Unified Deductive Hardware Verification

Amelia Dobis

PhD Student - Advisor: Mae Milano - Princeton PL/SNS

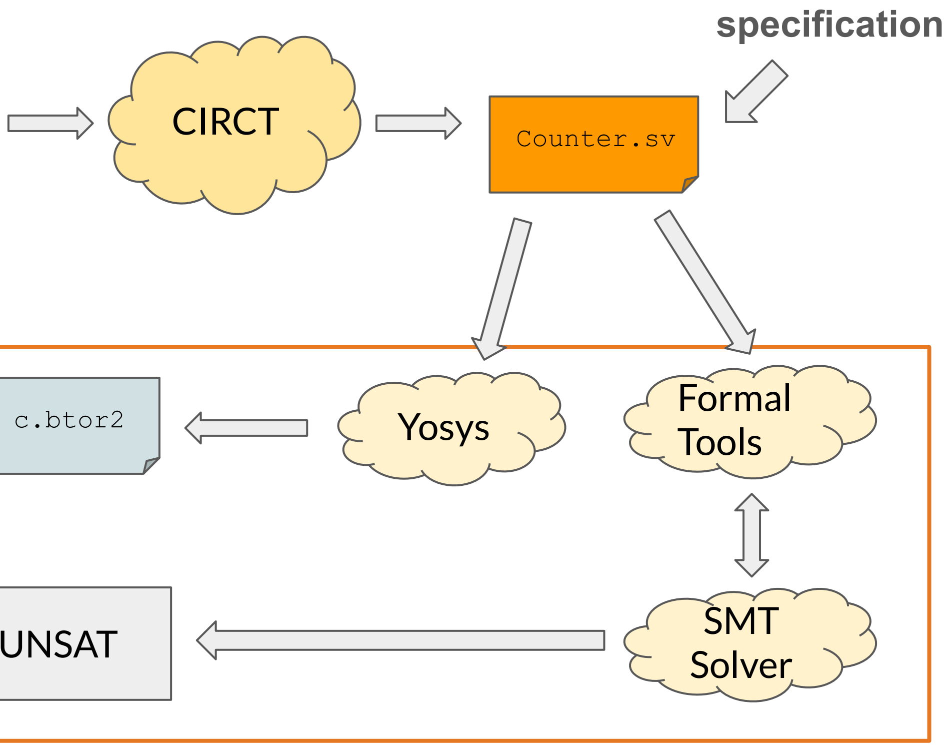
OPLSS 2025

Motivation: Practical Reality of Hardware Design

- Hardware is **hard**: Very difficult to implement correctly
- Mistakes are **expensive**: Tape-out costs millions of dollars
- We need **strong correctness guarantees** that dynamic testing methods don't give us
  - *<insert Dijkstra quote here>*

