Scalable Formal Verification in High-Level Hardware Languages

Amelia Dobis *- PhD Student, Princeton University*
Advised by Mae Milano

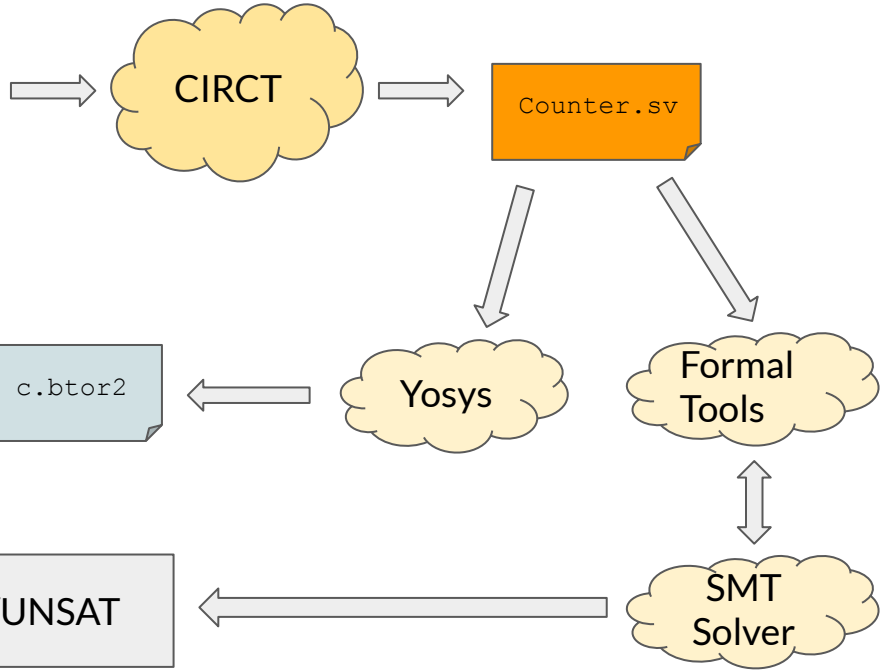
PRINCETON UNIVERSITY

Motivation: Formal Verification

- Hardware design is difficult -> **error prone.**
- Tape-out is expensive -> **no such thing as "affordable mistakes" .**
- Formal Verification gives **strong correctness guarantees**.
  - Based on SMT solvers like z3 or bitwuzla.

Motivation: Current Approach

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



Motivation: Problems with Current Approach

- Formal Verification tools are mostly **commercial** and **closed-source.**
- They **do not scale** well for large designs.
  - Engineers rely on manual workarounds and simulation instead.
- They are designed for **SystemVerilog**, not the high-level source languages.
- We need an **open-source** solution that is **fully integrated** into the languages engineers write.

Goal: Unified Formal Verification System

Goal: Design a verification system that is:

1. **Generic:** Single interface that works for all high-level hardware languages.

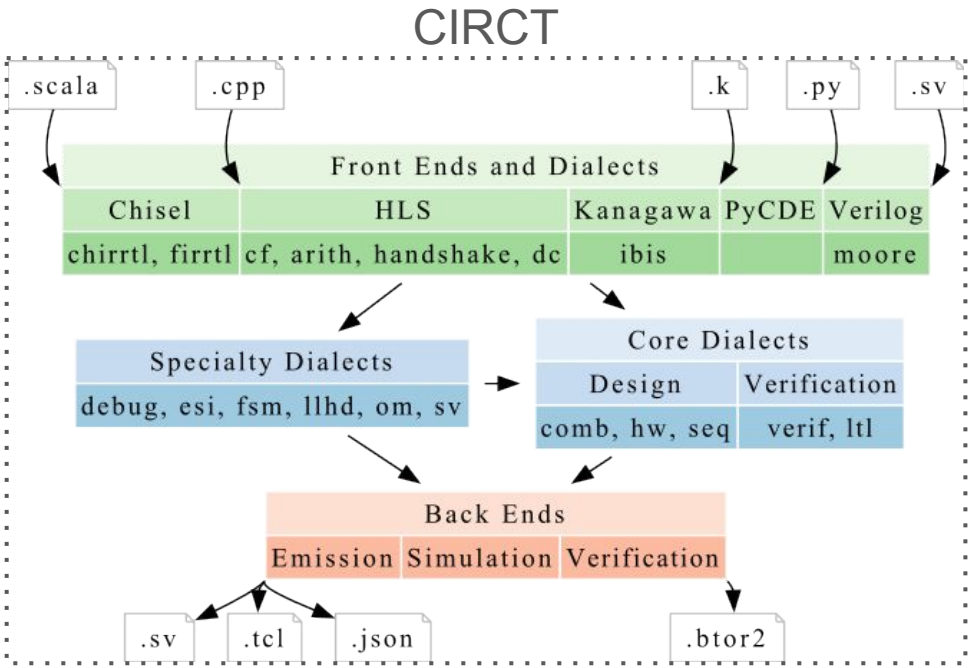
2. **Scalable:** Efficiently verifies large modular designs.

3. **Expressive:** Allows for designs to be accurately specified.

Background

Background: CIRCT

- **MLIR-based** Compiler for high-level hardware languages, e.g. Chisel, Kanagawa, Verilog.
- Lowers all design constructs down to a shared **core representation**.
- Good target to create **features** that can be **shared** across many hardware languages.



