



Improving Formal Verification Support in CIRCT

Amelia Dobis – April 2024 to August 2024



Goal of my work

⇒ Improve **efficiency** and **capabilities** of **Formal Verification** inside of CIRCT and Chisel.

1. Improve existing verification infrastructure.
2. Introduce a new dedicated end-to-end verification flow in Chisel and CIRCT.



Goal of my work

⇒ Improve **efficiency** and **capabilities** of **Formal Verification** inside of CIRCT and Chisel.

1. Improve existing verification infrastructure.
 - a. Update LTL dialect.
 - b. Improve SVA property emission.
2. Introduce a new dedicated end-to-end verification flow in Chisel and CIRCT.



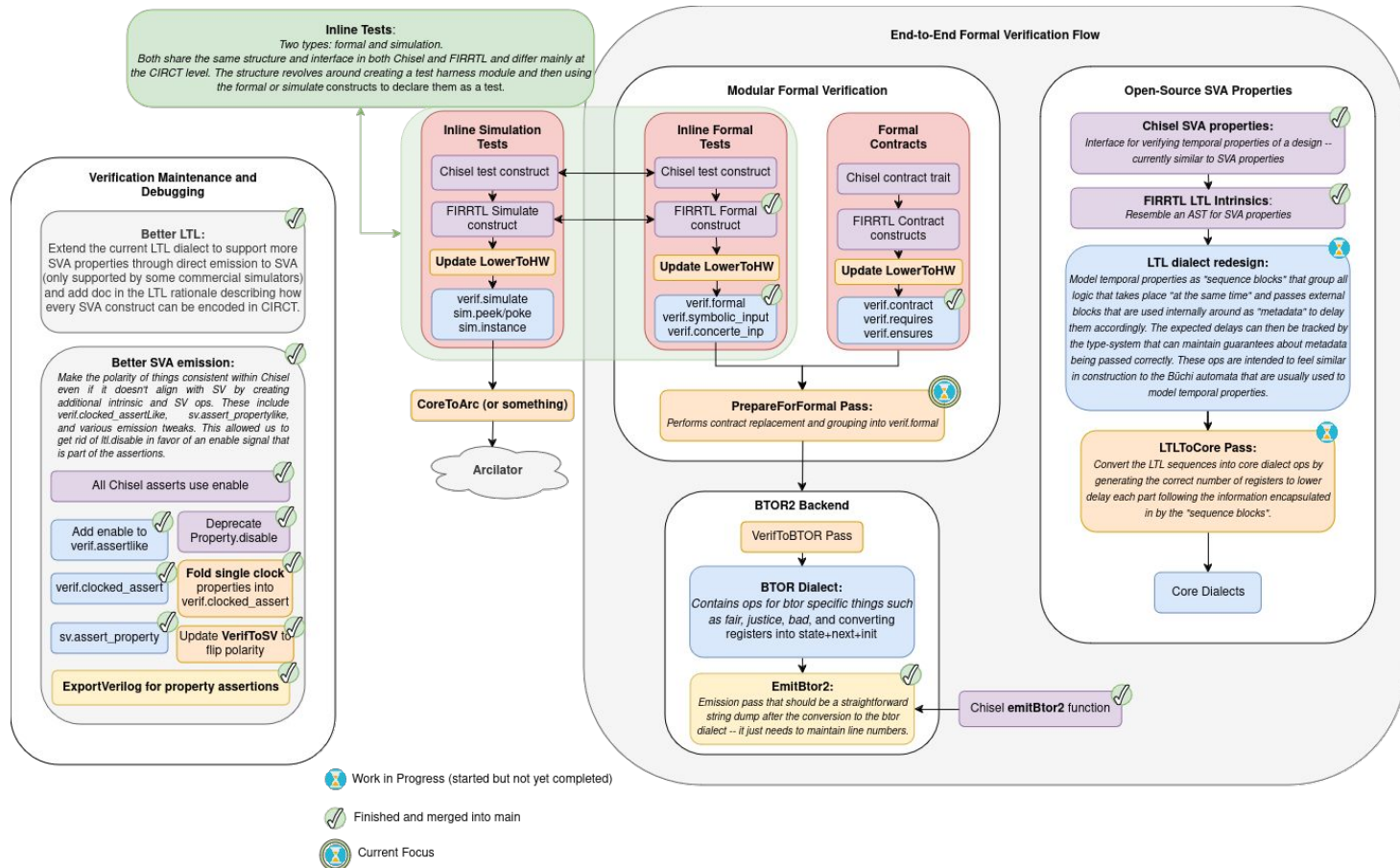
Goal of my work

⇒ Improve **efficiency** and **capabilities** of **Formal Verification** inside of CIRCT and Chisel.

1. Improve existing verification infrastructure.
 - a. Update LTL dialect.
 - b. Improve SVA property emission.

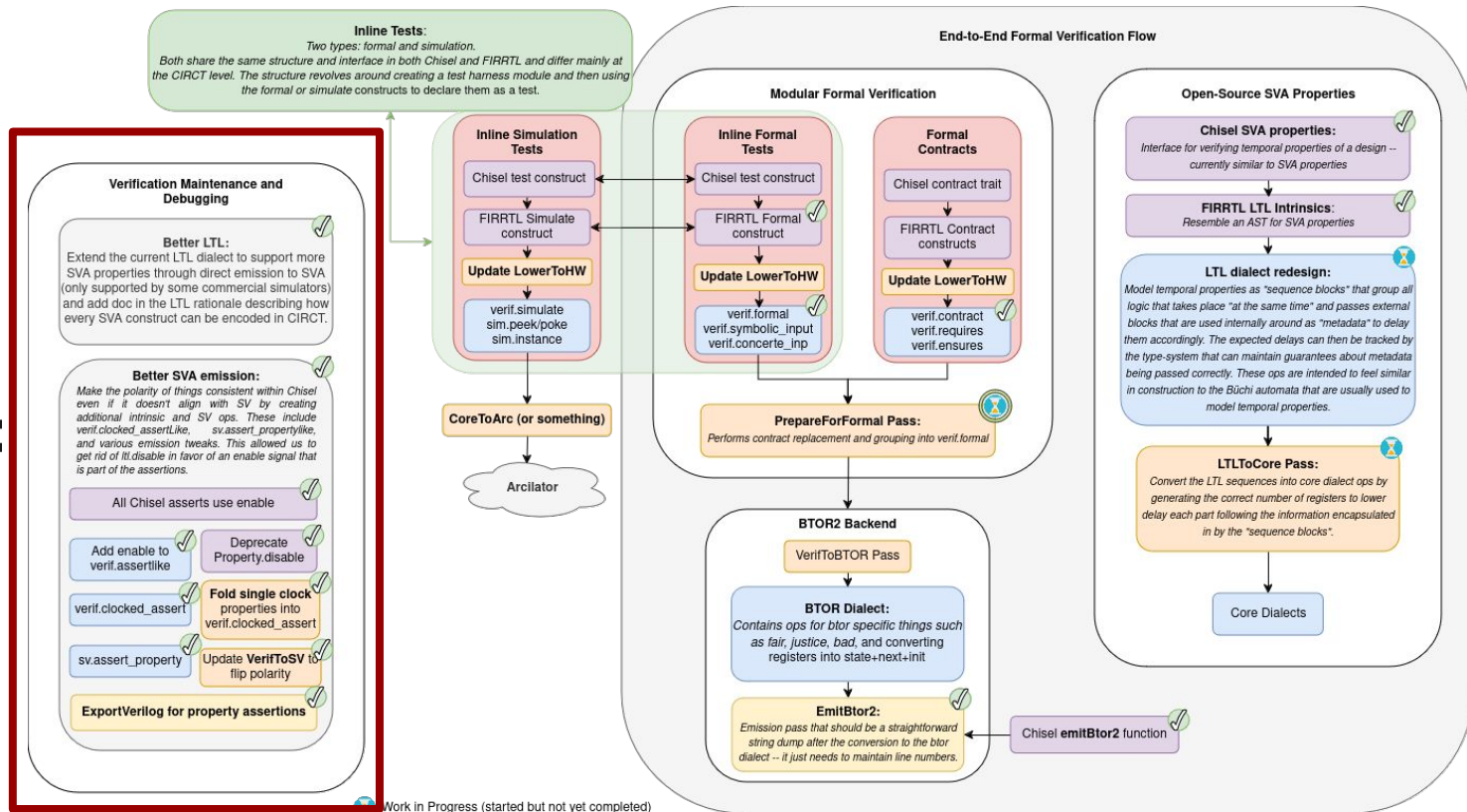
2. Introduce a new dedicated end-to-end verification flow in Chisel and CIRCT.
 - a. Integrate and expand **btor2 back-end**.
 - b. Explore the ideas related to **co-locating designs and tests** for formal tests.
 - c. **Modularize formal verification** through the introduction of formal contracts.
 - d. Redesign LTL dialect to better support property assertion synthesis.