Motivation: Hardware Verification is Terrible

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
}
```

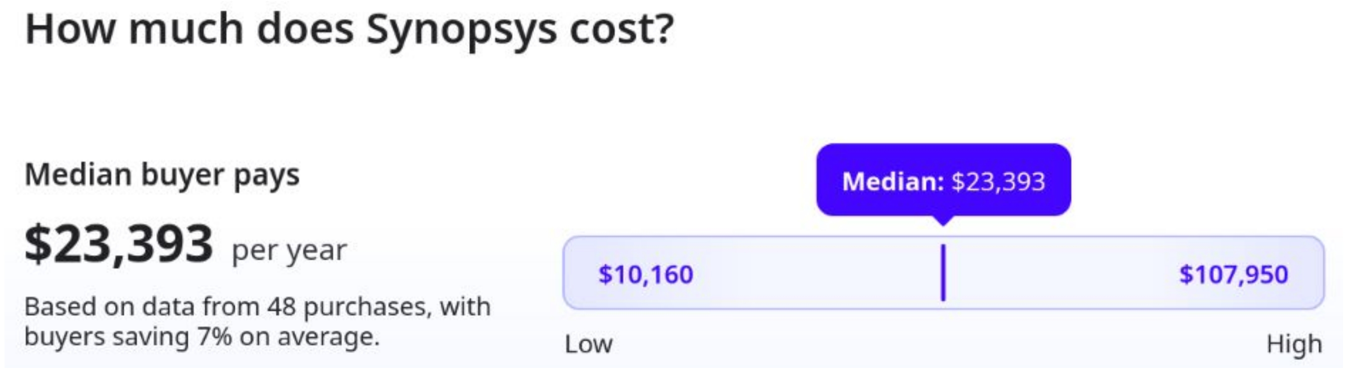


TOOLS FOR SV NOT YOUR LANGUAGE!!

Motivation: “reasons” why HW Verification is bad

- academia: “Hardware Verification is a solved problem just use BMC”
  – Peter Müller, Spring 2022

- industry: “You should use Synopsis VC Formal”
  – Every verification engineer, all day every day



How much does Synopsys cost?

Median buyer pays

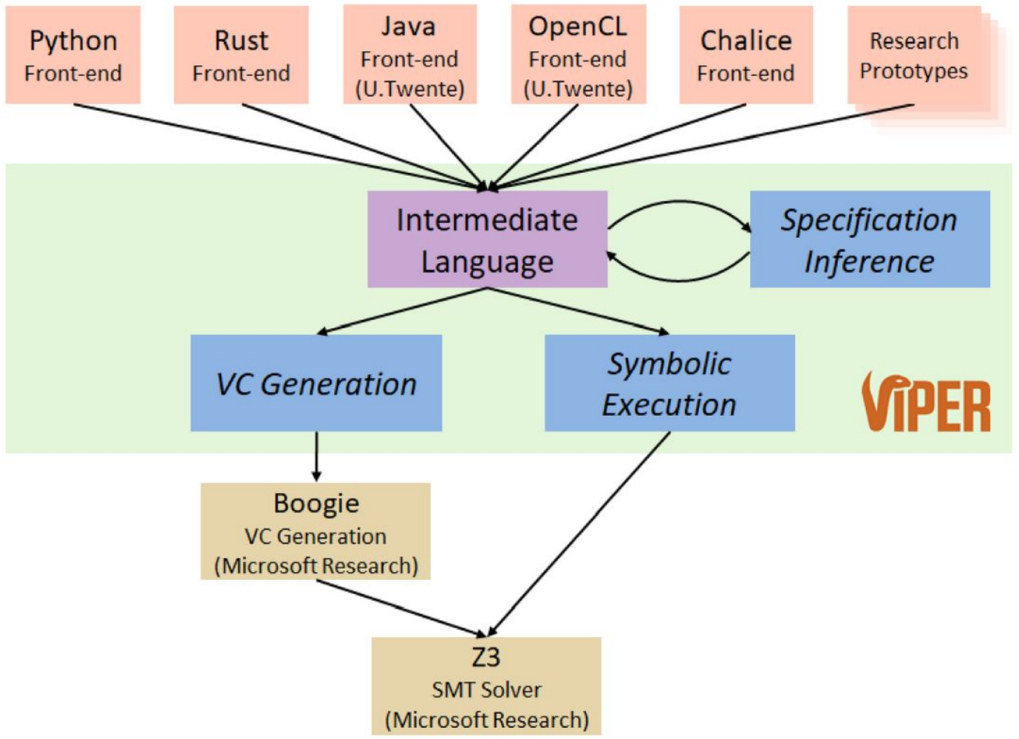
$23,393 per year

Based on data from 48 purchases, with buyers saving 7% on average.

Motivation: Deductive Program Verification is cool

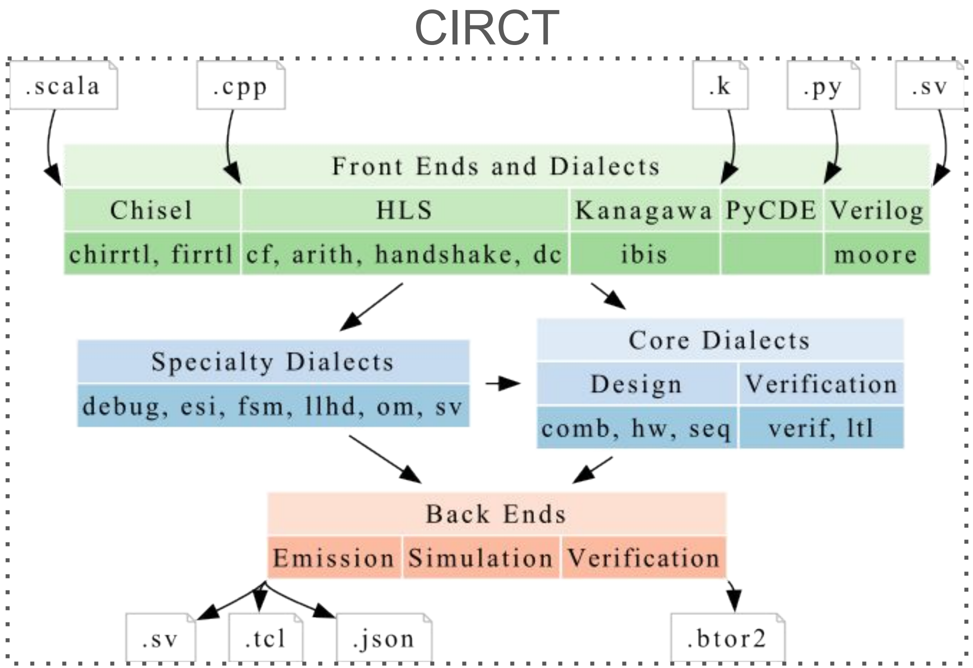
- Meanwhile: Software get to use interactive modular verification tools.
  - here's a demo
- These are unified and are adaptable to multiple (frontend) languages.
- **Hardware engineers deserve nice things too!**

Background: Deductive Program Verifiers



[1] VIPER TUTORIAL, http://viper.ethz.ch/tutorial/

Background: CIRCT



