PyCaliper: Python-Embedded Infrastructure for RTL Verification and Specification Synthesis

Adwait Godbole[1], Brian Huffman[2], Fangfei Liu[2], Carlos V Rozas[2], and Sanjit A. Seshia[1]

[1] University of California, Berkeley, CA, USA
[2] Intel Labs, USA

Abstract. We present PyCaliper: a Python-embedded framework to formulate, verify, and auto-synthesize specifications for hardware designs at the register transfer level (RTL). By being Python-embedded, PyCaliper is easy to use and benefits from object-oriented principles and Python's rich ecosystem. Further, PyCaliper is a common platform that integrates novel research techniques such as specification synthesis and mature, industry-scale tooling, thus allowing them to benefit from each other. We discuss the system and implementation of PyCaliper and demonstrate its use in two case studies: in the first we compare a custom verification backend with a commercial tool and gain insights about the former, and in the second we demonstrate invariant synthesis for an RTL design.

Keywords: formal specification · hardware verification · invariant synthesis.

1 Introduction

Formal verification of hardware designs provides strong guarantees about correctness and is, thus, a critical step in the chip design process. Verification accounts for as much as 20% of total development costs [40], which can be as high as $500M for a modern chip. Industry relies on proprietary verification tools such as Cadence/Jasper Proof Apps [13] and Synopsys Formality [47]. These are mature tools based on decades of research and engineering and able to handle industry-scale designs. However, as observed more than a decade ago [42,43], to fully leverage such tools one additionally needs to *synthesize* high-quality specifications and other proof artifacts. Over the past decade, the open-source community has seen development of RTL language services, verification infrastructure, novel algorithmic improvements, solver backends (e.g., [53,30,12,29,23,38]), and techniques to synthesize proof artifacts such as invariants/abstractions (e.g., [56,21]) that use underlying verification routines. Even so, these have not been widely adopted into industrial flows; industry and open-source tools remain largely disconnected. Importantly, there is a lack of infrastructure to connect tools and techniques that, while complementary in theory, are not integrated in practice. Such infrastructure for cooperating formal methods has several benefits including (a) easier integration of novel research (e.g., specification synthesis and learning) with mature industrial tools, (b) making open-source tools inter-operable with proprietary tools, and (c) enabling comparisons between open-source and proprietary tools.

Desiderata. This suggests developing verification infrastructure that is:

- *Compatible, yet independent.* Given their mainstream acceptance, it should be compatible with industry-standard tools. Simultaneously, it should not be restricted to these tools and should also operate with open-source tooling.
- *Accessible and easy to (re)use.* It should be accessible in a friendly language. Further, it should promote the reuse of artifacts (e.g., specifications, models, and proofs) built in it across proof tasks.
- *Close to the design.* It should operate close to the design implementation making it easier to formulate and debug rich specifications.

This paper presents PyCaliper, our attempt to build such infrastructure: PyCaliper is Python-embedded framework that provides a DSL to formulate RTL specifications and verification backends to verify these specifications, and synthesis engines to auto-generate missing parts of specifications.

PyCaliper System Overview. Figure 1 illustrates the PyCaliper architecture. PyCaliper specifications (Section 2), are formulated by extending the `SpecModule` abstract class. By being Python-embedded, PyCaliper leverages Python's rich ecosystem, and promotes reuse by exporting specifications as library modules. Further it allows flexible meta-programming that can be used to build domain-specific invariants targeted to certain kinds of properties. For example, besides generic specifications, we currently support invariants for two-trace security (hyper-)properties [17] (Table 2). Specifications can be complete or "incomplete", i.e., have missing parts, called *holes*.

PyCaliper is compatible yet independent: it has SVA and BTOR2 verification backends (Section 3) that currently interface with both, the commercial Jasper FPV App [13] and a custom BTOR2-based [34] symbolic execution. By closely mirroring the design hierarchy, PyCaliper allows deep-specification/verification that is close to the RTL implementation. A key component of PyCaliper are *invariant synthesis engines* (Section 4) that leverage automated techniques (e.g., [36,21,45]) to fill holes in incomplete specifications. In this way, PyCaliper supports *hardware specification engineering*: developing specification formalisms that are amenable to verification and synthesis, and techniques that can operate on these formalisms. PyCaliper is open-source on GitHub - https://github.com/pycaliper/pycaliper - with a web-page hosted at https://pycaliper.github.io.

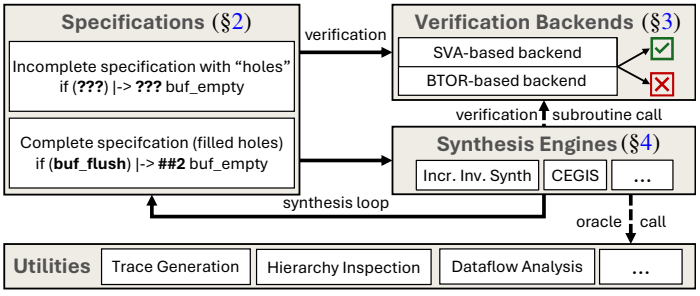

Fig. 1. PyCaliper System Architecture and Components.

Related work. Formal hardware verification tools such as EBMC [19], the Yosys [53] based SymbiYosys [15], mcy [54], and Pono [29] perform verification by compiling the RTL design and SVA properties to automated (SAT/SMT-based [6]) solvers. Tools based on IC3 and PDR [10,9] (e.g., AVR [23] and Avy [52]) perform unbounded verification of RTL by refinement-based invariant search. PyCaliper on the other hand, is not a verification tool in itself: our main contribution is Python-embedded infrastructure that enables programmatic RTL specification and interfaces with (SVA and BTOR2)-based verification engines. We use these verifiers as *oracles* to implement invariant synthesis techniques such as syntax-guided synthesis via CEGIS [3] and incremental invariant synthesis [36,11] (the latter itself uses an approach based on PDR).