Overview of our Solution

Goal: Unified Formal Verification System

Goal: Design a verification system that is:

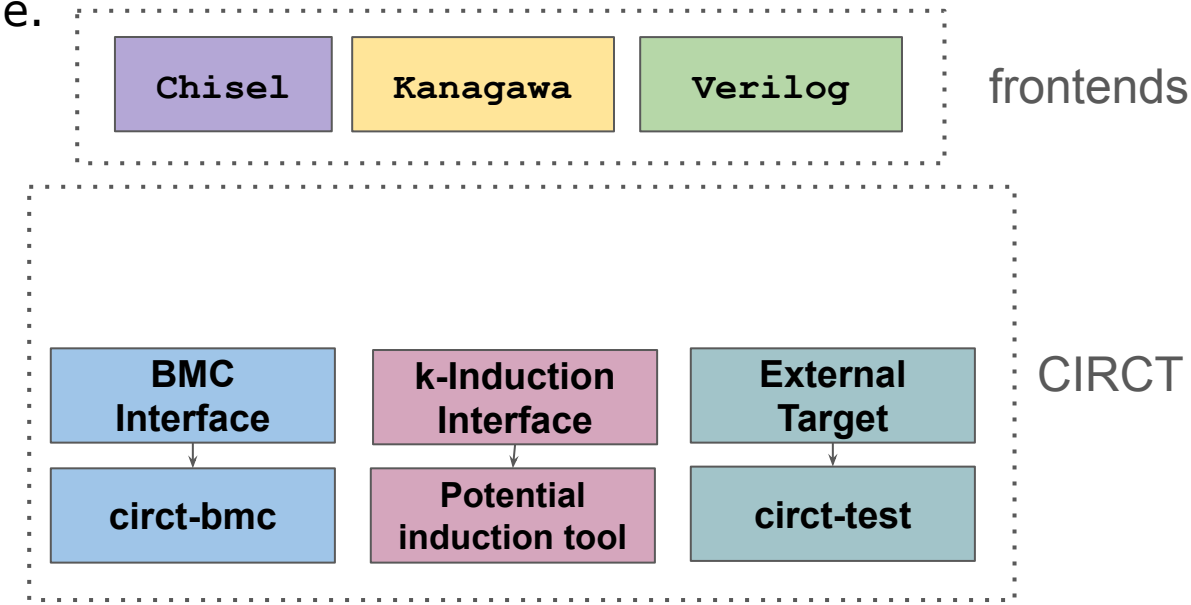
1. **Generic:** Single interface that works for all high-level hardware languages.

2. **Scalable:** Efficiently verifies large modular designs.

3. **Expressive:** Allows for designs to be accurately specified.

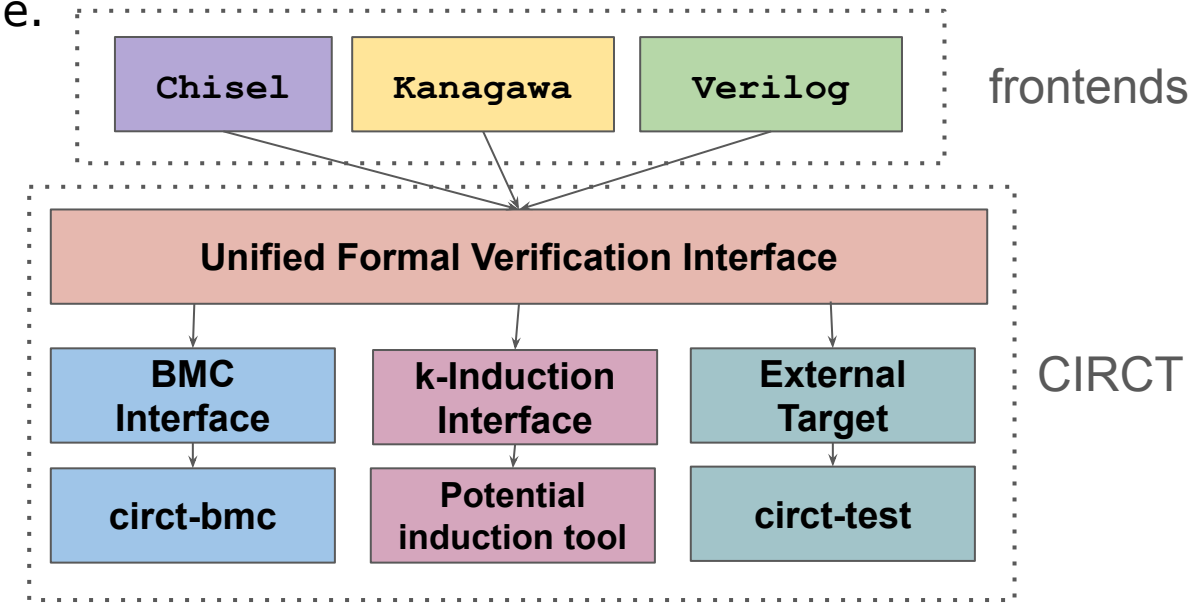
Generic: Single Interface for Formal Verification

- **Goal:** Single interface to encode a formal test case in a frontend language.



Generic: Single Interface for Formal Verification

- **Goal:** Single interface to encode a formal test case in a frontend language.