[2] CIRCT: Intermediate Representations as a Shared Foundation for Hardware Compilation; Amelia Dobis, Bea Healy, Schuyler Eldridge, Tobias Grosser, Andrew Lenharth, Andrew Young, Chris Lattner, Fabian Schuiki, Hideto Ueno, John Demme, Julian Oppermann, Lenny Truong, Leon Hielscher, Mae Milano, Martin Erhart, Mike Urbach, Morten Borup Petersen, Prithayan Barua, Robert Young, Stephen Neuendorffer, Will Dietz; 2025

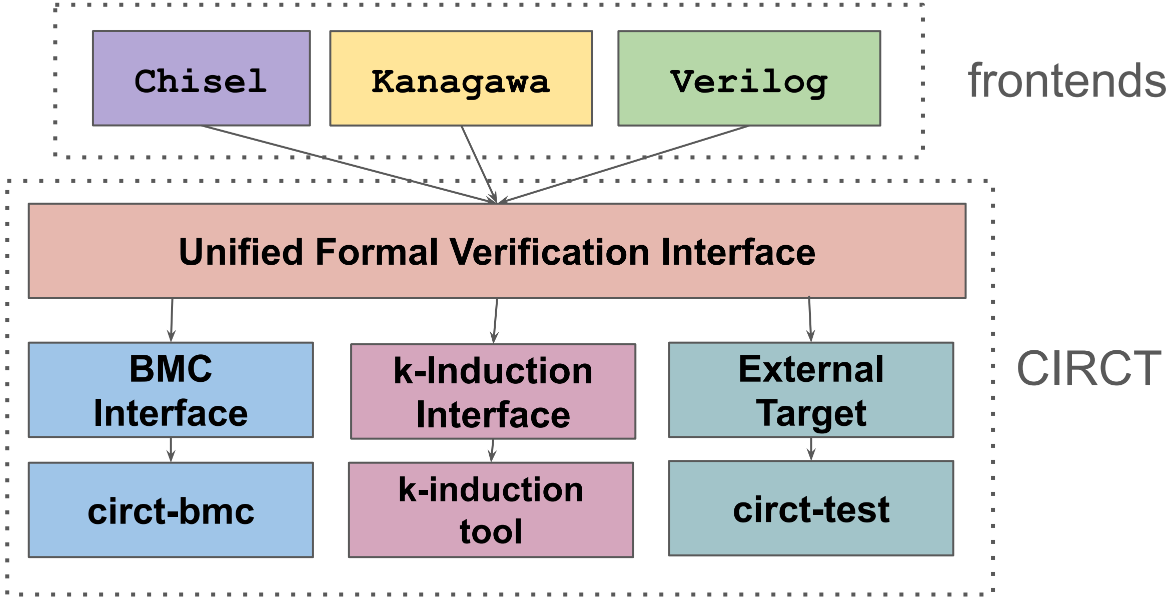
Goal: Deductive Hardware Verification

- Can we create a “Viper for Hardware”?
- How does **Program Verification** differ from **Hardware Verification?**
- What **underlying methods** do we need?
- Can we **bypass SystemVerilog**?

Unified Deductive Hardware Verification

Unified	**Deductive**	**Hardware Verification**
Single Interface	*Verification Condition Generation*	*Supports Sequential Designs*
Supports many front-ends	*Weakest Precondition computation*	*Supports Temporal Logic*
Supports many backends	*Maintains Modularity*	*Supports Parallel Verification*

Unified Deductive Hardware Verification



Unified Deductive Hardware Verification

- **How?** → Introduce *First Class Verification ops* to the CIRCT compiler