Overview:



Improving Existing Verification Infrastructure

Overview:





Verification Maintenance and Debugging

1. Updating LTL to improve expressiveness
2. Making SVA property emission more stable
3. Debugging



Updating LTL

- Write out an [SVA summary](#) which describes how each part of SVA can be mapped to CIRCT. This document was then [merged into the LTL rationale](#).
- [Introduce 3 new ops to the LTL dialect](#).
- [Porting the LTL intrinsics in Chisel](#) from intrinsic modules to intrinsic expressions, making the generated firrtl for property much smaller.
- [Providing a simpler interface](#) for property assertions.
- [Providing new interfaces](#) to access all of the ltl features that CIRCT supports.



Improving SVA Property Emission

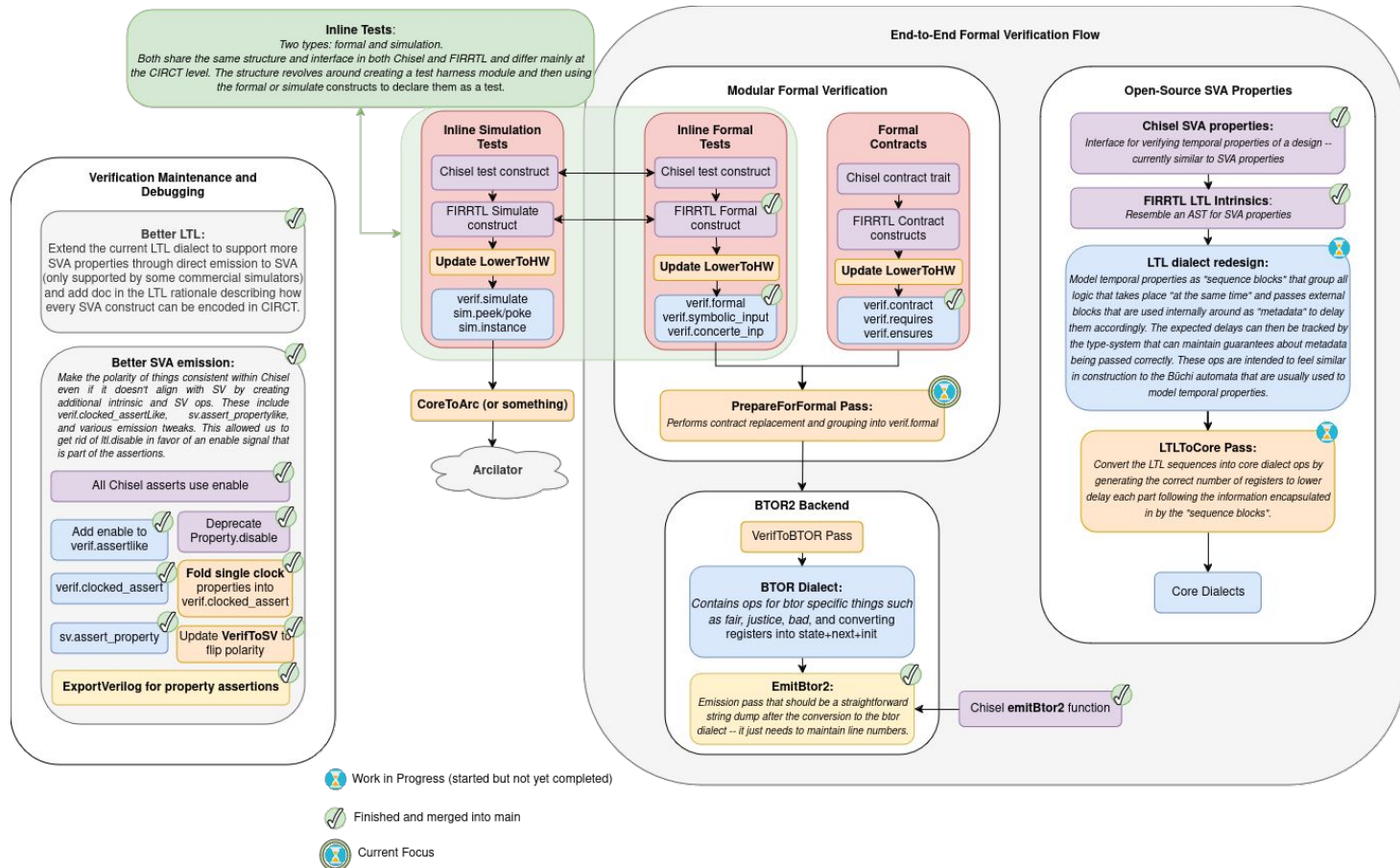
- [Flipped the polarity of all disable signals](#), such that everything in Chisel and CIRCT uses enables.
- [Removed the `ltl.disable op`](#) from the IR thus simplifying the emission of property assertions.
- [Added an enable signal to the `verif.assert`](#) like ops, to replace the removed `ltl.disable op`.
- [Deprecated disabling individual properties](#) in Chisel, as this couldn't be mapped to SV.
- [Introduced clocked assertions](#).
- [Fixed bugs related](#) to properties being used in disable signals (which is disallowed in SV).
- [Explored various encodings](#) for disable signals.
- [Explored using the LTL type system](#) to simplify the property assertion verifier logic.
- [Introduced the `sv.assert_property`](#)
- [Updated ExportVerilog](#) to support the new, simpler and more concise, emission of property assertions.