Generic: Single Interface for Formal Verification

- **How:** Express verification intent and constructs in the IR.

```
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

Generic: Single Interface for Formal Verification

- **How:** Express verification intent and constructs in the IR.

- New operations in MLIR:
  - `verif.formal @Sym {attr-dict} {<body>}`
    - declares a **formal test block** that can contain verification operations.
    - Each formal test will generate its own btor2 file.

  - `%val = symbolic_value : <type>`
    - declares a free variable.
    - Only valid inside of a `verif.formal` body.

Goal: Unified Formal Verification System

Goal: Design a verification system that is:

1. **Generic:** Single interface that works for all high-level hardware languages.

2. **Scalable:** Efficiently verifies large modular designs.

3. **Expressive:** Allows for designs to be accurately specified.

Scalable: Maintain Modularity during Verification

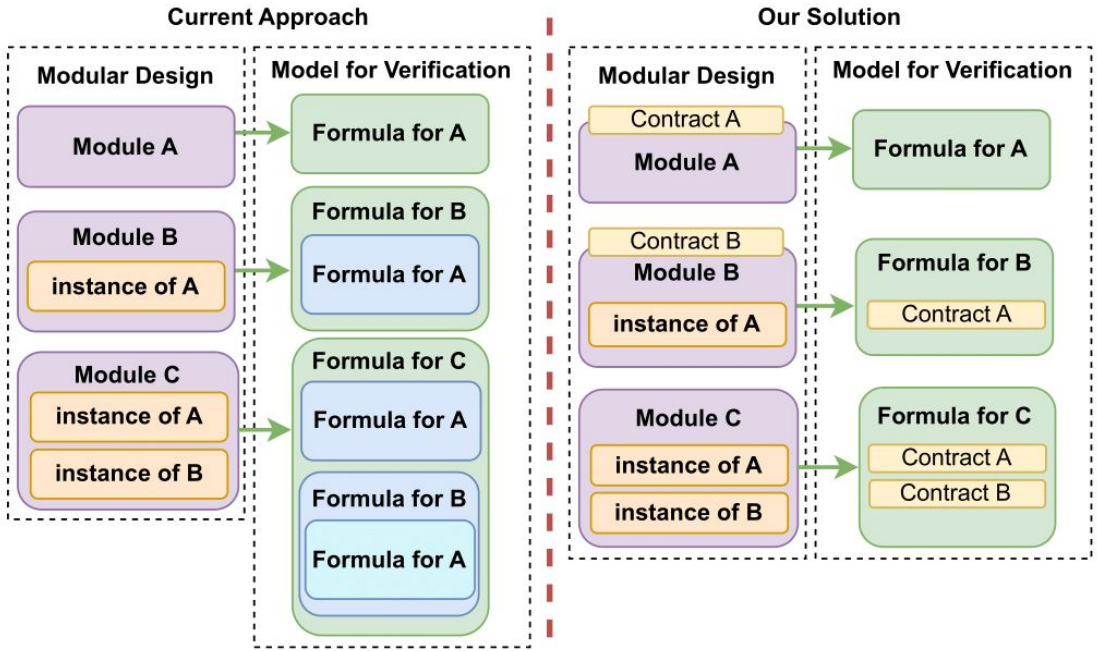
- **Goal**: Efficiently verify large modular designs.

- **Problem**: Current solutions ignore modularity during verification.

- **Solution:** Hardware Contracts
  - Express specifications at the Module boundaries through **pre- & post-conditions**
  - Use modular specification to abstract modules during verification
  - Only verify modules once – rather than *once per instance*

Scalable: Maintain Modularity during Verification

- **Idea:** Preserve modularity throughout verification process.



Scalable: Maintain Modularity during Verification

- **How:** Annotate Modules with contracts. Once Verified, use contracts to abstract module instances.

```
class A extends Module {
     val in = IO(Input(UInt(32.W)))
     val out = IO(Output(UInt(32.W)))