```
verif_op ::= assert <s> <clk> | assume <s> <clk> | cover <s> <clk>
           | formal <@sym> <body> | <s> = symbolic_input
op ::= verif.verif_op : <type>
```

```
class formalTest extends Module with Formal {
    // Inputs are interpreted as free/symbolic
    val a = IO(Input(UInt(32.W)))
    …
}
```

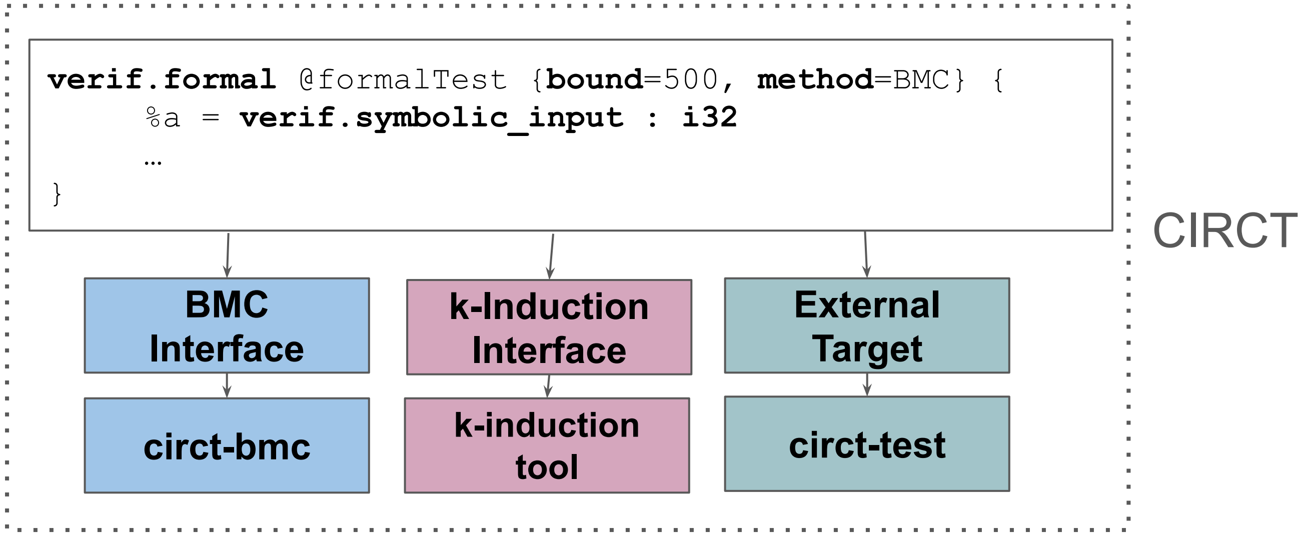
Chisel

↓

```
verif.formal @formalTest {bound=500, method=BMC} {
    %a = verif.symbolic_input : i32
    …
}
```

MLIR

Unified Deductive Hardware Verification



```
verif.formal @formalTest {bound=500, method=BMC} {
      %a = verif.symbolic_input : i32
      …
}
```

Unified Deductive Hardware Verification

Unified	**Deductive**	**Hardware Verification**
Single Interface 🙂	*Verification Condition Generation*	*Supports Sequential Designs*
Supports many front-ends 🙂	*Weakest Precondition computation*	*Supports Temporal Logic*
Supports many backends 🙂	*Maintains Modularity*	*Supports Parallel Verification*

Unified **Deductive** Hardware Verification

- **Goal:** Maintain modularity during verification
- **Problem:** Current verification tools duplicate verification tasks for module instances.
- **Solution:** Introduce modularity into the specification language.
    - Hoare Logic can help with this!

Unified **Deductive** Hardware Verification

- **Extend verif dialect to include hoare triples.**
- `%out = verif.contract (<inputs>) {<body>}`
  - declares a Hoare Triple → inputs will be abstracted during verification.
  - Output is the result that can referenced in postconditions
- `verif.requires %precondition : <type>`
  - declares a **precondition**
  - Only valid inside of a verif.contract body
- `verif.ensures %postcondition : <type>`
  - declares a **postcondition**
  - Only valid inside of a verif.contract body

Unified **Deductive** Hardware Verification

```
class A extends Module {
        val in = IO(Input(UInt(32.W)))
        val out = IO(Output(UInt(32.W)))
        contract {
                requires in >= 0.U
                ensures out === in + 42.U
        }
        // ... Module body ...
}
```

```
hw.module @A (in %in: i32, out %out: i32) {
        ;; Module body defining res
        %out = verif.contract %res {
                %c0 = hw.constant 0: i32
                %gt = comb.icmp bin ugte %in, %c0
                verif.requires %gt : i1
                %c42 = hw.constant 42 : i32
                %in42 = comb.add bin %in, %c42 : i32
                %post = comb.icmp bin eq %res, %in42
                verif.ensures %post
        }
        hw.output %out
}
```

Unified **Deductive** Hardware Verification