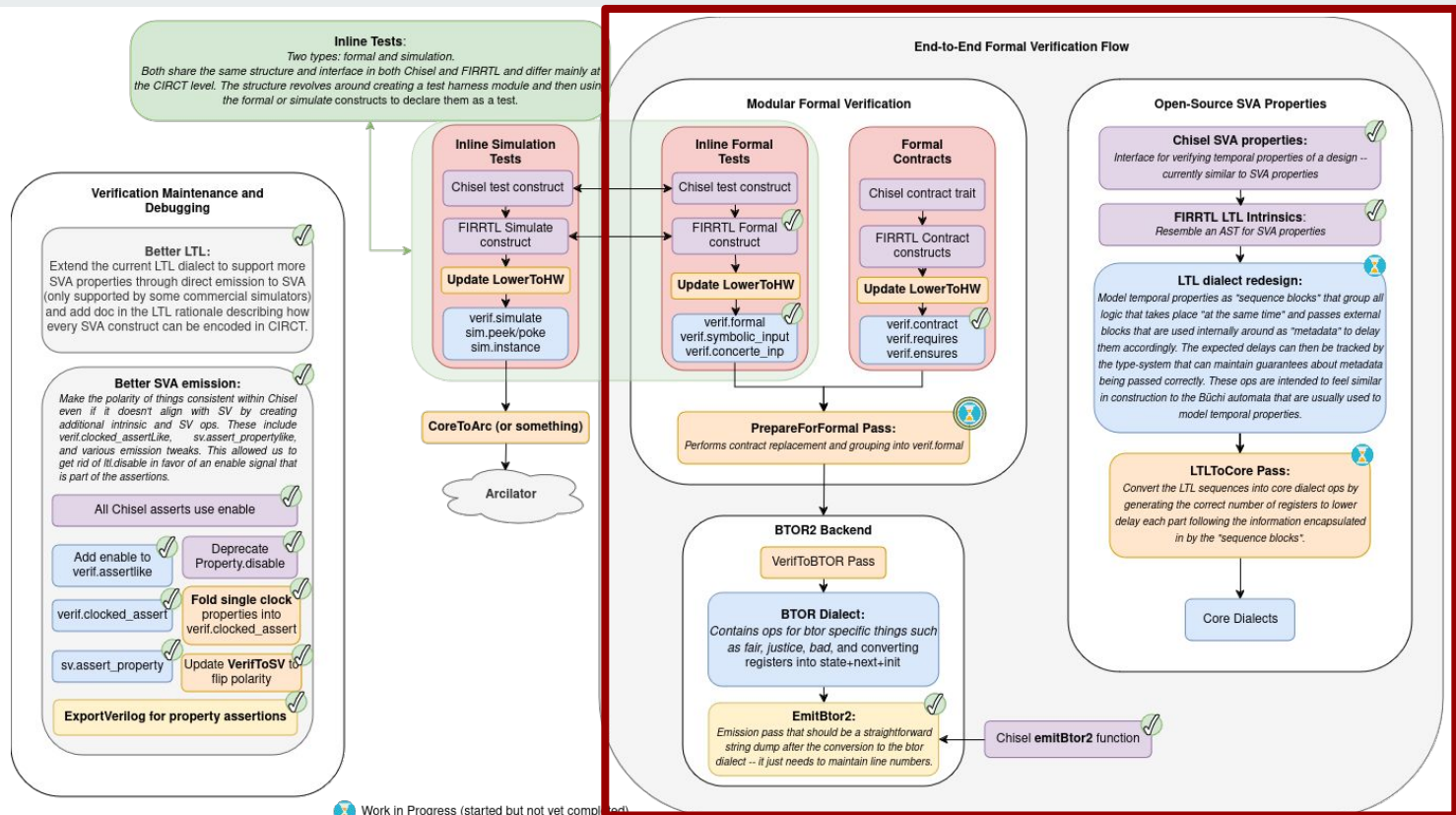
Verification Debugging

- ExpandWhens not supporting AssertProperty. This meant that property assertions declared under a when would simply be ignored (fixed in [PR#7021](#) and later improved in [PR#7150](#)).
- SVSim not firing assertions correctly when using Verilator (fixed in [PR#4087](#)).
- Firrtol lowering assertions incorrectly (fixed in [PR#7157](#)).

Overview:



Overview:



Work in Progress (started but not yet completed)

Finished and merged into main

Current Focus

Re-imagining Formal Verification in Chisel and CIRCT



Background: Formal Verification

What is Formal Verification?

- Idea: Instead of testing your design through simulation, prove its correctness statically using formal methods.
- How?
 - Annotate your design with a specification in the form of **assertions** and **assumptions**.
 - Use design + specification to generate **Verification Conditions (VC)**.
 - Check the satisfiability of the VCs using SMT solvers → if unsatisfiable then your design matches the spec.
- Why?
 - Provides stronger guarantees than simulation → checks are exhaustive.
 - Can quickly find edge case bugs in the design implementation.



Background: Verification Conditions

Idea: Convert design into conjunction of constraints (signal definitions) and negated assertion conditions.

```
val a = IO(Input(32.W))  
val b = a + 1.U  
assert (b > a)
```

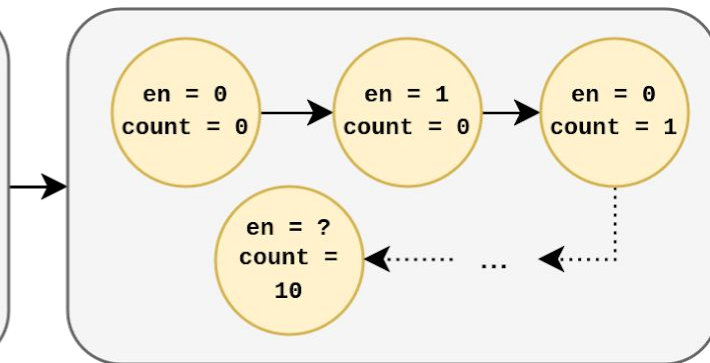


```
(and  
  (eq b (add a 1)) // define b  
  (not (gt b a)) // can assertion be violated?  
)
```


Background: Verification Conditions

Handling State: Create “state-transition systems” from registers + memories, requires **Bounded Model Checking** to be verified.

```
class MyCounter extends Module {  
  val en = IO(Input(Bool()))  
  val count = RegInit(0.U(32.W))  
  when(en && count == 22.U) { count := 0.U }  
  when(en && count != 22.U) { count := count + 1.U }  
  assert(count != 10.U)  
}
```