     contract {
          requires in >= 0.U              // precondition
          ensures out === in + 42.U       // postcondition
     }
     // ... Module body ...
}
```

Chisel

Scalable: Maintain Modularity during Verification

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

Chisel


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

MLIR

Scalable: Maintain Modularity during Verification

- New operations in MLIR:
  - `%out = verif.contract (<inputs>) {<body>}`
    - declares a **contract** that can contain pre- & postconditions.
    - Functions as a Hoare Triple → inputs will be abstracted during verification.
    - Output is the result that in postconditions

Scalable: Maintain Modularity during Verification

- New operations in MLIR:
  - `%out = verif.contract (<inputs>) {<body>}`
    - declares a **contract** that can contain pre- & postconditions.
    - Functions as a Hoare Triple → inputs will be abstracted during verification.
    - Output is the result that in postconditions
  - `verif.requires %precondition : <type>`
    - declares a precondition
    - Only valid inside of a `verif.contract` body

Scalable: Maintain Modularity during Verification

- New operations in MLIR:
  - `%out = verif.contract (<inputs>) {<body>}`
    - declares a **contract** that can contain pre- & postconditions.
    - Functions as a Hoare Triple → inputs will be abstracted during verification.
    - Output is the result that 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

Scalable: Maintain Modularity during Verification

- Verification Compilation Flow:
  - Lower frontend modules to MLIR modules containing contracts.
  - Lower formal tests to verif formal tests.

Scalable: Maintain Modularity during Verification

- Verification Compilation Flow:
  - Lower frontend modules to MLIR modules containing contracts.
  - Lower formal tests to verif formal tests.

  - 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

Scalable: Maintain Modularity during Verification