PyCaliper draws inspiration from the ease-of-use and flexibility provided by Python-embedded hardware tools: `cocotb` [18] (for test-based validation) and `MyHDL` [31] (for hardware design) and applies these ideas to hardware verification. Its use of *oracle-guided synthesis [26] and learning* follows UCLID5 [44,38], which pioneered the use of syntax-guided, oracle-guided, and learning-enabled synthesis in formal verification.

Tooling for hardware verification that leverages programming languages principles has recently seen a lot of interest. The MLIR-based CIRCT [28] infrastructure introduces *verification dialects*, which aim to provide a hardware model-checking tool-friendly intermediate representation (IR). Tools such as Kami [14] also perform hardware verification by formulating a specification language close to the implementation, but they use interactive theorem provers (e.g., Rocq, formerly Coq [49,7]) for verification. Other examples in the PL-for-HW space include design languages such as Clash [5] and Dahlia [35] that are based on HW-specific static types to ensure correctness, e.g., with respect to timing and resource usage. These are design languages, while PyCaliper is a specification DSL. However, future work can develop new specification constructs in PyCaliper inspired by the type semantics of these languages.

2 Python-Embedded RTL Specifications

We now describe key PyCaliper constructs and their semantics.

2.1 PyCaliper Specifications

Specification elements. PyCaliper specifications consist of elements detailed in Table 1. These naturally map to RTL constructs. For example, the `Logic` element is parameterized by its (integer) width and corresponds a reg/wire/logic signal in the Verilog/SystemVerilog design. The abstract `SpecModule` class represents the specification for an RTL module and is inherited by each PyCaliper specification class.

Figure 2 illustrates such a specification for a `fifo` queue module, with the `__init__` function initializes the module with a set of signals. Note that since these specifications are ordinary Python classes, they can be easily augmented with usual Python code. For example, in the `fifo` module the sizes of the read and write pointers are defined as `Logic` elements with size dependent on the depth of the FIFO queue. The remainder of the class uses these elements to define the specification.

SV RTL module

```
module fifo #(
    parameter DEP=8
) (
  input  logic clk,
  input  logic reset,
  input  logic push,
  input  logic pop,
  input  TYPE  d_in,
  output TYPE  d_out,
  output logic empty
);
  logic [$log2(DEP):0] rd_ptr;
  logic [$log2(DEP):0] wr_ptr;

  // FIFO memory
  TYPE [DEP-1:0] fifo;
  ...

  // FIFO module logic ...
endmodule:fifo
```

```
class fifo(Module):
  def __init__(self, cf: config_t):
    super().__init__()
    self.DEPTH = cf.depth
    self.DIDX = int(
          math.log2(self.DEPTH)+1)
    self.WIDTH = cf.width

    # Input signals
    self.reset = Logic()
    self.push = Logic()
    self.pop = Logic()
    ...

    # Internal signals
    self.rd_ptr = Logic(self.DIDX)
    self.wr_ptr = Logic(self.DIDX)
    self.fifo_ents = LogicArray(lambda:
        Logic(self.WIDTH), self.DEPTH)
    ...
```

PyCaliper spec. class

```
def input(self):
  self.eq(self.reset)
  self.eq(self.push)
  self.eq(self.pop)
  self.when(self.push)(self.d_in)
  self.inv(~self.empty | ~self.pop)

def state(self):
  self.eq(self.rd_ptr)
  self.eq(self.wr_ptr)

  for i in range(self.DEPTH):
    self.when(
      self.wr_ptr-self.rd_ptr >
      Const(i,self.DIDX)-self.rd_ptr
    )(self.fifo_ents[i])

def output(self):
  self.eq(self.avail_entries)
  self.when(~self.empty)(self.d_out)
```

Fig. 2. Left: A `fifo` SystemVerilog RTL module for a FIFO queue. Right: A PyCaliper `SpecModule` specification corresponding to a no-stale-data property (critical for verifying data-at-rest optimizations against attacks, e.g., [50]) for the `fifo` module.

Element	Mapping to RTL Construct
`SpecModule`	An RTL `module` in the design hierarchy.
`Struct`	A SystemVerilog `struct`.
`Group`	Bundle of signals under a hierarchical name (e.g., `generate` block).
`Logic`	Represents an RTL `reg`/`wire` in the design.
`LogicArray`	Represents an (n-D) array of logic signals.

Table 1. Elements in PyCaliper specifications and RTL constructs they map to.

Inductive specification. Inductive specifications in PyCaliper consist of three components: `input`, `state`, and `output` blocks. These correspond to global assumptions, the state invariants, and output assertions, respectively. In general, the `input`, `state`, and `output` functions can be thought of as predicates over the state of the RTL design. With this view, inductive proofs aim to verify the following property:

$$T_{\text{RTL}} \models (\texttt{input}, \texttt{state}, \texttt{output}) \triangleq$$
$$\forall S, S'.\ \texttt{input}(S) \wedge \texttt{state}(S) \wedge T_{\text{RTL}}(S, S') \Rightarrow (\texttt{state}(S') \wedge \texttt{output}(S'))$$

Invariants. PyCaliper supports both single trace invariants defined over one design instance as well as two-trace invariants defined over a miter (self-composition) of the design. The latter are used for proofs of security (hyper)-properties [17] such as no-stale-data leakage/observational-determinism [55]. Table 2 (top) illustrates these invariants and their semantics.