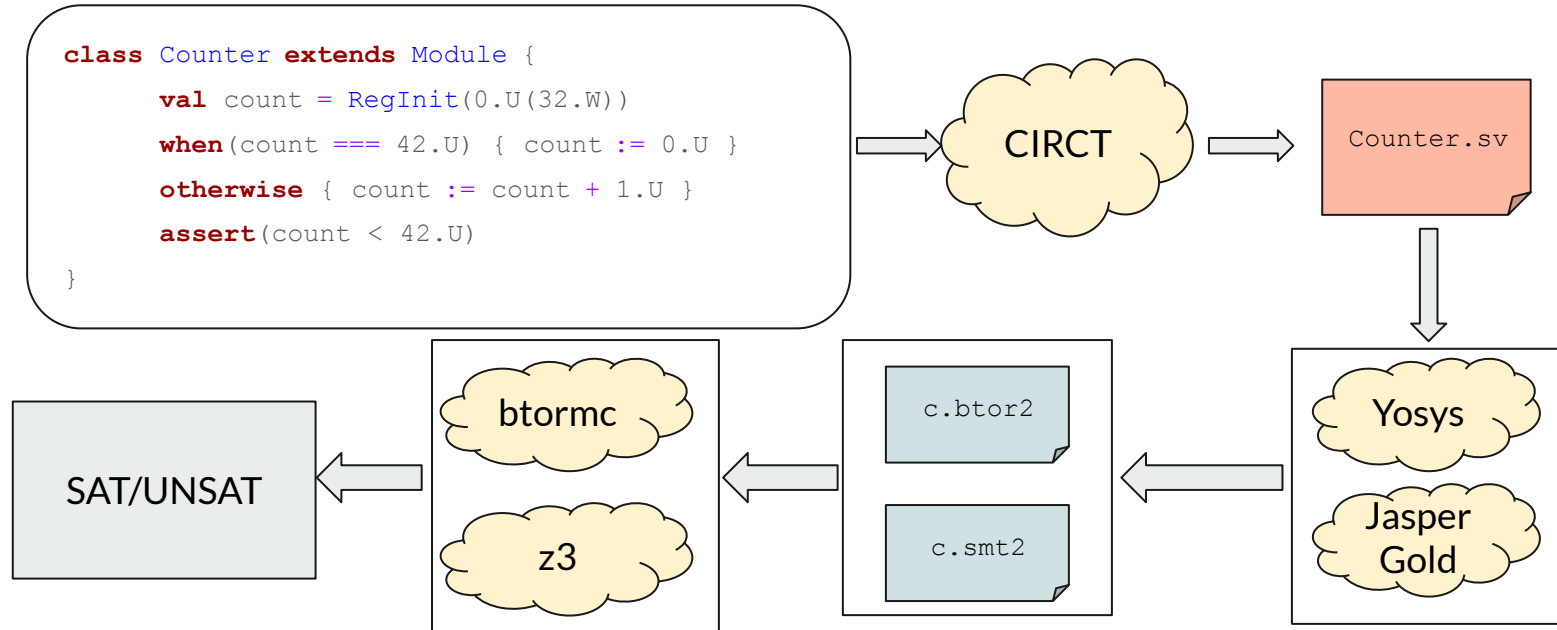


Background: BTOR2 and btormc

- BTOR2:
 - SMTLib-like format that allows for the explicit encoding of state-transition systems.
 - Supports **bitvector** and **array** theories.
 - No need to manually unroll states, e.g.
- BTORMC:
 - Bounded Model Checker.
 - Supports btor2 format, uses the **boolector** SMT solver.
 - Optimized for solving in bitvector and array theories.

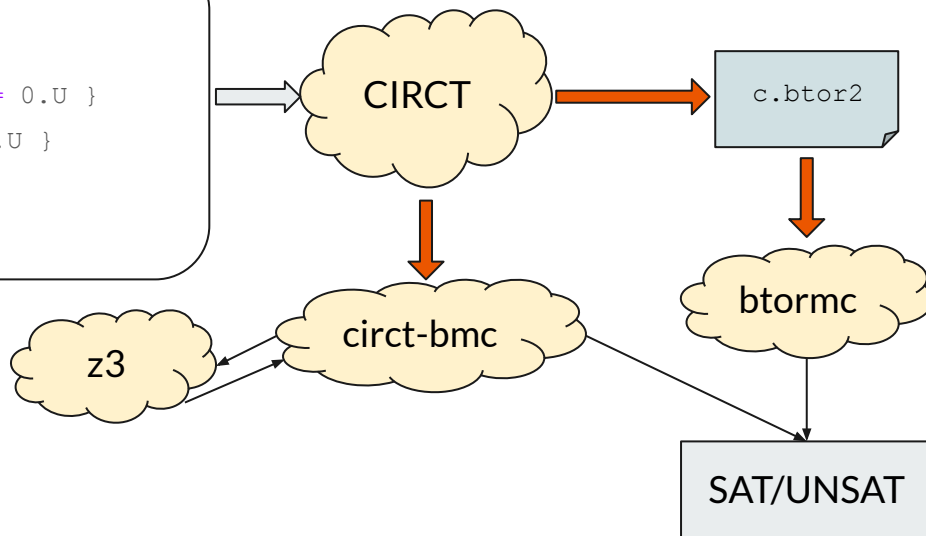
```
1 sort bitvector 32 ; declare a 32-bit type
2 state 1 count      ; declare a 32-bit state
3 one 1              ; declare a 32-bit constant of value 1
4 and 1 2 3
5 next 1 2 4         ; count := count & 1
```