- Verification Compilation Flow:
  - Lower frontend modules to MLIR modules containing contracts.
  - Lower formal tests to verif formal tests.

  - 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

Scalable: Maintain Modularity during Verification

- Verification Compilation Flow:

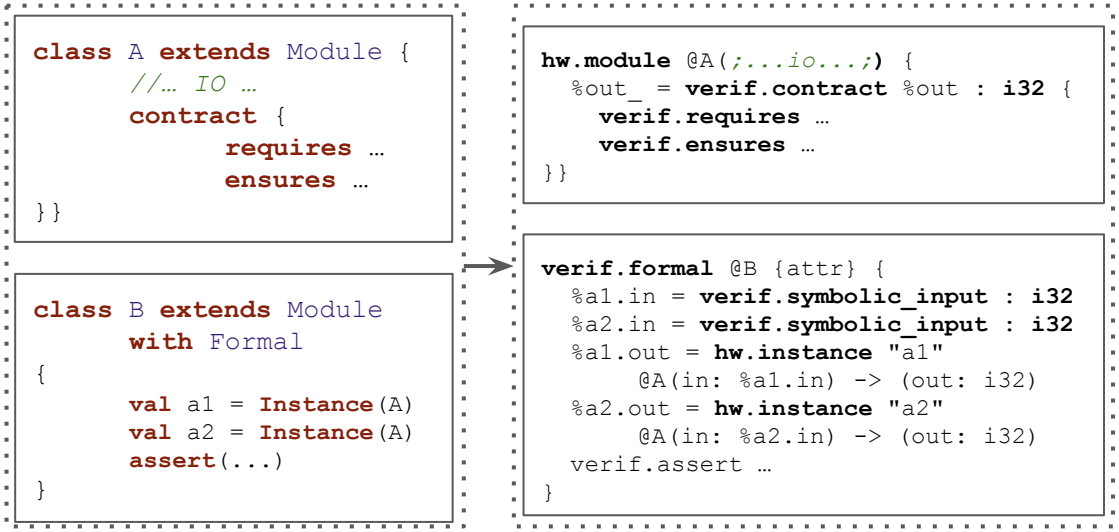
```
class A extends Module {
      //… IO …
      contract {
            requires …
            ensures …
}}
```

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

frontend

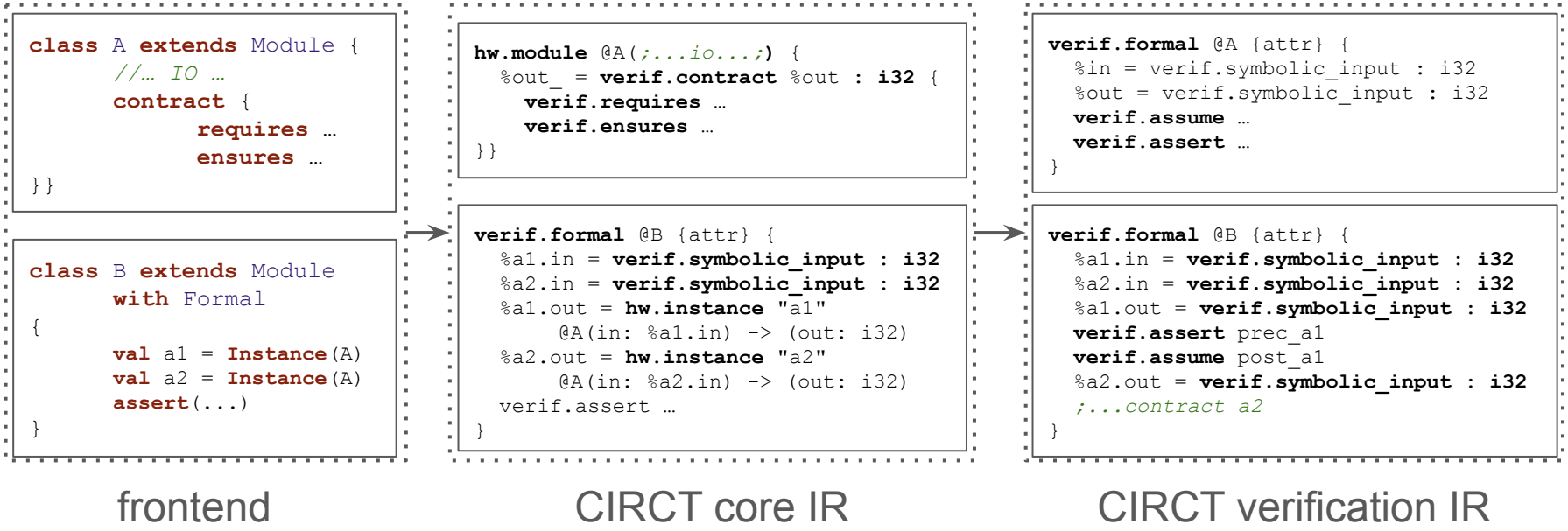
Scalable: Maintain Modularity during Verification

- Verification Compilation Flow:



Scalable: Maintain Modularity during Verification

- Verification Compilation Flow:



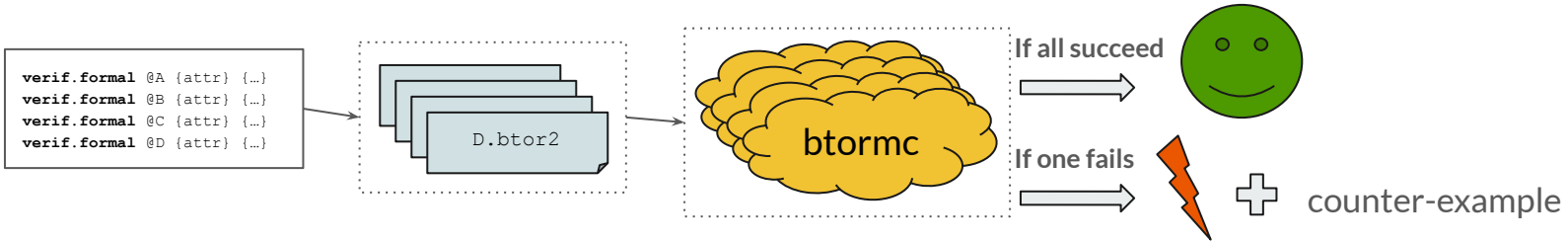
Scalable: Maintain Modularity during Verification

Why?

Scalable: Maintain Modularity during Verification

Why?

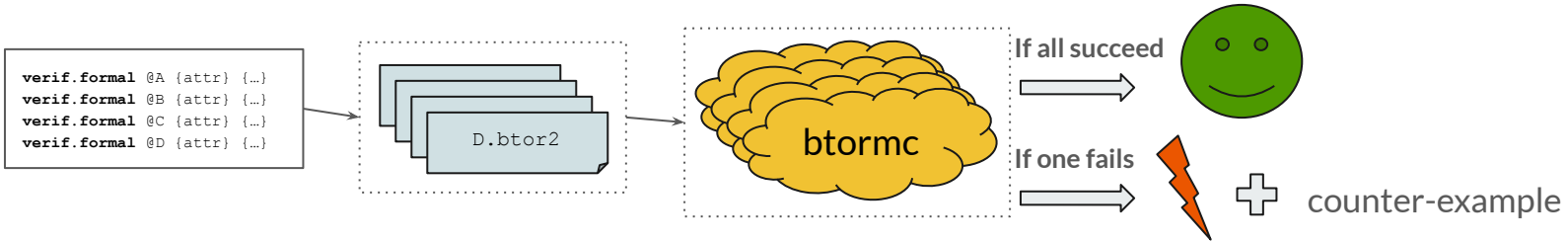
- Enables Solver Parallelism



Scalable: Maintain Modularity during Verification

Why?

- Enables Solver Parallelism



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

Goal: Unified Formal Verification System

Goal: Design a verification system that is:

1. **Generic:** Single interface that works for all high-level hardware languages.

2. **Scalable:** Efficiently verifies large modular designs.

3. **Expressive:** Allows for designs to be accurately specified.

Expressive: Enable Accurate Specifications

- **Goal**: Accurately specify designs.

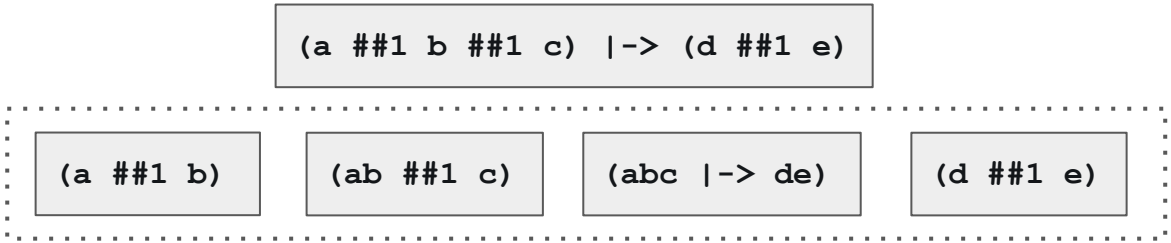
- **Problem**: Sequential hardware requires temporal specifications.
  - Modal Logic is complex and hard to support.
  - Only supported by expensive commercial verification tools.

- **Solution:** Incrementally lower temporal specifications into synthesizable hardware before verification.
  - Enables the use of LTL-like formulae in specifications
  - Lowers complex expressions to standard, widely supported constructs.

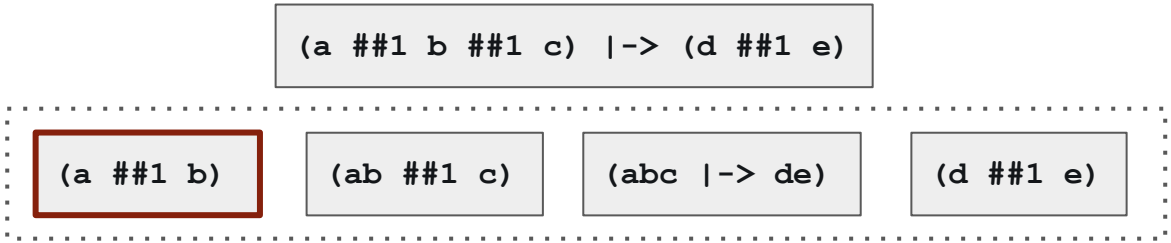
Expressive: Enable Accurate Specifications