Synthesis holes. A key feature of PyCaliper is the ability to synthesize invariants for incomplete specifications. PyCaliper supports *holes* that identify syntactic templates for the invariant, much like sketching-based synthesis [46,45] or syntax-guided-synthesis (SyGuS) [3]. PyCaliper currently supports two holes shown in Table 2 (lower half). The first (`self.ceqhole(e, slist)`) aims to identify a subset of signals from `slist` that are equal in the two copies of the miter conditional on expression `e` being true. The second (`self.ctrlhole(ctrl, dep)`) aims to generate a lookup-table (LUT) of form `if (ctrl == X1) Y1 elif (ctrl == X2) Y2 ...` provides the value

Invariant or Hole	Semantics or Synthesis Spec. (Holes)
`self.eq(x: Logic)`	`A.x` $=$ `B.x`
`self.when(expr: Expr)(x: Logic)`	`expr(A)` $\wedge$ `expr(B)` $\implies$ `A.x` $=$ `B.x`
`self.inv(expr: Expr)`	`expr(A)`
`self.ceqhole(e: Expr, slist: list[Logic])`	$?S \subseteq$ `slist`. $\wedge_{s \in S}$ `self.when(e)(s)`
`self.ctrlhole(ctrl: Logic, dep: Logic)`	?LUT. `self.inv(dep == LUT(ctrl))`

Table 2. PyCaliper invariants (above), synthesis holes (below) and their corresponding semantics. We denote the two module instances in the miter as `A` and `B`.

of the dependent signal (`dep`) as a function of the control signal (`ctrl`). PyCaliper implements *synthesis engines* that synthesize a valid solution satisfying the specification. We discuss these in Section 4.

Bounded specifications. In addition to invariant-based unbounded proofs, PyCaliper also supports bounded verification. A bounded specification can be formulated by defining a function (e.g., `f`) such that `f(i)` adds the assumptions and assertions for the `i`-th cycle. Assumptions and assertions are added by calling `pycassume` and `pycassert` respectively with the required property.

```
vals = [randint(0, MAX), randint(0, MAX)]
ins = [Const(val, 32) for val in vals]
expected = Const(sum(vals) % MAX, 32)

class Adder(Module):

  @unroll(2)
  def simstep(self, step: int = 0):
    self.pycassume(self.rst_ni) # active low
    if step == 0:
      self.pycassume(self.a_i == ins[0])
      self.pycassume(self.b_i == ins[1])
    elif step == 1:
      self.pycassert(self.sum_o == expected)
```

Fig. 3. Bounded specification using the `unroll(k)` construct.

Example 1. Figure 3 provides an example for an `Adder` module in which the `simstep` function specifies that the output with two (randomly chosen) inputs `a` and `b` equals `a + b` modulo wrap-around.

Further PyCaliper provides the `unroll(k)` function decorator that internally unrolls `f` for `k` cycles and verifies the property.

2.2 Keeping Specifications close to the Implementation

Organizing specification using `SpecModules`. A full design specification often requires multiple properties (e.g., SVA `property` statements) that complement each other. For instance describing the stage-wise behavior of even a simple pipelined processor requires a collection of properties, one for every pipeline stage. The standard approach to this has been to collect all SVA properties alongside the design module. However, this leads to a large monolithic specification that is hard to maintain and understand. By organizing multiple properties into a `SpecModule`, PyCaliper leverages principles of aggregation/modularity which have been widely adopted in SE/OOP communities [8]. We discuss benefits of this organization in this section and Section 2.3.

Mirroring the design hierarchy. Since a `SpecModule` class corresponds to its RTL counterpart and can contain other `SpecModule` members (corresponding to RTL submodules), PyCaliper specifications mirror the RTL design hierarchy. Further specification elements (e.g., `Logic`) within a `SpecModule` also map to design signals.

The benefits of verification frameworks that operate "close to" the implementation have been demonstrated in previous work [4]. This allows specifications that are easier to formulate (due to low overhead of context-switching between implementation and specification) and more expressive (as the specification can be defined over implementation signals as opposed to some abstraction thereof). PyCaliper extends these advantages to RTL verification.

Elaboration. PyCaliper specifications are elaborated before being passed to the verification or synthesis backends. Elaboration uses Python's introspection features (dynamically examining objects in a scope) to traverse the specification hierarchy. A key aspect of elaboration is resolving signal names to their full hierarchical paths in the design. For example, elaboration of a `UART` specification with two (transmit/receive) `fifo` instances, `tx_fifo` and `rx_fifo` respectively will consist of a `tx_fifo.wr_ptr` `Logic` element corresponding to the write pointer of the transmit `fifo`.

2.3 Extensibility and Reuse of Specifications

Software, in particular open-source software, has long benefited from reuse. A key driver of this has been *libraries* that package reusable components. PyCaliper similarly enables reuse of RTL specifications through parameterization and refinement.

Parameterization. PyCaliper `SpecModules` can be instantiated with varying arguments. This is useful when formulating parameterized specifications for components such as FIFOs (seen in buffers), and tagged lookup tables (as seen in caches, predictors). For example, in Figure 2, the `fifo` module is parameterized by the `DEPTH` and element `WIDTH`, passed through the `config` argument. Parameterized specifications of this form can be packaged into Python libraries, much like Verification IPs [41] except being open-source and extensible.