Motivation: Formal Verification

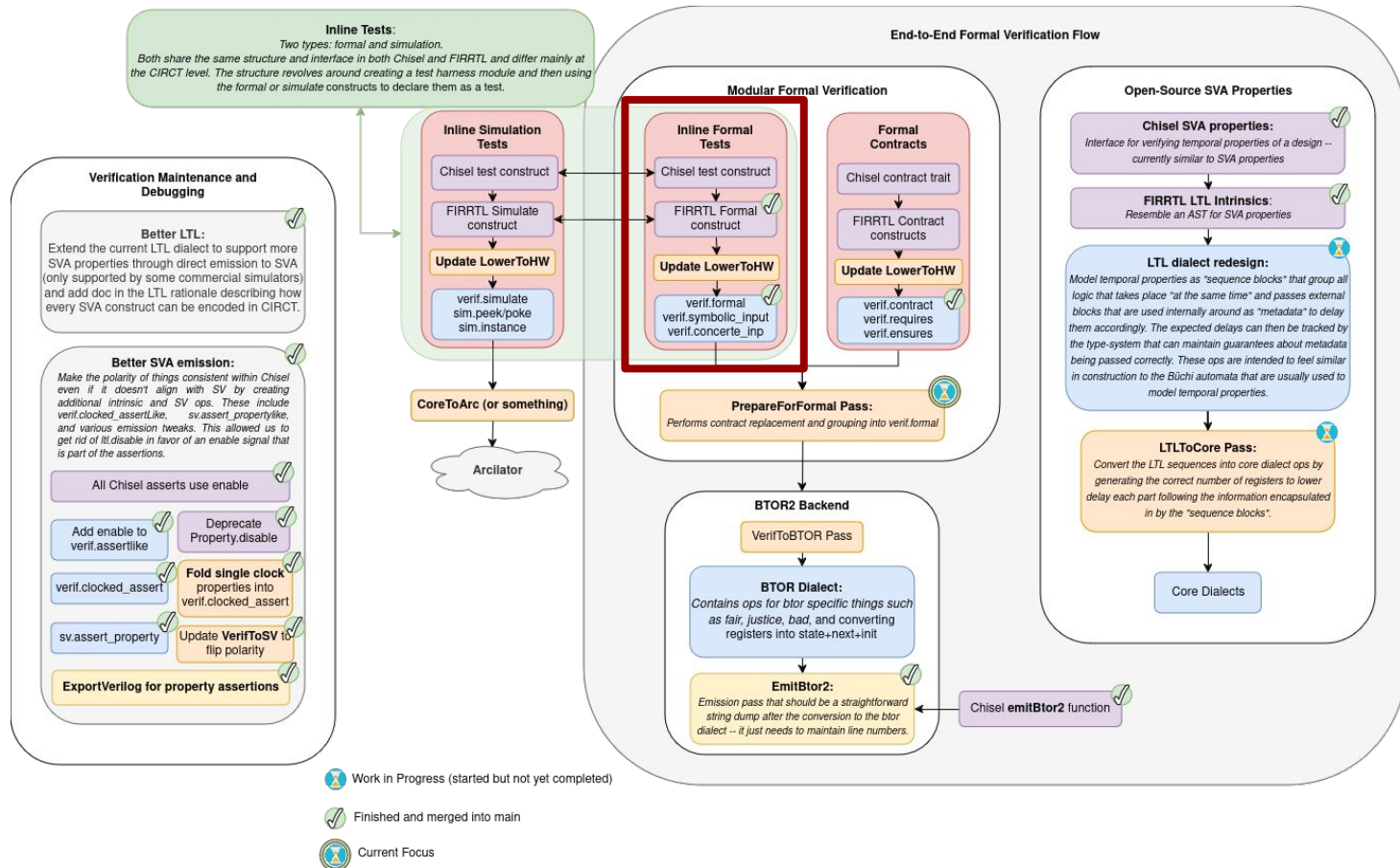


Motivation: Formal Verification

```
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)  
}
```



Overview:





Inline Formal: Unified formal test interface

- Idea: Create a unified interface for all formal tests.
 - Allows for simple user interface.
 - Back-end can be defined through build parameters.
 - Test is co-located with design



Inline Formal: Unified formal test interface

```
class Foo extends Module {  
  val in = IO(Input(UInt(32.W))  
  val out = IO(Output(UInt(32.W))  
  
  /* body of the module */  
  
  // Some formal test  
  test formal testFoo(500) {  
    val dut = Instantiate(Foo)  
    AssertProperty(/* some property */)   
  }  
}
```