```
(a ##1 b ##1 c) |-> (d ##1 e)
```

Expressive: Enable Accurate Specifications

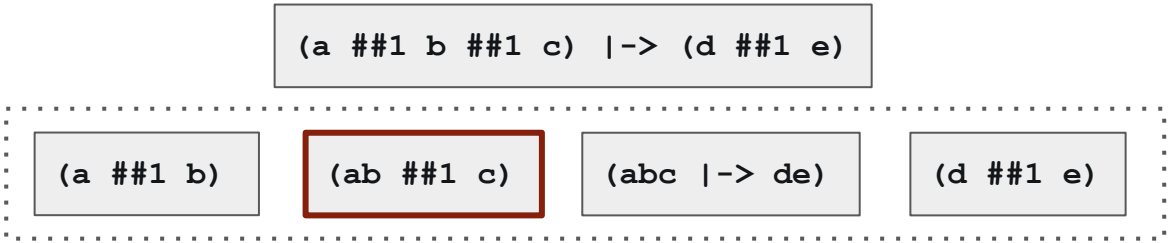


Expressive: Enable Accurate Specifications




```
%ab = ltl.seq {
  %a_1 = ltl.delay %a, 1 : i1
  %res = comb.and bin %a_1, %b : i1
  ltl.yield %res
} : !ltl.sequence<1>
```

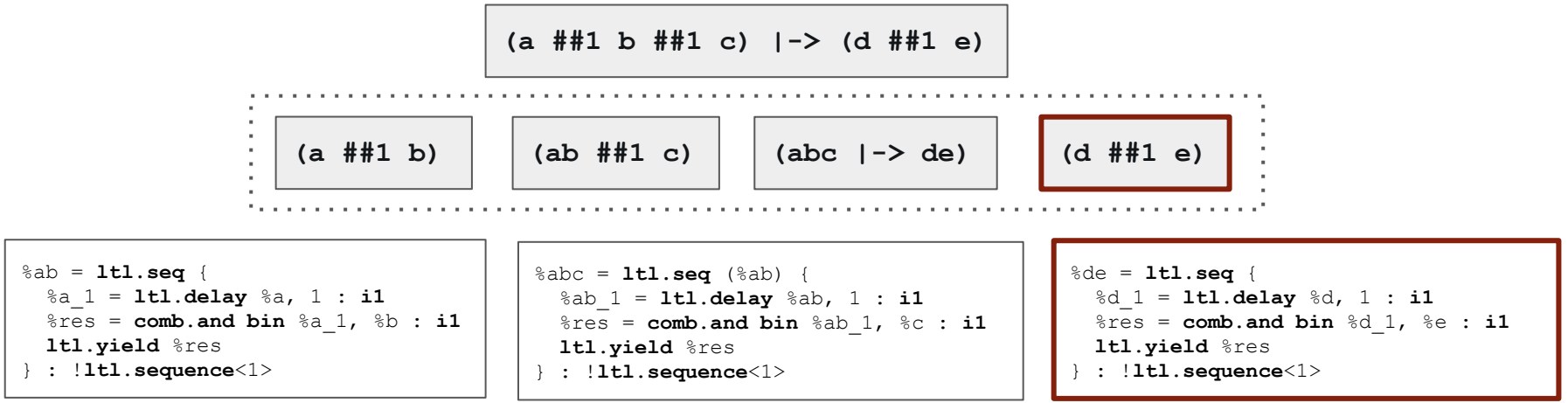
Expressive: Enable Accurate Specifications



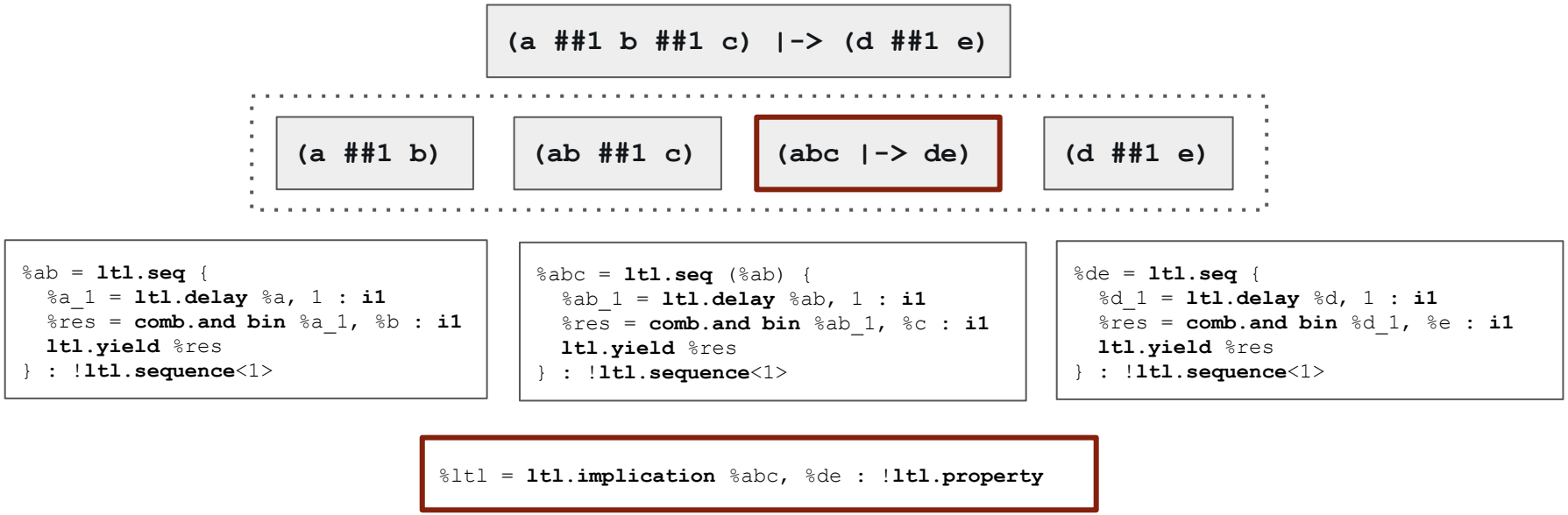

```
%ab = ltl.seq {
  %a_1 = ltl.delay %a, 1 : i1
  %res = comb.and bin %a_1, %b : i1
  ltl.yield %res
} : !ltl.sequence<1>
```