Refinement. PyCaliper currently supports refinement-based proofs between two specifications under a particular refinement relation. Suppose there are two modules `M1` and `M2` with inductive specifications ($\texttt{input}_1$, $\texttt{state}_1$, $\texttt{output}_1$) and ($\texttt{input}_2$, $\texttt{state}_2$, $\texttt{output}_2$) respectively. Further suppose that $R(S_1, S_2)$ is a (binary) refinement relation between `M1` and `M2`. Here S_1 and S_2 are states of `M1` and `M2` respectively. A refinement proof establishes validity of the properties:

$$\Phi_I = \texttt{input}_1(S_1) \wedge \texttt{state}_1(S_1) \wedge R(S_1, S_2) \Rightarrow \texttt{input}_2(S_2) \wedge \texttt{state}_2(S_2)$$

... inputs hold in refinement

$$\Phi_O = \texttt{state}_2(S_2) \wedge \texttt{output}_2(S_2) \wedge R(S_1, S_2) \Rightarrow \texttt{state}_1(S_1) \wedge \texttt{output}_1(S_1)$$

... outputs hold in refinement

Refinement proofs are crucially used in compositional assume-guarantee reasoning [51,37]: they imply that any design that implements `M2` also implements specification `M1` under the refinement relation R. Symmetrically any proof that holds under the environment of `M1` also holds under the environment of `M2`.

Example 2. Consider the `fifo` buffer from Figure 2. This `fifo` refines a `Counter` module with a `ctr` signal and increment/decrement (`incdec`) operation (based on example in [14]), with the refinement relation R:

$$R \equiv (\texttt{ctr} = \texttt{wr\_ptr} - \texttt{rd\_ptr}) \wedge (\texttt{wr\_ptr} = \texttt{incdec}) \wedge (\texttt{rd\_ptr} = \texttt{\~{}incdec})$$

PyCaliper currently supports refinement proofs where both `M1` and `M2` modules transition synchronously in lock-step. In general, refinement relations may allow one module to *stutter*, i.e., not transition while the other module transitions. Such stuttering refinement [16,32] can be used to relate two modules that have similar functional behaviors but different timing behaviors. For example, Burch-Dill refinement [33] allows verification of a pipelined processor design against its non-pipelined counterpart.

3 Verification Backends

PyCaliper specifications are elaborated and dispatched to verification backends. We currently support two backends using SVA (SystemVerilog Assertions) [1], and BTOR2 [34] formats. We illustrate these in Figure 4 and now discuss them.

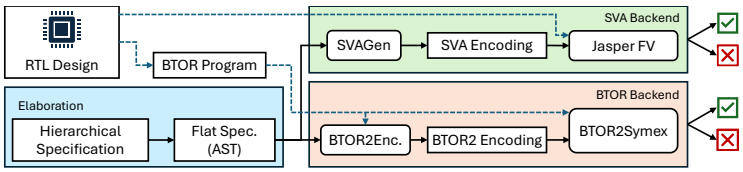

Fig. 4. PyCaliper Verification Infrastructure: SVA and BTOR2 backends.

3.1 SVA (System Verilog Assertions) Backend

The SVA backend consumes the elaborated PyCaliper specification and generates a SystemVerilog Assertion (SVA) file which is loaded into the proof engine. It then communicates with the proof engine through an oracle interface while referencing properties from the SVA file.

SVA generation. The generated SVA file begins with proof harness, which includes clocking/reset assumptions and a counter FSM (used to track steps in a k-inductive proof). Further it compiles the `input`, `state`, and `output` PyCaliper expressions into SVA properties. Finally, it uses the `ctr` signal from the FSM to define assumptions and assertions. For example, for a (k-)inductive proof:

```
P_in: assume ctr < k |-> input(...)     P_pre: assume ctr < k |-> state(...)
P_asrt: property ctr == k |-> state(...) && output(...)
```

Oracle-interface. After loading the above SVA file, PyCaliper communicates with the proof engine through a query-response interface shown in Table 3. Queries include enabling/disabling assumptions, property checks and trace generation.[3] Thus, the proof engine acts as an oracle for verification/synthesis, similar to the techniques in [25,39].

Actions
`enable_assm`
`disable_assm`
`cover`
`check`
`get_trace`

Table 3. Oracle Action Interface

PyCaliper currently supports the (licensed) Jasper Formal Property Verification (Jasper FPV) App [13] as the proof engine. However, we can enable support for other engines (e.g., open-source tools such as SymbiYosys [15]) by implementing the oracle interface. PyCaliper interacts with Jasper FPV via a TCP interface which is used to send Tcl commands (e.g., `prove P_asrt` for a `check` queries) and receive responses.

3.2 BTOR2 Backend

Yosys-generated BTOR2 Program. PyCaliper uses Yosys [53] to convert the (Verilog/SystemVerilog) RTL design into the BTOR2 [34] format. This involves standard synthesis steps such as flattening memories, hierarchy removal, and technology mapping. Importantly, the BTOR2 program preserves signal hierarchy information from the original RTL design which is used to map the PyCaliper specification signals with their design signal counterparts.

Symbolic Execution. The BTOR2 backend consumes the BTOR2 program and performs symbolic execution. The symbolic execution engine unrolls the BTOR2 program for a certain number of steps (dependent on the k-induction parameter `unroll`ing bound). Further it compiles the (inductive or `unroll`) specification into BTOR2 expressions and evaluates them on the unrolled symbolic state. The resulting formula is verified using SMT-based verification techniques. While we currently use Boolector as the SMT backend, this can be easily extended to other QFBV solvers [6] by adding appropriate operator mappings.