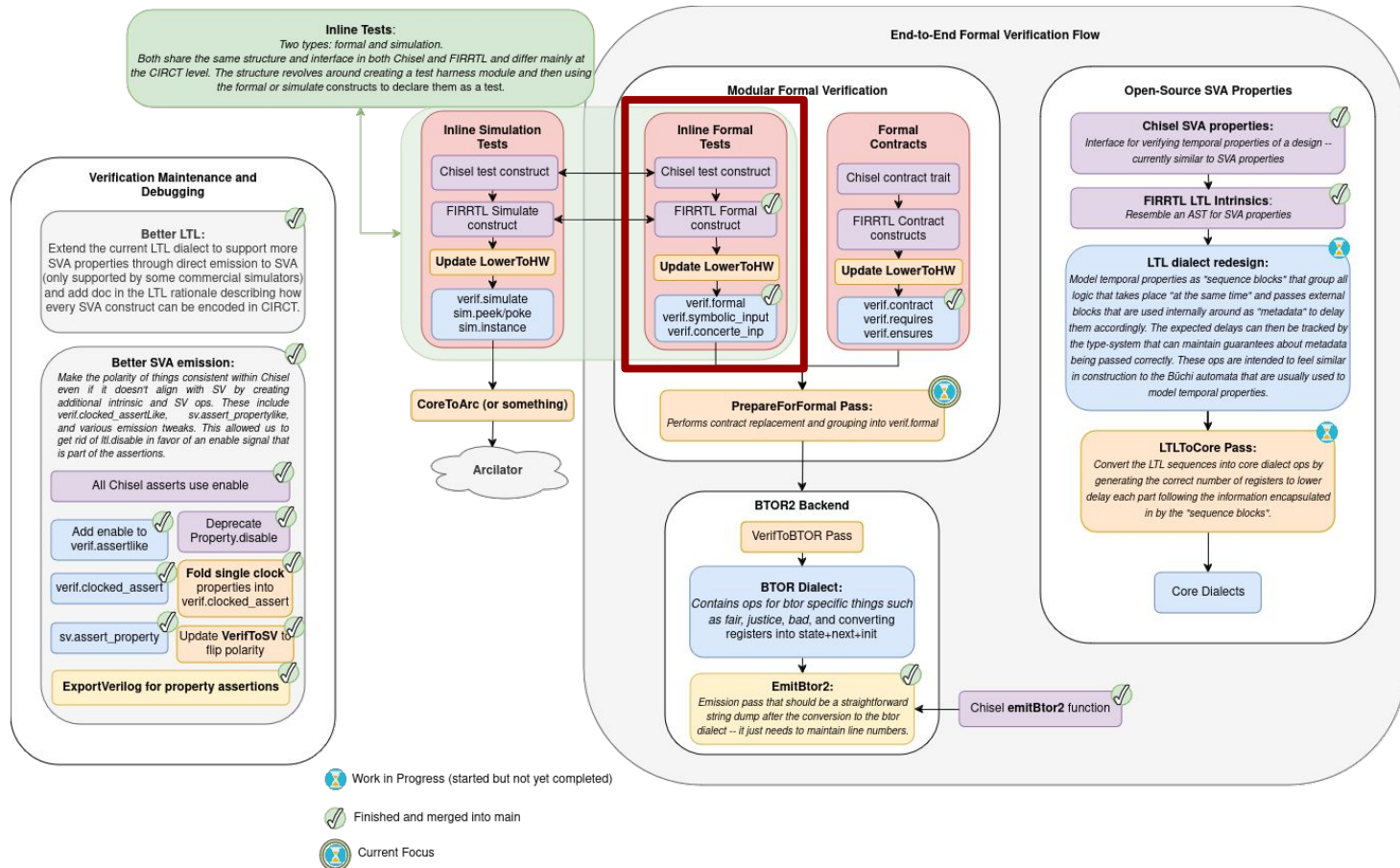


Inline Formal: Unified formal test interface

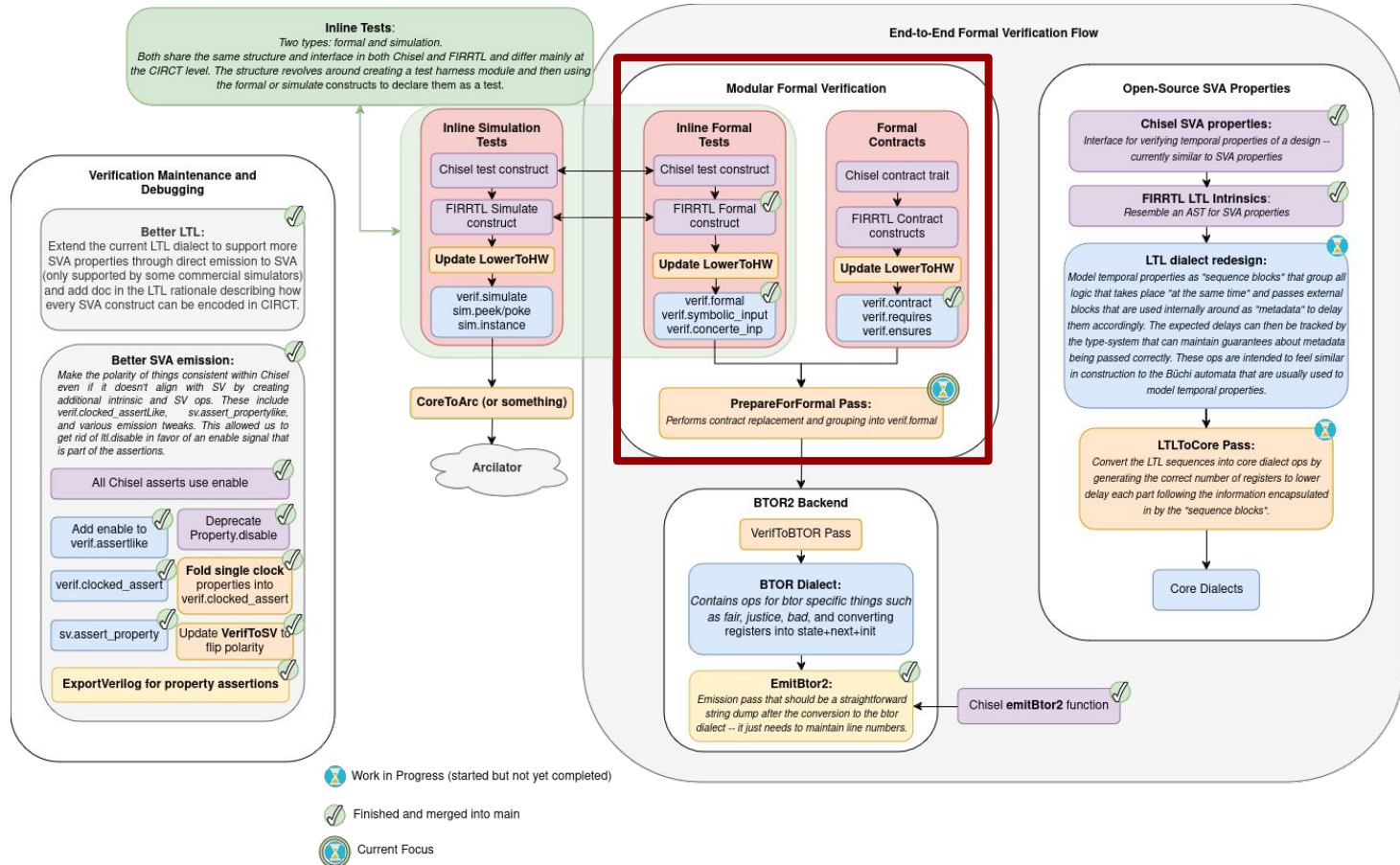
```
class Foo extends Module {  
  val in = IO(Input(UInt(32.W))  
  val out = IO(Output(UInt(32.W))  
  
  /* body of the module */  
  
  // Some formal test  
  test formal testFoo(500) {  
    val dut = Instantiate(Foo)  
    AssertProperty(/* some property */)   
  }  
}
```

- Formal tests can target either **btor2** or **circt-bmc**.
- Formal tests are ignored during SV generation for synthesis.
- Formal tests are included in SV generation for testing.
- Verif constructs: [PR #7145](#)
- FIRRTL op: [PR #7374](#)

Overview:



Overview:





Handling Modularity in Designs under Verification

- Problem: Current solutions for generating VCs for module instances:
 - **Inline Module VC** → Requires re-checking the same VCs multiple times → slow verification
 - **Manually define assumptions** to abstract away certain parts
 - Difficult to get right
 - Very manual process
 - Often incomplete abstraction

→ We only want to verify a module exactly once.

(not once per instance)



Handling Modularity in Designs under Verification

- Idea: Allow for user to define “verification checkpoints” that can be used as abstractions to verify module instances.
- Formal Contracts: Define a contract that the module is proven to support.
 - **Pre-conditions**: Specifications over the module’s inputs
 - “What do I expect correct inputs to look like?”
 - **Post-conditions**: Guarantees for the module’s outputs
 - “Given certain inputs, what do my outputs look like?”



Handling Modularity in Designs under Verification

- Module Correctness: Assuming our preconditions can we use our module's definition to prove our post-conditions.
 - `VC: {Pre-conditions} -> ({Body} AND NOT({Post-conditions}))`
- Instance Correctness: Knowing that our Module is correct, our pre-conditions holding implies that our post-conditions hold.
 - `Assert({Pre-conditions}) + Assume({Post-conditions})`

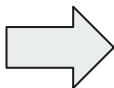


Inline Formal: Unified formal test interface

```
class Foo extends Module with Contract {  
  val in = IO(Input(UInt(32.W))  
  val out = IO(Output(UInt(32.W))  
  
  // define contract  
  contract {  
    require(in > 0.U)  
    require(in < 1000.U)  
    ensure(/* some post-condition */)  
  }  
  
  /* body of the module */  
  
  // Some formal test  
  test formal testFoo(500) {  
    val dut = Instantiate(Foo)  
    AssertProperty(/* some property */)  
  }  
}
```

