c = pyrtl.Input(8, 'c')
alu_op = pyrtl.Input(1, 'alu_op')
num_inputs = pyrtl.Input(1, 'num_inputs')
out = pyrtl.Output(8, 'out')

out <<= ALU(alu_op, num_inputs, a, b, c)
```

```python
sim_trace = pyrtl.SimulationTrace()
sim = pyrtl.Simulation(tracer=sim_trace)

cycle = 0
for alu_op in (AluOp.ADD, AluOp.MUL):
  for num_inputs in (NumInputsOp.TWO, NumInputsOp.THREE):
    a = random.randint(0, 10)
    b = random.randint(0, 10)
    c = random.randint(0, 10)
    sim.step({
      'a': a,
      'b': b,
      'c': c,
      'alu_op': alu_op,
      'num_inputs': num_inputs
    })
    out = sim_trace.trace['out'][cycle]

    if num_inputs == 0:
      assert out == (a+b) if alu_op == 0 else out == (a*b)
    else:
      assert out == (a+b+c) if alu_op == 0 else out == (a*b+c)

    cycle += 1
```

Example 4.4: Left: PyRTL specification of a simple ALU. Right: PyRTL functional verification of the simple ALU.

designers, as it enables elegant meta-programming through the use of type class polymorphism and higher order functions, practice has shown that getting working engineers to adopt a Haskell-based DSL is challenging.

For example, Bluespec SystemVerilog (BSV) [34], a term rewriting system (TRS) that describes circuits as a set of guarded atomic actions (rules), originally had a Haskell-like syntax. However, to appeal to a wider audience, it has since adopted an imperative syntax that is closer to behavioral Verilog. BSV rules describe a circuit's behavior as state updates and outputs predicated on current states and inputs. Abstractly, these rules are atomic

```
import chisel3._

class AddMul extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val alu_op = Input(UInt(1.W))
    val out = Output(UInt(8.W))
  })
  when (io.alu_op === 0.U) {
    io.out := io.a + io.b
  } .otherwise {
    io.out := io.a * io.b
  }
}

class ALU extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val c = Input(UInt(8.W))
    val alu_op = Input(UInt(1.W))
    val num_inputs = Input(UInt(1.W))
    val out = Output(UInt(8.W))
  })
  val alu = Module(new AddMul)
  alu.io.alu_op := io.alu_op
  alu.io.a := io.a
  alu.io.b := io.b
  val c_temp = Wire(UInt(8.W))
  when (io.num_inputs === 0.U) {
    c_temp := 0.U
  } .otherwise {
    c_temp := io.c
  }
  io.out := alu.io.out + c_temp
}
```

```
import chisel3.iotesters.{PeekPokeTester, Driver, ChiselFlatSpec}

class ALUTests(alu: ALU) extends PeekPokeTester(alu) {
  for (alu_op <- 0 until 2) {
    for (num_inputs <- 0 until 2) {
      val a = rnd.nextInt(10)
      val b = rnd.nextInt(10)
      val c = rnd.nextInt(10)

      var output = 0
      if (alu_op == 0) {
        output = (a+b)
      } else {
        output = (a*b)
      }
      if (num_inputs == 1) {
        output += c
      }
      poke(alu.io.a, a)
      poke(alu.io.b, b)
      poke(alu.io.c, c)
      poke(alu.io.alu_op, alu_op)
      poke(alu.io.num_inputs, num_inputs)
      step(1)
      expect(alu.io.out, output)
    }
  }
}