4 Synthesis Engines

4.1 Background: Formal Synthesis

The problem of formal synthesis of functions can be formulated as follows. Given a target function f to be synthesized, a specification ϕ that f should satisfy, and a concept class $\mathcal{F}$ describing possible function implementations, the synthesis problem aims find $f \in \mathcal{F}$ such that f satisfies ϕ. Different formal synthesis problems formulate ϕ and $\mathcal{F}$ in different ways (e.g., as a set of input-output examples or logical constraints, or a formal language). For example, Syntax-guided synthesis (SyGuS) [3] is a grammar-based approach that uses a syntactic grammar G to induce the space of possible implementations $\mathcal{F}$. In some cases (e.g., Houdini [20,36]) the space of implementations allows for specialized algorithms to search for f.

[3] We require dynamically disabling assumptions to perform invariant synthesis (Section 4).

In the case of PyCaliper's specification synthesis feature, the synthesis target f corresponds to the `state` invariant function in `SpecModule`. PyCaliper allows the `state` function to contain holes that act as a templates for missing parts of specifications. PyCaliper implements synthesis engines that are tailored to the structure of these holes and are based on invariant/abstraction synthesis techniques from literature (e.g., [20,45,25]). We now discuss these in more detail.

4.2 Incremental Invariant Synthesis

Recall that `self.ceqhole(e, siglist)` asks to identify signals `s` $\in$ `siglist` such that `self.when(e)(s)` is a valid invariant. PyCaliper implements a fuel-based variant of the incremental invariant synthesis algorithm [36,11] for this hole template. It treats the invariants $A = \{\texttt{self.when(e)(s)}\}_{\texttt{s}}$ as candidate atoms in a grammar, and searches for a conjunction of a subset of these atoms to form the overall invariant.

The algorithm maintains two subsets of A: `assms` (speculatively guessed assumptions) and `assrts` (assertions). They are always relative inductive, denoted as `assms` $\rightharpoonup$ `assrts`. This means `assms` and the background properties in `input` and (the non-hole part of) `state` imply `assrts` at the next step of execution:

$$\texttt{assms} \rightharpoonup \texttt{assrts} \triangleq$$
$$\forall S, S'.\texttt{input}(S) \wedge \texttt{state}(S) \wedge \texttt{assms}(S) \wedge T_{RTL}(S, S') \implies \texttt{assrts}(S')$$

The algorithm admits a fuel parameter f^* that imposes an upper bound on $|\texttt{assms}| - |\texttt{assrts}|$ during the search. Intuitively, a higher f^* allows for more speculative guessing (with higher coverage), while a lower f^* forces the algorithm to be more conservative.

$$\text{dive}\ \frac{a = \texttt{strategy}(\texttt{assms}, \texttt{assrts}) \in A \quad |\texttt{assrts}| - |\texttt{assms}| > f^*}{(\texttt{assrts}, \texttt{assms}) \rightarrow (\texttt{assrts}, a :: \texttt{assms})}$$

$$\text{backtrack}\ \frac{\texttt{assms} = a :: \texttt{assms}' \quad \texttt{assms}' \rightharpoonup \texttt{assrts}'}{(\texttt{assrts}, \texttt{assms}) \rightarrow (\texttt{assrts}', \texttt{assms}')} \qquad \text{saturate}\ \frac{a \in A \setminus \texttt{assrts} \quad \texttt{assms} \rightharpoonup a}{(\texttt{assrts}, \texttt{assms}) \rightarrow (a :: \texttt{assrts}, \texttt{assms})}$$

$$\text{done}\ \frac{\texttt{assms} \subseteq \texttt{assrts} \quad T_{RTL} \models (\texttt{input}, \texttt{state} \cup \texttt{assrts}, \texttt{output})}{(\texttt{assrts}, \texttt{assms}) \rightarrow \text{success}(\texttt{assrts})}$$

Fig. 5. Declarative rules for incremental invariant synthesis.

We present the algorithm as a declarative rule-set in Figure 5. The algorithm is parameterized by a strategy function that decides what candidate from A to add to the current `assms` during a *dive* step. Addition is only possible if there is fuel remaining. The *saturate* steps adds all assertions that are relative inductive to the current `assms`. If the strategy explored bad candidate assumptions, the algorithm may need to *backtrack* (i.e., remove an assumption) and explore other candidates. Once it finds a self inductive `assrts` that can provide the overall property (`output`), the concludes with *success*. If *dive* cannot pick a new (unexplored) assumption, or runs out of fuel, synthesis fails.

This algorithm is parameterized by the strategy function (`strategy`) which has contextual knowledge of `assms` and `assrts` as well as possibly other information such as the design hierarchy, signal dependencies, etc. PyCaliper allows the user to provide a custom strategy (or use one of in-built strategies). We explore two such strategies in our experimental evaluation (Section 5.2).

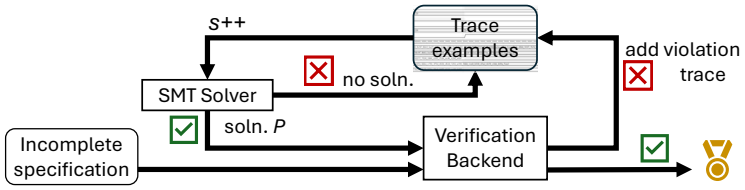

Fig. 6. CEGIS-based routine for synthesizing `ctrlhole`s.