```
%abc = ltl.seq (%ab) {
  %ab_1 = ltl.delay %ab, 1 : i1
  %res = comb.and bin %ab_1, %c : i1
  ltl.yield %res
} : !ltl.sequence<1>
```

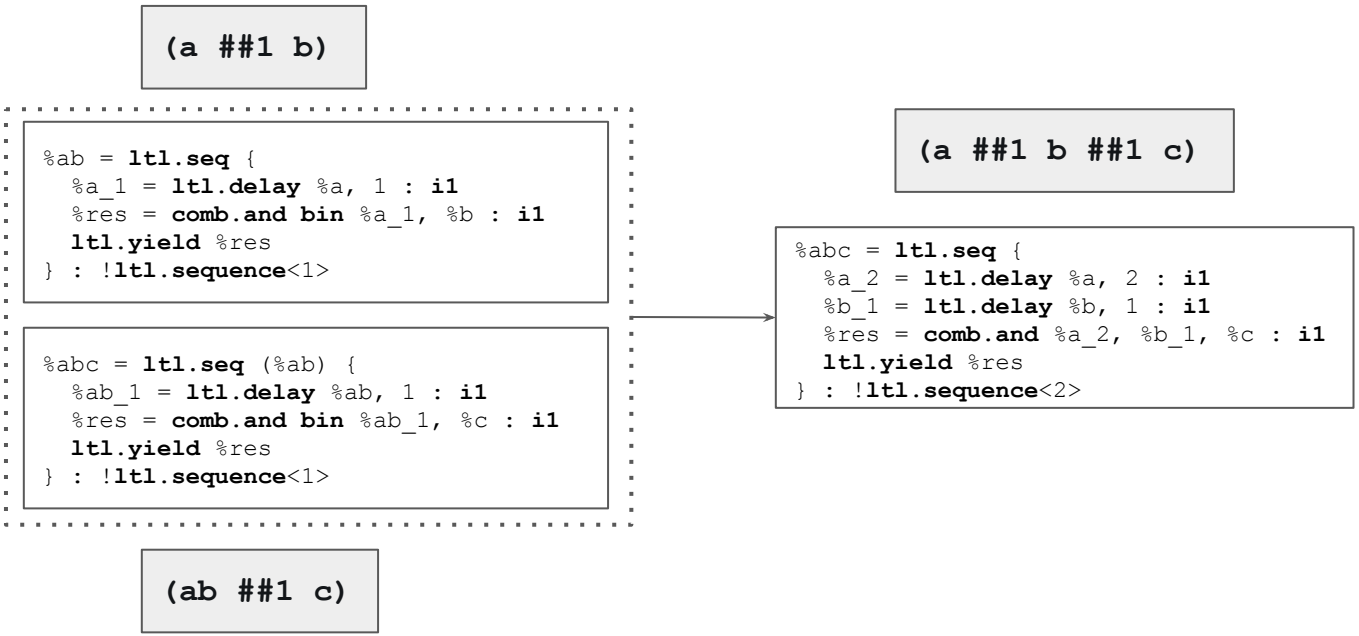
Expressive: Enable Accurate Specifications



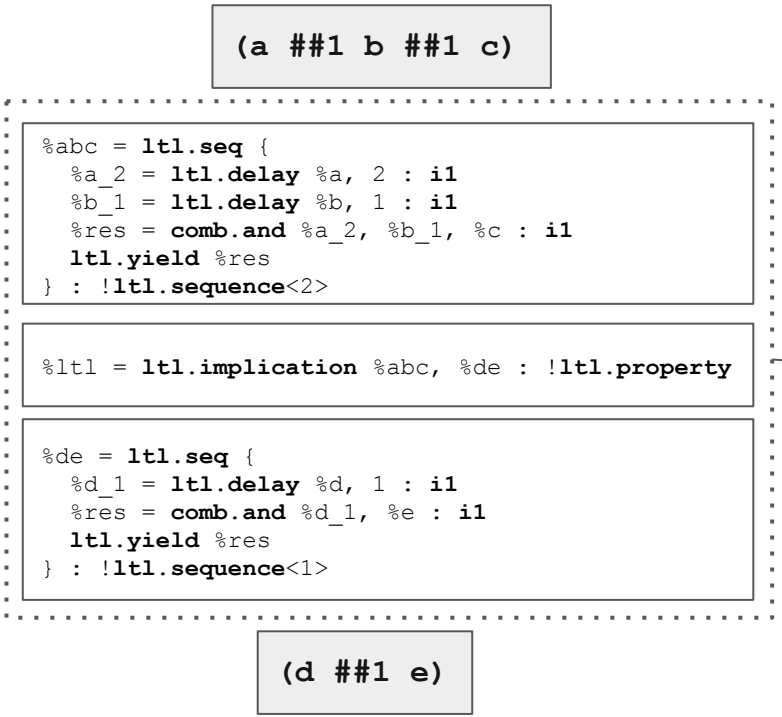
Expressive: Enable Accurate Specifications

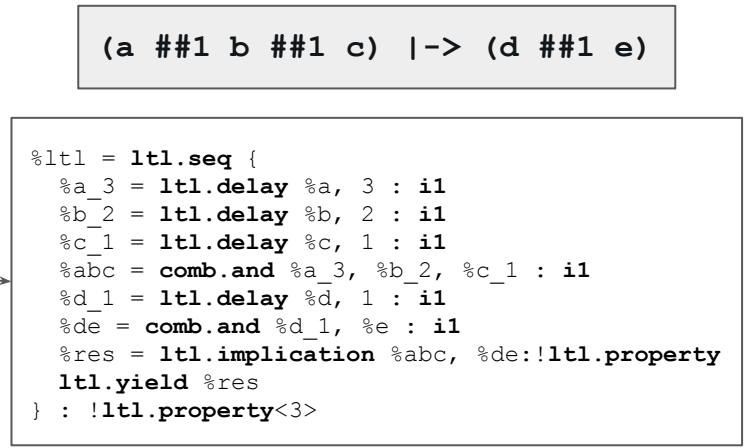


Expressive: Enable Accurate Specifications



Expressive: Enable Accurate Specifications





Expressive: Enable Accurate Specifications

With this design we achieve:

- **Expressiveness**: we can encode a wide variety of LTL expressions.

- **Compositionality**: composed LTL formulae lower correctly without additional effort.

- **Synthesizability**: formulae are easier to convert than the traditional method of building a monitor automata.

Goal: Unified Formal Verification System

Goal: Design a verification system that is:

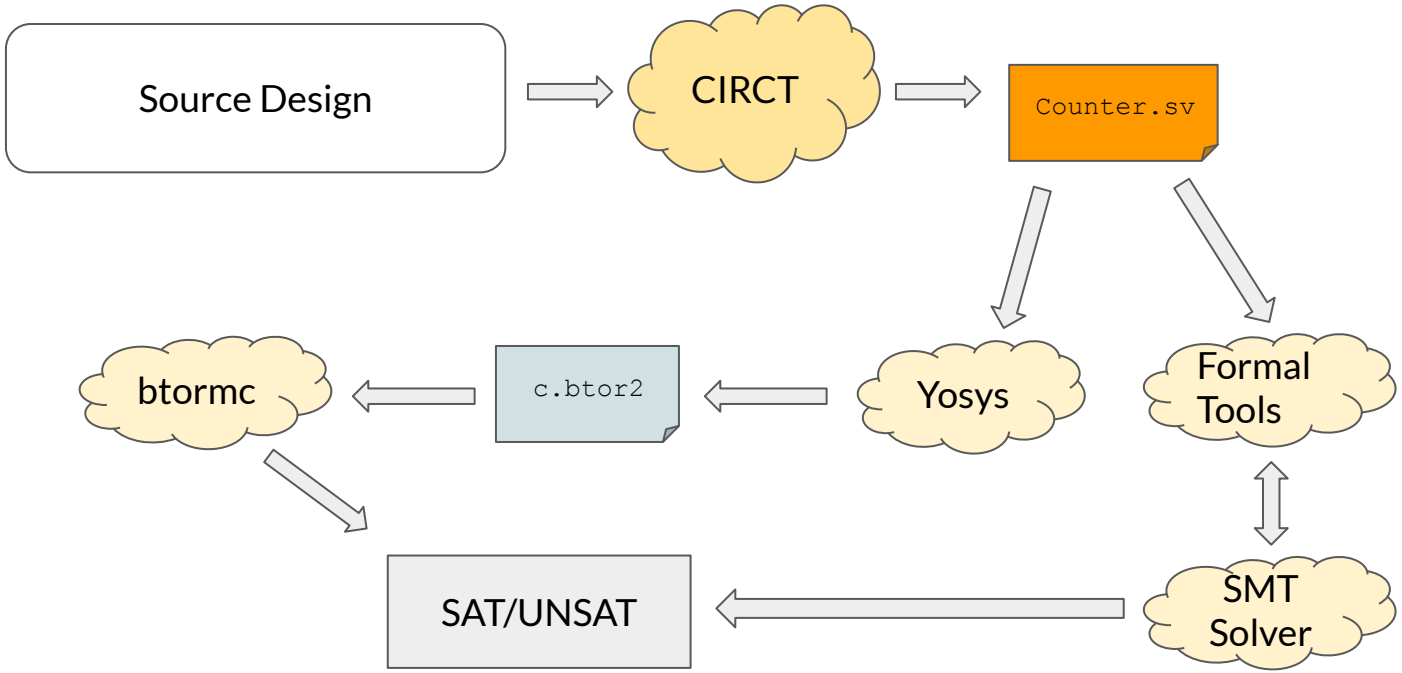
1. **Generic:** Single interface that works for all high-level hardware languages.