```
hw.module @A (in %in: i32, out %out: i32) {
     ;; Module body defining res
     %out = verif.contract %res {
          …
          verif.requires %gt : i1
          …
          verif.ensures %post
     }
     hw.output %out
}
```

What do we do with this?
→ Verification Condition Generation

Unified **Deductive** Hardware Verification

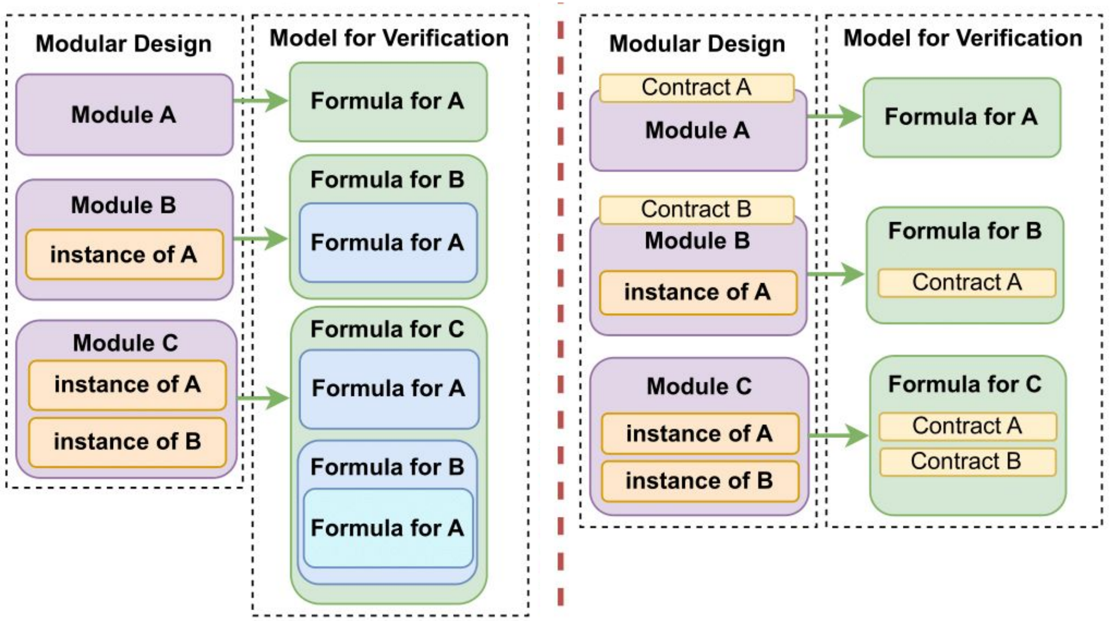
```
hw.module @A (in %in: i32, out %out: i32) {
     ;; Module body defining res
     %out = verif.contract %res {
          …
          verif.requires %gt : i1
          …
          verif.ensures %post
     }
     hw.output %out
}
```

What do we do with this?

~~**→ Verification Condition Generation**~~

→ BMC Problem generation

Unified **Deductive** Hardware Verification



Unified **Deductive** Hardware Verification

- Goal: Generate **verif.formal** tests for every module.

- Convert modules into formal tests by:
    - Replace **inputs** and **outputs** with **symbolic variables**
    - **Assume** all **preconditions** on the inputs
    - **Assert** all **postconditions** on the outputs

- Replace module instances with their contracts where:
    - All **preconditions** are **asserted** on the inputs given to the instance
    - All **postconditions** are **assumed** on the result of the instance

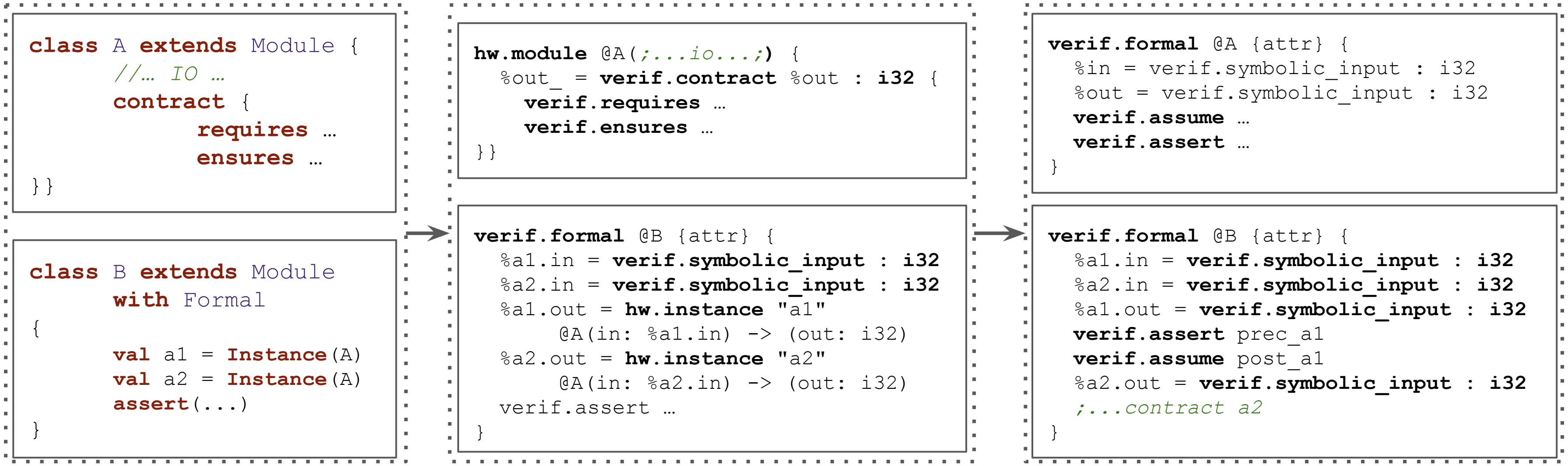
Unified **Deductive** Hardware Verification