4.3 Counterexample-guided Inductive Synthesis

The `self.ctrlhole(ctrl, dep)` hole is synthesized by generating a lookup-table that provides the value of the `dep` (dependent) signal as a function of the `ctrl` (controlling) signal. PyCaliper implements a Counterexample Guided Inductive Synthesis (CEGIS) based synthesis routine illustrated in Figure 6. The routine starts with a set of traces obtained from the design. Then it searches for a lookup table of size `s` of the form

```
(ite (= ctrl c_1) d_1 (ite (= ctrl c_2) d_2 ... d_s))
```

that matches the traces. It does so by invoking an SMT solver with `c_i` and `d_i` as free variables, and the traces as constraints. If the constraints are unsatisfiable, it means that the current lookup table is too small, and the size of the lookup table `s` is incremented (up to a bound). Otherwise the satisfying assignment (to `c_i` and `d_i`) provides a program P such that `dep` = P(`ctrl`) for these traces. This program is checked for inductiveness using a verification backend (Section 3). If verification fails, the trace counterexample is added to the set of traces, and the process is repeated again (up to a limit). Adding the trace guarantees that the (incorrect) lookup table candidate will not be produced in a subsequent iteration.

5 Case Studies

We present two case studies that demonstrate PyCaliper's ability to interface with both open-source and commercial verification backends, as well as its synthesis capabilities to complete missing parts of specifications.

5.1 Mitigating BTI with TAGE-based Predictor Isolation

In this case study we modify an open-source branch predictor design [2] to mitigate Branch Target Injection (BTI) [24] attacks. We then formally verify the modified design using PyCaliper's verification backends.

Branch Target Injection (BTI). Indirect branch predictors (IBPs) speculatively predict the targets of indirect branches using lookup tables. The Branch Target Injection (BTI) attack [24] allows a low-privileged (e.g., user-space) process to poison table entries, which allows malicious code execution in high-privileged (e.g., kernel) mode. BTI leverages the shared state maintained by the IBP across different privilege levels.

IsoTAGE: **TAGE-based Predictor Isolation.** To mitigate BTI, we augment the TAGE-based branch predictor design [2] by adding, to each prediction entry, a `domain` field that matches the privilege level of the process that created that entry. A prediction is only used if the `domain` field matches the current privilege level. This ensures that entries modified in low-privilege mode are not used in high-privilege mode, thus preventing cross-domain poisoning.

Formally Verifying IsoTAGE: How do the backends compare? We aim to verify the no-poisoning formally using PyCaliper. We formalize this as a property that says that provided the true branch target in privileged mode (`PRIV`) is in the valid kernel-code segment (less than a boundary `BDRY`), the generated prediction is also in the valid kernel-code segment. Formally, we identify the inductive specification:

$$\texttt{input} \equiv \texttt{mode} = \texttt{PRIV} \implies \texttt{targ\_i} < \texttt{BDRY}$$
$$\texttt{state} \equiv \forall i \in \texttt{IDXSET}.(\texttt{table[i].domain} = \texttt{PRIV} \implies \texttt{table[i].targ} < \texttt{BDRY})$$
$$\texttt{output} \equiv \texttt{prev\_mode} = \texttt{PRIV} \implies \texttt{targ\_o} < \texttt{BDRY}$$

We verify this property using 2-induction with both the SVA and BTOR2 backends and present the results in Table 4.

BHT_WIDTH	SVA Backend			BTOR2 Backend				
	TT (s)	ST (s)	PT (s)	TT (s)	ST (s)	PT (s)	Len. (#stmts)	Elems.
4	4.0	**2.6**	1.4	3.6	**2.6**	1	5465	598
5	3.5	**2.5**	1.0	6.2	**2.2**	4	9029	1046
6	3.9	**3.0**	0.9	16.1	**2.1**	14	16142	1942
7	5.1	**4.1**	1.0	55.8	**3.8**	52	30376	3734
8	7.8	**6.9**	0.9	196.1	**6.1**	190	58915	7318
9	12.7	**11.4**	1.3	753.9	**11.9**	742	116186	14486

Table 4. Performance comparison of the SVA and BTOR2 backends on the IsoTAGE benchmark. Legend: TT = total time, ST = solving time, PT = (design) parsing time.

Interestingly, we observe that the solving times (ST) of the BTOR2 backend (using the Boolector solver [12]) is comparable to SVA backend using the Jasper FPV App. However, the parse time (PT) is where BTOR2 backend falls behind. This is likely due to the fact that we currently use a Python-based `btor2` parsing library - we plan to report this result and improve the parsing performance. This insightful comparison was possible due to PyCaliper's compatibility with both verification backends.

5.2 Synthesizing Invariants for Dynamic Cache-partitioning

In the second case study, we perform invariant synthesis for the DAWG (Dynamically Allocated Way Guard) [27] mitigation, that partitions cache ways to ensure that there is

no cross-domain information leakage (formulated as a non-interference property [48]). Previous work [22] performs unbounded verification of this property by manually identifying invariants. We apply the incremental invariant synthesis engine (Section 4.2) to synthesize k out of 8 cache way invariants for $k = 2, 3, 4$. This is a very challenging problem as the "invariant-width" [36], i.e., the number of invariants that need to be added together to ensure self-inductiveness increases with k.

We attempt two strategies for selecting the invariant candidate order: random (in-built) and a custom (rudimentary) neural strategy. In the latter, we prompt an LLM (gpt-4o) to provide an order of candidates, providing the current set of invariants and the cache partitioning property as context. Implementing this strategy with PyCaliper required ∼80 lines of Python code and around 30 minutes of human effort.