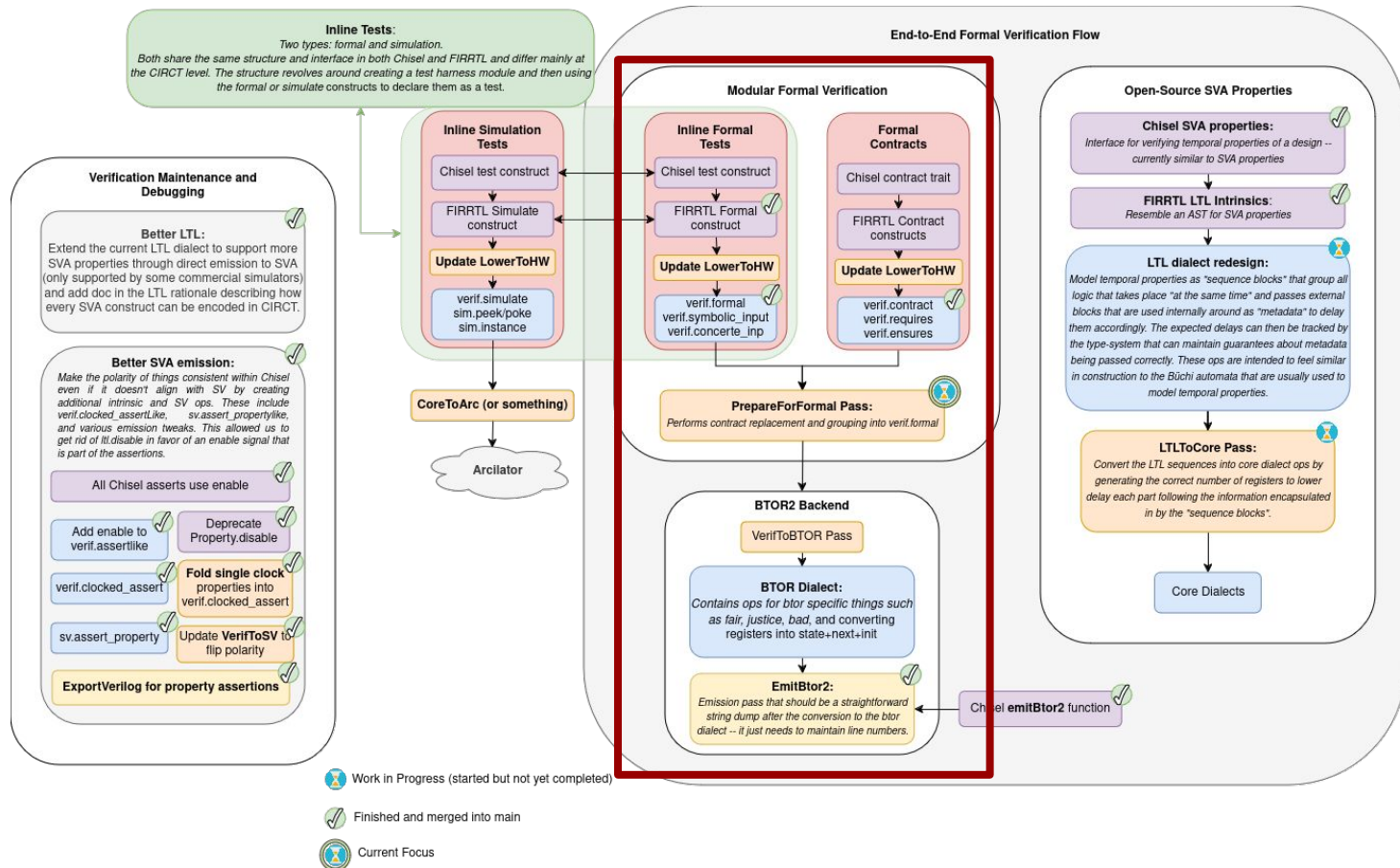
Inline Formal: Unified formal test interface

```
class Foo extends Module with Contract {  
  val in = IO(Input(UInt(32.W))  
  val out = IO(Output(UInt(32.W))  
  
  // define contract  
  contract {  
    require(in > 0.U)  
    require(in < 1000.U)  
    ensure(/* some post-condition */)  
  }  
  
  /* body of the module */  
  
  // Some formal test  
  test formal testFoo(500) {  
    val dut = Instantiate(Foo)  
    AssertProperty(/* some property */)  
  }  
}
```



```
// module test  
test formal Foo(500) {  
  val dut = Instantiate(Foo)  
  Assume(dut.in > 0.U)  
  Assume(dut.in < 1000.U)  
  Assert(/*Body*/ && /*Post-conditions*/)  
}  
  
// module instance test  
test formal testFoo(500) {  
  val dut = Instantiate(Foo)  
  Assert(dut.in > 0.U)  
  Assert(dut.in < 1000.U)  
  Assume(/*Post-conditions*/)  
  AssertProperty(/* some property */)  
}
```

Overview:



End-to-End Verification Flow

Chisel

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

  // define contract
  contract {
    require(in > 0.U)
    require(in < 1000.U)
    ensure(/* some post-condition */)
  }

  /* body of the module */

  // Some formal test
  test formal testFoo(500) {
    val dut = Instantiate(Foo)
    AssertProperty(/* some property */)
  }
}

class Bar extends Module with Contract {
  val in = IO(Input(UInt(32.W))
  val sign = IO(Input(Bool()))
  val out = IO(Output(UInt(32.W))

  contract {
    require(sign |-> in > 0.U)
    require(!sign |-> in < 0.U)
    // Ensures support SVA properties
    ensure((sign |> out > 0.U) &&
           (!sign |> out < 0.U))
  }

  val foo1 = Instantiate(Foo)
  val foo2 = Instantiate(Foo)

  /* body of bar that uses multiple Fools */

  test formal testBar(500) {
    val foo1 = Instantiate(Bar)
    AssertProperty(/*some property*/)
  }
}

object Bar extends App {
  ChiselStage.emitBtor2(new Bar)
}
```

End-to-End Verification Flow

Chisel

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

  // define contract
  contract {
    require(in > 0.U)
    require(in < 1000.U)
    ensure(/* some post-condition */)
  }

  /* body of the module */

  // Some formal test
  test formal testFoo(500) {
    val dut = Instantiate(Foo)
    AssertProperty(/* some property */)
  }
}

class Bar extends Module with Contract {
  val in = IO(Input(UInt(32.W))
  val sign = IO(Input(Bool()))
  val out = IO(Output(UInt(32.W))

  contract {
    require(sign |-> in > 0.U)
    require(!sign |-> in < 0.U)
    // Ensures support SVA properties
    ensure((sign |=> out > 0.U) &&
      (!sign |=> out < 0.U))
  }

  val foo1 = Instantiate(Foo)
  val foo2 = Instantiate(Foo)

  /* body of bar that uses multiple Fools */

  test formal testBar(500) {
    val foo1 = Instantiate(Bar)
    AssertProperty(/*some property*/)
  }
}

object Bar extends App {
  ChiselStage.emitBtor2(new Bar)
}
```

sbt run



FIRRTL

```
circuit Bar:
  public module Foo:
    input in : UInt<32>
    output out : UInt<32>
    contract:
      node prec0 = gt(in, 0)
      require prec0
      node prec1 = lt(in, 1000)
      require prec1
      node post = ;;some post-condition;;
      ensure post
      ;; Body of the module

  ;; Some Formal Test
  public module _Foo:
    input s_in : UInt<32>
    inst dut of Foo
    connect dut.in, s_in
    intrinsic(circt_verif_assert(...))

  formal testFoo of _Foo, bound = 500

  public module Bar:
    input in : UInt<32>
    input sign : Bool
    output out : UInt<32>
    contract:
      node prec0 = intrinsic(circt_ltl_implication(...))
      require prec0
      node prec1 = intrinsic(circt_ltl_implication(...))
      require prec1
      node post = ;;some post-condition;;
      ensure post

    inst foo1 of Foo
    inst foo2 of Foo

    ;; body of bar that uses multiple Fools ;;

  public module _Bar:
    input s_in : UInt<32>
    input s_en : Bool
    inst dut of Bar
    connect dut.in, s_in
    connect dut.en, s_en
    intrinsic(circt_verif_assert(...))

  formal testBar of _Bar, bound = 500
```

End-to-End Verification Flow

Chisel

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

  // define contract
  contract {
    require(in > 0.U)
    require(in < 1000.U)
    ensure(/* some post-condition */)
  }

  /* body of the module */

  // Some formal test
  test formal testFoo(500) {
    val dut = Instantiate(Foo)
    AssertProperty(/* some property */)
  }
}

class Bar extends Module with Contract {
  val in = IO(Input(UInt(32.W)))
  val sign = IO(Input(Bool()))
  val out = IO(Output(UInt(32.W)))

  contract {
    require(sign |> in > 0.U)
    require(!sign |> in < 0.U)
    // Ensures support SVA properties
    ensure((sign |> out > 0.U) &&
      (!sign |> out < 0.U))
  }

  val foo1 = Instantiate(Foo)
  val foo2 = Instantiate(Foo)