```
hw.module @A (in %in: i32, out %out: i32) {
     ;; Module body defining res
     %out = verif.contract %res {
          …
          verif.requires %gt
          …
          verif.ensures %post
     }
     hw.output %out
}
```

```
verif.formal @A {...} {
     ;; Module body defining res
     %in = verif.symbolic_input : i32
     %out = verif.symbolic_input : i32
     …
     verif.assume %gt
     …
     verif.assert %post
}
```

Unified **Deductive** Hardware Verification

- Verification Compilation Flow (Weakest Precondition Computation):



```
class A extends Module {
      //… IO …
      contract {
            requires …
            ensures …
}}
```

```
class B extends Module
      with Formal
{
      val a1 = Instance(A)
      val a2 = Instance(A)
      assert(...)
}
```

frontend

```
hw.module @A(;...io...;) {
  %out_ = verif.contract %out : i32 {
    verif.requires …
    verif.ensures …
}}
```

```
verif.formal @B {attr} {
  %a1.in = verif.symbolic_input : i32
  %a2.in = verif.symbolic_input : i32
  %a1.out = hw.instance "a1"
       @A(in: %a1.in) -> (out: i32)
  %a2.out = hw.instance "a2"
       @A(in: %a2.in) -> (out: i32)
  verif.assert …
}
```

CIRCT core IR

```
verif.formal @A {attr} {
  %in = verif.symbolic_input : i32
  %out = verif.symbolic_input : i32
  verif.assume …
  verif.assert …
}
```