class ALUTester extends ChiselFlatSpec {
  behavior of "ALU"
  backends foreach {backend =>
    it should s"perform correct math operation on dynamic operand in $backend" in {
      Driver(() => new ALU, backend)((alu) => new ALUTests(alu)) should be (true)
    }
  }
}
```

Example 4.5: Left: Chisel specification of a simple ALU. Right: Chisel functional verification of the simple ALU.

and are applied sequentially, one rule at a time. However, in practice this would lead to extremely inefficient hardware. Therefore, the BSV compiler attempts to schedule these rules concurrently when possible. When multiple rules can update the same state element they must be scheduled sequentially. The choice of schedule can have significant impact on the quality of the resulting hardware. Kôika [10] is a BSV derivative which aims to eliminate this by giving engineers direct control over the schedule.

A related line of work is high-level synthesis [14] (HLS) which allows designers to describe the behavior of circuits using a high-level programming language such as C, C++, SystemC, or Matlab. HLS programs describe the algorithmic behavior of a circuit, eschewing low-level details like pipelining and resource allocation. An

HLS compiler then determines some minimal set of resources which are capable of performing the described algorithm and an associated schedule of computation, i.e. where and when each operation in the source program takes place. While HLS is a popular design paradigm and can provide significant engineering efficiency gains, it often produces low-performance RTL [1].

Contemporary work on ISA specification falls into two main categories: ad hoc specification of existing ISAs [21, 36] and frameworks which are more analogous to PEak for specifying ISAs such as SAIL [23], ILA [25], and ISA-Formal [37]. These systems use declarative descriptions of the semantics of instructions as state updates predicated on the bit-level representation of an instruction. These are powerful tools, but they cannot be used to generate RTL. While this disconnect makes sense when verifying new RTL against an existing ISA specification, it is tedious when the ISA itself being developed, as for each new candidate ISA, both its RTL and its specification must be written separately. In contrast, PEak uses a procedural model in which bit-level encodings are decoupled from the behavioral specification. Further, PEak can be used both for specification and RTL-generation.

6 Conclusion

PEak is built on top of `hwtypes` and `ast_tools`. `hwtypes` provides a Pythonic interface to functional simulation, formal SMT models, and RTL generation via Magma. `ast_tools` provides infrastructure for Python AST analysis and transformations and enables the reinterpretation of Python control flow. PEak provides designers with the means to specify a single source of truth for hardware design, which has proven to be a useful paradigm for enabling novel automated design methodologies which incorporate formal methods. The design decisions made when creating PEak, including the focus on an object-oriented view of the hardware, raising of the level of abstraction through an implicit clocking and wiring model, multiple interpretations including a functional, hardware, and formal model, strong support for ADT types, and access to the AST, have all been instrumental in making PEak an excellent language for hardware design. PEak is easy to learn and has features that simplify both design productivity and verification. PEak has enabled us to develop three generations of CGRA architectures, a PE specialization framework, and a rewrite rule synthesis technique. We hope that PEak, along with `hwtypes` and `ast_tools`, will also encourage future work in this domain.

7 Acknowledgments

This work was supported by funding from SRC JUMP 2.0 PRISM Center, NSF CAREER (award number: 2238006), DARPA DSSoC, Stanford Agile Hardware (AHA) Center, Stanford SystemX Alliance and Apple Stanford EE PhD Fellowship in Integrated Systems.

References

[1] Abhinav Agarwal, Man Cheuk Ng, et al. 2010. A comparative evaluation of high-level hardware synthesis using reed–solomon decoder. *IEEE Embedded Systems Letters* 2, 3 (2010), 72–76.

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. ClaSH: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 714–721.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC) 2012*. IEEE, 1212–1221.

[4] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. 2020. Creating an Agile Hardware Design Flow. In *Design Automation Conference (DAC)*.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[6] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability, Second Edition*, Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, Chapter 33, 825–885. http://www.cs.stanford.edu/~barrett/pubs/BSST21.pdf

[7] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[8] C. Gordon Bell and Allen Newell. 1970. The PMS and ISP Descriptive Systems for Computer Structures *(AFIPS'70 (Spring))*. Association for Computing Machinery, New York, NY, USA, 351–374. https://doi.org/10.1145/1476936.1476993

[9] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. *ACM SIGPLAN Notices* 34, 1 (1998), 174–184.

[10] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–257.

[11] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. In *2022 IEEE Symposium on VLSI Technology and Circuits*.

[12] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.23919/FPL.2017.8056860

[13] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. 2022. Synthesizing Instruction Selection Rewrite Rules from RTL using SMT. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 139–150.

[14] Luka Daoud, Dawid Zydek, and Henry Selvaraj. 2014. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. In *Advances in Systems Science: Proceedings of the International Conference on Systems Science 2013 (ICSS 2013)*. Springer, 483–492.

[15] Andrew Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. 2023. Verification of Chisel Hardware designs with ChiselVerify. *Microprocessors and Microsystems* 96 (2023), 104737.

[16] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 408–422.

[17] Kathleen Feng, Alex Carsello, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: Coarse-Grained Reconfigurable Array-Based SoC for Dense Linear Algebra Acceleration. In *2022 IEEE Hot Chips 34 Symposium (HCS)*.

[18] Kathleen Feng, Taeyoung Kong, Kalhan Koul, Jackson Melchert, Alex Carsello, Qiaoyi Liu, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zach A. Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2024. Amber: A 16-nm System-on-Chip With a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. *IEEE Journal of Solid-State Circuits* 59, 3 (2024), 947–959. https://doi.org/10.1109/JSSC.2023.3313116

[19] Python Software Foundation. 2023. The Python Language Reference. https://docs.python.org/3/reference/datamodel.html#basic-customization.

[20] Marco Gario and Andrea Micheli. 2015. PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*.

[21] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and formal verification of x86 machine-code programs that make system calls. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. 91–98.

[22] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51.

[23] Kathryn E Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture*. 635–646.

[24] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[25] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (Dec 2018), 24 pages. https://doi.org/10.1145/3282444

[26] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[27] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. 2023. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Trans. Embed. Comput. Syst.* (2023).

[28] Kalhan Koul, Maxwell Strange, Jackson Merlchert, Alex Carsello, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengelek, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Zhouhua Xie, Christopher Torng, Joel Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. 2024. Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications. In *IEEE Symposium on VLSI Technology & Circuits (VLSI)*. IEEE.

[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[30] Derek Lockhart, Stephen Twigg, Doug Hogberg, George Huang, Ravi Narayanaswami, Jeremy Coriell, Uday Dasari, Richard Ho, Doug Hogberg, George Huang, Anand Kane, Chintan Kaur, Tao Kaur, Adriana Maggiore, Kevin Townsend, and Emre Tuncer. 2018. Experiences Building Edge TPU with Chisel. In *2018 Chisel Community Conference (CCC)*.

[31] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 280–292.

[32] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*. Springer, 61–70.

[33] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A Horowitz, Pat Hanrahan, and Priyanka Raina. 2023. APEX: A Framework for Automated Processing Element Design Space Exploration using Frequent Subgraph Analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 33–45.

[34] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 69–70.

[35] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 389–402.

[36] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 161–168.

[37] 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 *International Conference on Computer Aided Verification*. Springer, 42–58.

[38] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL'88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[39] Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, et al. 2010. Rethinking digital design: Why design must change. *IEEE Micro* 30, 6 (2010), 9–24.

[40] Lenny Truong and Pat Hanrahan. 2019. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[41] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark Barrett, et al. 2020. fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components. In *International Conference on Computer Aided Verification*. Springer, 403–414.

[42] Keyi Zhang, Zain Asgar, and Mark Horowitz. 2022. Bringing Source-Level Debugging Frameworks to Hardware Generators. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1171–1176. https://doi.org/10.1145/3489517.3530603

Received 1 February 2024; revised 11 August 2024; accepted 17 October 2024