2. **Scalable:** Efficiently verifies large modular designs.

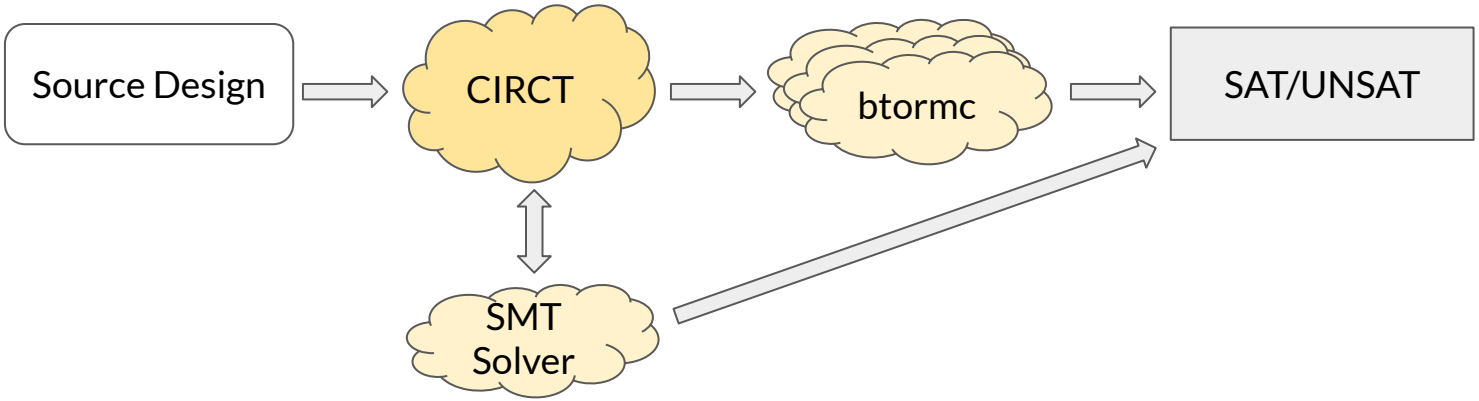
3. **Expressive:** Allows for designs to be accurately specified.

Conclusion

Conclusion: Current Approach



Conclusion: Our Solution



Conclusion

Designed a scalable unified verification system for hardware using:

1. **Unified Formal Test Interface**
   a. Allows any frontend to gain access to Formal Verification *for free*.

2. **Hardware Contracts – Maintain Modularity during Verification**
   a. Enable Solver Parallelism
   b. Simplify individual verification tasks

3. **General LTL support**
   a. IR that encodes LTL expressions in a composable manner
   b. Incrementally lower expression to synthesizable hardware through simple passes

Repositories

- **Btor2-opt**: Experimentation and Benchmarking
  - https://github.com/dobios/btor2-opt
- **CIRCT**: Final MLIR implementation of language constructs
  - https://github.com/llvm/circt
- **Patronus**: Implementation of solver optimizations
  - https://github.com/cucapra/patronus

Questions?