In Table 5, we present the number of verification calls (#VC) and the number of nodes explored (#nodes) in the search as an average computed over 20 runs. We use the SVA-backend with the Jasper FPV App as the verification engine. We see that even our very rudimentary neural strategy outperforms the random strategy, likely by being able to detect a pattern in the invariants using the provided context. The takeaways from this experiment are twofold. Firstly, PyCaliper allows leveraging industry-scale backends as verification oracles in an invariant synthesis task. Secondly, PyCaliper uses the Python ecosystem to enable easy prototyping of new ideas.

	Random		Neural	
k	#VC $\pm\ \sigma$	#nodes $\pm\ \sigma$	#VC $\pm\ \sigma$	#nodes $\pm\ \sigma$
2	16.7 ± 2.3	5.5 ± 0.6	14.3 ± 2.6	4.9 ± 0.9
3	101.6 ± 47.1	13.3 ± 5.5	54.0 ± 21.8	8.5 ± 3.9
4	785.7 ± 91.2	54.3 ± 4.8	163 ± 74.2	11.6 ± 8.6

Table 5. IIS (Section 4.2) with order strategies.

6 Conclusion

PyCaliper's ability to leverage Pythonic introspection, meta-programming and object-oriented features enables easy, flexible and reusable (packageable) RTL specification that is close to the implementation. Further, PyCaliper provides verification backends which can interface both industry-scale tooling as well as custom/open-source verification engines. PyCaliper uses these verification backends as oracles to implement (parameterizable) synthesis techniques to auto-generate missing parts of specifications. Thus, PyCaliper supports holistic *hardware specification engineering* as a combination of accessible and flexible hardware specification, verification and synthesis.

Acknowledgments. We thank Federico Mora, Pei-Wei Chen (UC Berkeley), Yatin A. Manerkar (Univ. of Michigan), Elizabeth Polgreen (Univ. of Edinburgh), and Scott Constable (Intel Labs), John Matthews (formerly Intel Labs) for their feedback. This work was supported in part by Intel Corporation under the Scalable Assurance program, DARPA contract FA8750-23-C-0080 (ANSR), and by the iCyPhy center.

Disclosure of Interests. The authors declare that they have no interests other than their affiliations and funding sources mentioned in the acknowledgments.

References

1. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) pp. 1–1315 (2018)
2. Aaron Shappell: tage-predictor. https://github.com/aaronshappell/tage-predictor (2024), GitHub repository.
3. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), https://ieeexplore.ieee.org/document/6679385/
4. Appel, A.W., Beringer, L., Chlipala, A., Pierce, B.C., Shao, Z., Weirich, S., Zdancewic, S.: Position paper: the science of deep specification. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 375(2104), 20160331 (2017), https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2016.0331
5. Baaij, C., Kooijman, M., Kuper, J., Boeijink, A., Gerards, M.: C?ash: Structural descriptions of synchronous hardware using haskell. In: 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools. pp. 714–721 (2010)
6. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, chap. 26, pp. 825–885. IOS Press (2009)
7. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
8. Blume, M., Appel, A.W.: Hierarchical modularity. ACM Trans. Program. Lang. Syst. 21(4), 813–847 (Jul 1999), https://doi.org/10.1145/325478.325518
9. Bradley, A.R.: Sat-based model checking without unrolling. In: Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 70–87. VMCAI'11, Springer-Verlag, Berlin, Heidelberg (2011)
10. Bradley, A.R.: Understanding ic3. In: Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing. pp. 1–14. SAT'12, Springer-Verlag, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-31612-8_1
11. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Aspects Comput. 20(4-5), 379–405 (2008), https://doi.org/10.1007/s00165-008-0080-9
12. Brummayer, R., Biere, A.: Boolector: An efficient smt solver for bit-vectors and arrays. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,. pp. 174–177. TACAS '09, Springer-Verlag, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-642-00768-2_16
13. Cadence: Jasper FPV App. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-verification-platform/formal-property-verification-app.html (2024)
14. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: a platform for high-level parametric hardware specification and its modular verification. Proc. ACM Program. Lang. 1(ICFP) (Aug 2017), https://doi.org/10.1145/3110268
15. Claire Wolf, et. al.: Symbiyosys. https://github.com/YosysHQ/sby (2022)
16. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing. pp. 240–248. PODC '86, Association for Computing Machinery, New York, NY, USA (1986), https://doi.org/10.1145/10590.10611

17. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. 18(6), 1157–1210 (Sep 2010)
18. cocotb: cocotb. https://www.cocotb.org/, https://docs.cocotb.org/en/stable/# (2024), Webpage.
19. diffblue: EBMC: Enhanced Bounded Model Checker. https://www.cprover.org/ebmc/, https://github.com/diffblue/hw-cbmc (2024), GitHub repository.
20. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity. pp. 500–517. FME '01, Springer-Verlag, Berlin, Heidelberg (2001)
21. Godbole, A., Cheang, K., Manerkar, Y.A., Seshia, S.A.: Lifting micro-update models from RTL for formal security analysis. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. pp. 631–648. ASPLOS '24, Association for Computing Machinery, New York, NY, USA (2024), https://doi.org/10.1145/3620665.3640418
22. Godbole, A., Ye, L., Manerkar, Y.A., Seshia, S.A.: Modelling and verification of security-oriented resource partitioning schemes. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023. pp. 268–273. IEEE (2023), https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_35
23. Goel, A., Sakallah, K.A.: AVR: Abstractly Verifying Reachability. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 413–422. Springer (2020), https://doi.org/10.1007/978-3-030-45190-5_23
24. Intel Corporation: Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html (2018), Intel Advisory
25. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. pp. 215–224. ICSE '10, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1806799.1806833
26. Jha, S., Seshia, S.A.: A Theory of Formal Synthesis via Inductive Learning. Acta Informatica 54(7), 693–726 (2017)
27. Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., Emer, J.: Dawg: a defense against cache timing attacks in speculative execution processors. In: Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture. pp. 974–987. MICRO-51, IEEE Press (2018), https://doi.org/10.1109/MICRO.2018.00083
28. Lattner, et. al.: CIRCT: Circuit IR Compilers and Tools. https://circt.llvm.org/, https://github.com/llvm/circt (2025)
29. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.: Pono: A Flexible and Extensible SMT-Based Model Checker. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. pp. 461–474. Springer-Verlag, Berlin, Heidelberg (2021), https://doi.org/10.1007/978-3-030-81688-9_22
30. Michael Popoloski: slang. https://sv-lang.com/ (2024)
31. myhdl: MyHDL. https://www.cocotb.org/, https://docs.cocotb.org/en/stable/# (2024), Webpage.

32. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 284–296. Springer-Verlag, Berlin, Heidelberg (1997), https://doi.org/10.1007/BFb0058037
33. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 284–296. Springer-Verlag, Berlin, Heidelberg (1997), https://doi.org/10.1007/3-540-58179-0_44
34. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018), https://doi.org/10.1007/978-3-319-96145-3_32
35. Nigam, R., Atapattu, S., Thomas, S., Li, Z., Bauer, T., Ye, Y., Koti, A., Sampson, A., Zhang, Z.: Predictable accelerator design with time-sensitive affine types. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 393–407. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3385412.3385974
36. Padon, O., Wilcox, J.R., Koenig, J.R., McMillan, K.L., Aiken, A.: Induction duality: primal-dual search for invariants. Proc. ACM Program. Lang. 6(POPL) (Jan 2022), https://doi.org/10.1145/3498712
37. Pnueli, A.: In Transition From Global to Modular Temporal Reasoning about Programs. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984. NATO ASI Series, vol. 13, pp. 123–144. Springer (1984), https://doi.org/10.1007/978-3-642-82453-1_5
38. Polgreen, E., Cheang, K., Gaddamadugu, P., Godbole, A., Laeufer, K., Lin, S., Manerkar, Y.A., Mora, F., Seshia, S.A.: UCLID5: multi-modal formal modeling, verification, and synthesis. In: 34th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 13371, pp. 538–551. Springer (2022)
39. Polgreen, E., Reynolds, A., Seshia, S.A.: Satisfiability and synthesis modulo oracles. CoRR abs/2107.13477 (2021), https://arxiv.org/abs/2107.13477
40. Semiconductor Engineering: Big Trouble at 3nm. https://semiengineering.com/big-trouble-at-3nm/ (2018)
41. Semiconductor Engineering: Verification IP (VIP). https://semiengineering.com/knowledge_centers/intellectual-property/verification-ip-vip/ (2024)
42. Seshia, S.A.: Sciduction: Combining induction, deduction, and structure for verification and synthesis. In: Proceedings of the Design Automation Conference (DAC). pp. 356–365 (June 2012)
43. Seshia, S.A.: Combining induction, deduction, and structure for verification and synthesis. Proceedings of the IEEE 103(11), 2036–2051 (2015)
44. Seshia, S.A., Subramanyan, P.: Uclid5: Integrating modeling, verification, synthesis, and learning. In: Proceedings of the 15th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE) (October 2018)
45. Solar-Lezama, A.: The sketching approach to program synthesis. In: Proceedings of the 7th Asian Symposium on Programming Languages and Systems. pp. 4–13. APLAS '09, Springer-Verlag, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-642-10672-9_3
46. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 404–415. ASPLOS XII,

Association for Computing Machinery, New York, NY, USA (2006), https://doi.org/10.1145/1168857.1168907
47. Synopsys: Formality Equivalence Checking. https://www.synopsys.com/implementation-and-signoff/signoff/formality-equivalence-checking.html (2024)
48. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Proceedings of the 12th International Conference on Static Analysis. pp. 352–367. SAS'05, Springer-Verlag, Berlin, Heidelberg (2005), https://doi.org/10.1007/11547662_24
49. The Rocq/Coq Development Team: Rocq Prover. https://rocq-prover.org/, https://coq.inria.fr/ (2024), Webpage.
50. Vicarte, J.R.S., Flanders, M., Paccagnella, R., Garrett-Grossman, G., Morrison, A., Fletcher, C.W., Kohlbrenner, D.: Augury: Using data memory-dependent prefetchers to leak data at rest. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022. pp. 1491–1505. IEEE (2022), https://doi.org/10.1109/SP46214.2022.9833570
51. Viswanathan, M., Viswanathan, R.: Foundations for Circular Compositional Reasoning. In: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,. pp. 835–847. ICALP '01, Springer-Verlag, Berlin, Heidelberg (2001)
52. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. pp. 260–276. Springer-Verlag, Berlin, Heidelberg (2014), https://doi.org/10.1007/978-3-319-08867-9_17
53. Wolf, C., Glaser, J., Kepler, J.: Yosys-A Free Verilog Synthesis Suite (2013)
54. YosysHQ: Mutation Cover with Yosys. https://github.com/YosysHQ/mcy (2024), GitHub repository.
55. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: 16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA. p. 29. IEEE Computer Society (2003), https://doi.org/10.1109/CSFW.2003.1212703
56. Zhang, H., Yang, W., Fedyukovich, G., Gupta, A., Malik, S.: Synthesizing environment invariants for modular hardware verification. In: Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings. pp. 202–225. Springer-Verlag, Berlin, Heidelberg (2020), https://doi.org/10.1007/978-3-030-39322-9_10