```
verif.formal @B {attr} {
  %a1.in = verif.symbolic_input : i32
  %a2.in = verif.symbolic_input : i32
  %a1.out = verif.symbolic_input : i32
  verif.assert prec_a1
  verif.assume post_a1
  %a2.out = verif.symbolic_input : i32
  ;...contract a2
}
```

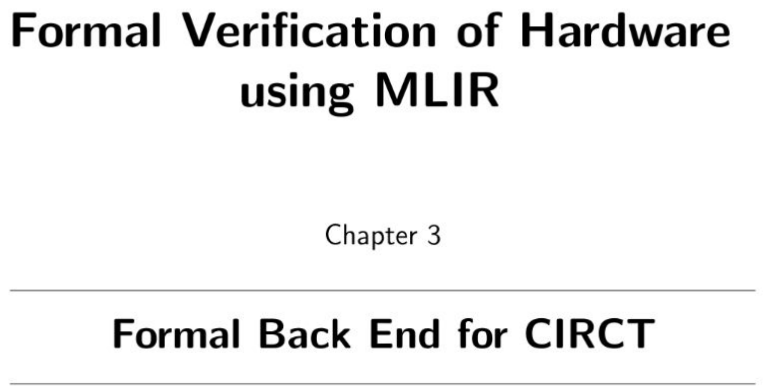
CIRCT verification IR

Unified Deductive Hardware Verification

Unified	**Deductive**	**Hardware Verification**
Single Interface ☺	*Verification Condition Generation* ☺	*Supports Sequential Designs*
Supports many front-ends ☺	*Weakest Precondition computation* ☺	*Supports Temporal Logic*
Supports many backends ☺	*Maintains Modularity* ☺	*Supports Parallel Verification*

Unified Deductive **Hardware Verification**

- Goal: Allow for generation of Bounded Model Checking (BMC) problems from CIRCT.
- How? Lower to **BTOR2** from the CIRCT verification IR.
- idea: Convert design into state-transition system + FOL

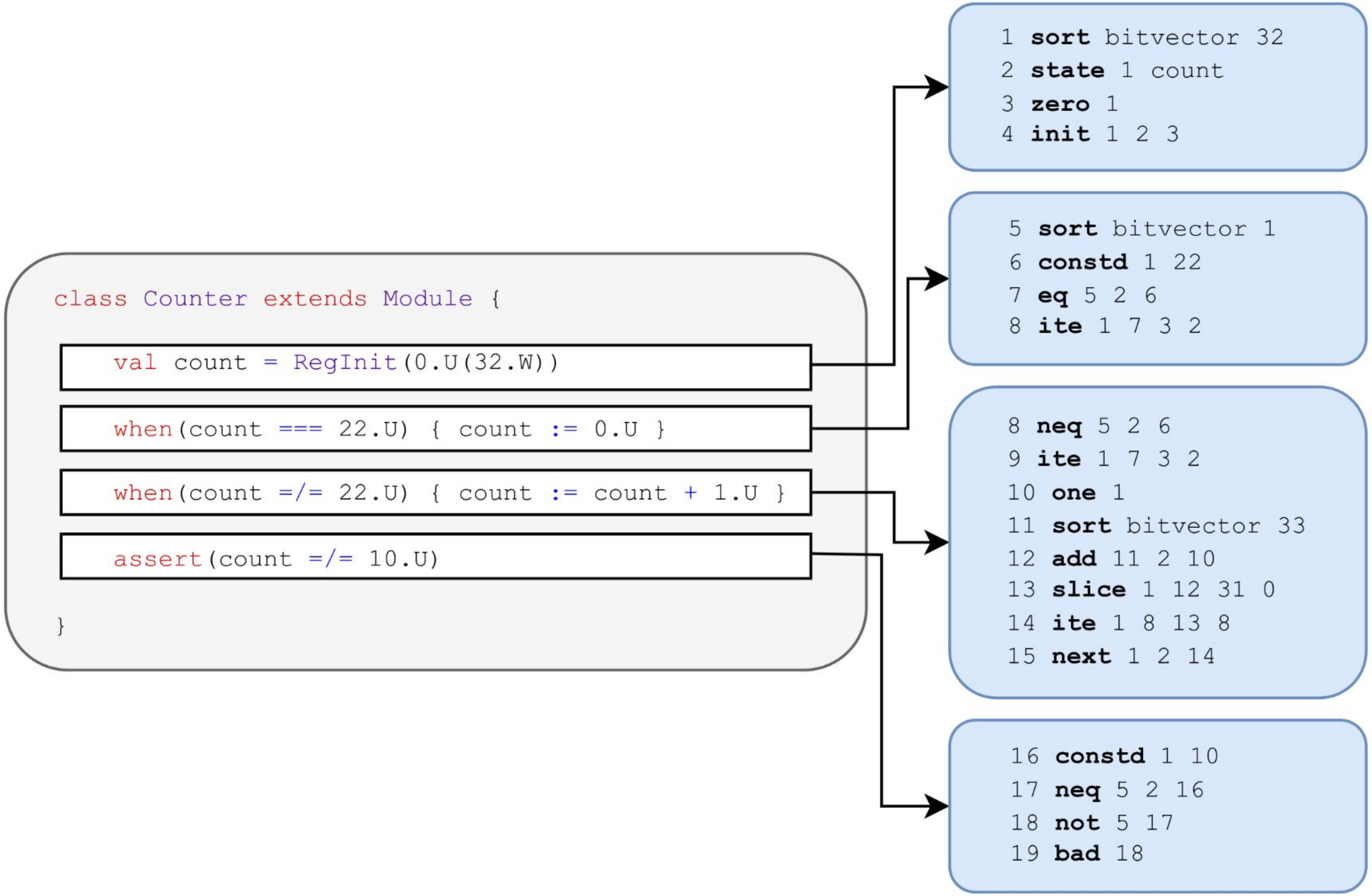


Formal Verification of Hardware using MLIR

Chapter 3

Formal Back End for CIRCT

[3] Formal Verification of Hardware using MLIR, Amelia Dobis, Master Thesis, ETHZ 2023-2024



```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 22.U) { count := 0.U }
    when(count =/= 22.U) { count := count + 1.U }
    assert(count =/= 10.U)
}
```

```
1 sort bitvector 32
2 state 1 count
3 zero 1
4 init 1 2 3
```

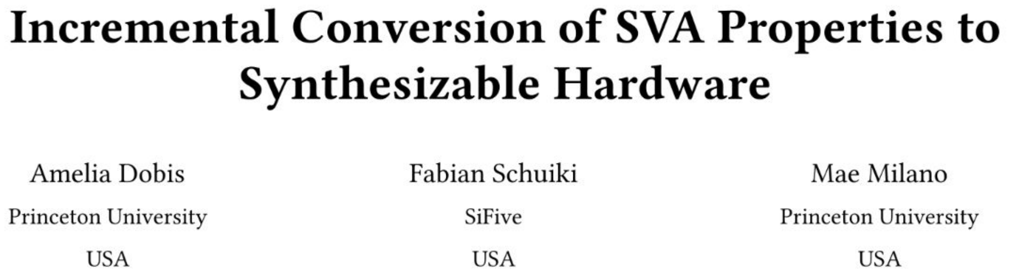
```
5 sort bitvector 1
6 constd 1 22
7 eq 5 2 6
8 ite 1 7 3 2
```

```
8 neq 5 2 6
9 ite 1 7 3 2
10 one 1
11 sort bitvector 33
12 add 11 2 10
13 slice 1 12 31 0
14 ite 1 8 13 8
15 next 1 2 14
```

```
16 constd 1 10
17 neq 5 2 16
18 not 5 17
19 bad 18
```

Unified Deductive **Hardware Verification**

- Goal: Support Temporal Logic in Specifications.
- How: Design a "reactive" IR that encodes LTL through small incremental transformations.
- idea: encode LTL expression as "triggering asynchronous blocks".



Incremental Conversion of SVA Properties to Synthesizable Hardware

Amelia Dobis
Princeton University
USA

Fabian Schuiki
SiFive
USA

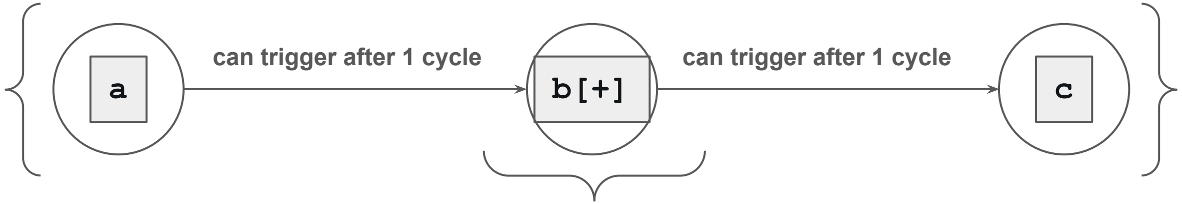
Mae Milano
Princeton University
USA

[4] Incremental Conversion of SVA Properties to Synthesizable Hardware; Amelia Dobis, Fabian Schuiki, Mae Milano; 2025

Unified Deductive **Hardware Verification**

```
a ##1 b[+] ##1 c
```

trigger: 1
valid: observe `a`

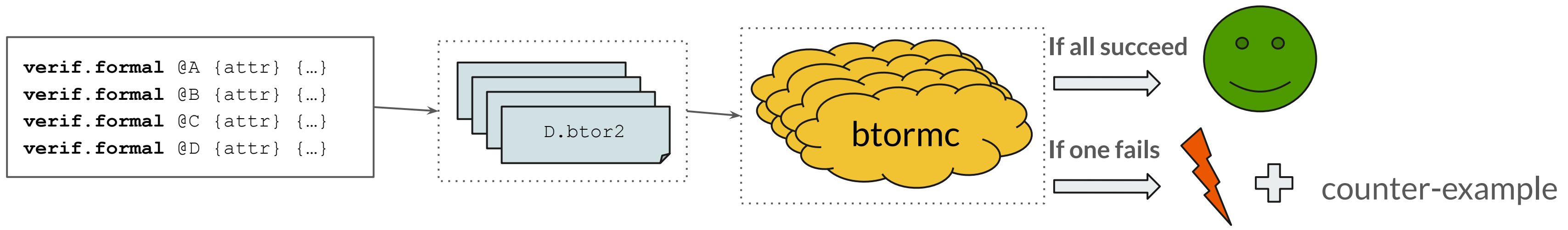


trigger: observe `reg(valid(b+))`
valid: observe `c & has_been_triggered`

trigger: observe `reg(valid(a))`
`has_been_triggered:` `trigger | has_been_triggered`
`has_observed_b:` `has_been_triggered & (has_observed_b | b)`
valid: observe `has_observed_b & has_been_triggered`

[4] Incremental Conversion of SVA Properties to Synthesizable Hardware; Amelia Dobis, Fabian Schuiki, Mae Milano; 2025

Unified Deductive **Hardware Verification**


verif.formal @A {attr} {…}
verif.formal @B {attr} {…}
verif.formal @C {attr} {…}
verif.formal @D {attr} {…}
D.btor2
btormc
If all succeed
If one fails
counter-example

- Enables Solver Parallelism
- Simplifies individual verification problems
  - No single verification task needs to solve for the entire system

Unified Deductive Hardware Verification

Unified	**Deductive**	**Hardware Verification**
Single Interface ☺	*Verification Condition Generation* ☺	*Supports Sequential Designs* ☺
Supports many front-ends ☺	*Weakest Precondition computation* ☺	*Supports Temporal Logic* ☺
Supports many backends ☺	*Maintains Modularity* ☺	*Supports Parallel Verification* ☺

Conclusion

- Introduced the concept of **Deductive Hardware Verification.**
  - **idea**: use small bmc problems in a similar way as SMT Solver Queries
- Designed a **unified** system to support many hardware languages.
- Implemented System as part of the **CIRCT Core.**
- Tooling not perfect but a good start to make hardware verification as efficient as program verification.

Resources

CIRCT: Final MLIR implementation of language constructs

https://github.com/llvm/circt

Formal Verification of Hardware using MLIR, ETHZ Master Thesis

https://doi.org/10.3929/ethz-b-000668906

Incremental Conversion of SVA Properties to Synthesizable Hardware, LATTE'2025

https://capra.cs.cornell.edu/latte25/paper/14.pdf