  /* body of bar that uses multiple Fools */

  test formal testBar(500) {
    val foo1 = Instantiate(Bar)
    AssertProperty(/*some property*/)
  }
}

object Bar extends App {
  ChiselStage.emitBtor2(new Bar)
}
```

sbt run



FIRRTL

```
circuit Bar:
  public module Foo:
    input in : UInt<32>
    output out : UInt<32>
    contract:
      node prec0 = gt(in, 0)
      require prec0
      node precl = lt(in, 1000)
      require precl
      node post = ;;some post-condition;;
      ensure post
    ;; Body of the module

  ;; Some Formal Test
  public module _Foo:
    input s_in : UInt<32>
    inst dut of Foo
    connect dut.in, s_in
    intrinsic(circt_verif_assert(...))

  formal testFoo of _Foo, bound = 500

  public module Bar:
    input in : UInt<32>
    input sign : Bool
    output out : UInt<32>
    contract:
      node prec0 = intrinsic(circt_ltl_implication(...))
      require prec0
      node precl = intrinsic(circt_ltl_implication(...))
      require precl
      node post = ;;some post-condition;;
      ensure post

    inst foo1 of Foo
    inst foo2 of Foo

    ;; body of bar that uses multiple Fools ;;

  public module _Bar:
    input s_in : UInt<32>
    input s_en : Bool
    inst dut of Bar
    connect dut.in, s_in
    connect dut.en, s_en
    intrinsic(circt_verif_assert(...))

  formal testBar of _Bar, bound = 500
```

End-to-End Verification Flow

FIRRTL

lowerToHW

```
module {
  hw.module @Foo(in %in : i32, out : i32) {
    %foo.0 = verif.contract(%out) : i32 -> (i32) {
      ^bb0(%foo.0 : i32):
        %c0_i32 = hw.constant 0 : i32
        %prec0 = comb.icmp bin ugt %in, %c0_i32 : i32
        verif.require %prec0 : i1
        %c1000_i32 = hw.constant 1000 : i32
        %precl = comb.icmp bin ult %in, c1000_i32 : i32
        verif.require %precl : i1
        %post = ...
        verif.ensure %post : i1
        verif.yield %foo.0 : i32
      }
    %out = ... : i32
    hw.output %foo.0
  }


  verif.formal @testFoo(k = 500) {
    %s_in = verif.symbolic_input : i32
    %foo.0 = hw.instance "foo" @Foo(
      in: %s_in : i32
    ) -> (" " : i32)
    %spec = ...
    verif.assert %spec : i1
  }
}
```

```
hw.module @Bar(in %in : i32, in %sign : i1, out : i32) {
  %bar.0 = verif.contract(%out) {
    ^bb0(%bar.0 : i32):
      %c0_i32 = hw.constant 0 : i32
      %ingt0 = comb.icmp bin ugt %in, %c0_i32 : i32
      %prec0 = ltl.implication %sign, %ingt0 : ltl.property
      verif.require %prec0 : ltl.property
      %inlt0 = comb.icmp bin ult %in, %c0_i32 : i32
      %true = hw.constant 1 : i1
      %ns = comb.xor %sign, %true : i1
      %precl = ltl.implication %ns, %inlt0 : ltl.property
      verif.require %precl : ltl.property
      verif.ensure ...
      verif.yield %bar.0 : i32
    }
  %foo1.0 = hw.instance "foo1" @Foo(...) -> (...)
  %foo2.0 = hw.instance "foo2" @Foo(...) -> (...)
  %out = ...
  hw.output %bar.0
}

verif.formal @testBar(k = 500) {
  %s_in = verif.symbolic_input : i32
  %s_sign = verif.symbolic_input : i1
  %bar.0 = hw.instance "dut" @Bar(
    in: %s_in : i32, sign: %s_sign: i32
  ) -> (" " : i32)
  %spec = ...
  verif.assert %spec : i1
}
```

CORE

End-to-End Verification Flow

FIRRTL 

```
module {  
  hw.module @Foo(in %in : i32, out : i32) {  
    %foo.0 = verif.contract(%out) : i32 -> (i32) {  
      ^bb0(%foo.0 : i32):  
        %c0_i32 = hw.constant 0 : i32  
        %prec0 = comb.icmp bin ugt %in, %c0_i32 : i32  
        verif.require %prec0 : i1  
        %c1000_i32 = hw.constant 1000 : i32  
        %precl = comb.icmp bin ult %in, %c1000_i32 : i32  
        verif.require %precl : i1  
        %post = ...  
        verif.ensure %post : i1  
        verif.yield %foo.0 : i32  
      }  
      %out = ... : i32  
      hw.output %foo.0  
    }  
  }  
}
```

```
verif.formal @testFoo(k = 500) {  
  %s_in = verif.symbolic_input : i32  
  %foo.0 = hw.instance "foo" @Foo(  
    in: %s_in : i32  
  ) -> (" " : i32)  
  %spec = ...  
  verif.assert %spec : i1  
}
```

```
hw.module @Bar(in %in : i32, in %sign : i1, out : i32) {  
  %bar.0 = verif.contract(%out) {  
    ^bb0(%bar.0 : i32):  
      %c0_i32 = hw.constant 0 : i32  
      %ingt0 = comb.icmp bin ugt %in, %c0_i32 : i32  
      %prec0 = ltl.implication %sign, %ingt0 : ltl.property  
      verif.require %prec0 : ltl.property  
      %inlt0 = comb.icmp bin ult %in, %c0_i32 : i32  
      %true = hw.constant 1 : i1  
      %ns = comb.xor %sign, %true : i1  
      %precl = ltl.implication %ns, %inlt0 : ltl.property  
      verif.require %precl : ltl.property  
      verif.ensure ...  
      verif.yield %bar.0 : i32  
    }  
    %foo1.0 = hw.instance "foo1" @Foo(...) -> (...)  
    %foo2.0 = hw.instance "foo2" @Foo(...) -> (...)  
    %out = ...  
    hw.output %bar.0  
  }  
}
```

```
verif.formal @testBar(k = 500) {  
  %s_in = verif.symbolic_input : i32  
  %s_sign = verif.symbolic_input : i1  
  %bar.0 = hw.instance "dut" @Bar(  
    in: %s_in : i32, sign: %s_sign : i32  
  ) -> (" " : i32)  
  %spec = ...  
  verif.assert %spec : i1  
}
```

CORE

End-to-End Verification Flow

Formal

Core



PrepareForFormal

```
module {  
  // k can be a pass argument for modules  
  verif.formal @Foo(k = 500) {  
    %s_in = verif.symbolic_input : i32  
    %foo.0 = verif.symbolic_input : i32  
    %c0_i32 = hw.constant 0 : i32  
    %prec0 = comb.icmp bin ugt %in, %c0_i32 : i32  
    verif.assume %prec0 : i1  
    %c1000_i32 = hw.constant 1000 : i32  
    %prec1 = comb.icmp bin ult %in, %c1000_i32 : i32  
    verif.assume %prec1 : i1  
    %post = ...  
    verif.assert %post : i1  
    // rest of the module  
  }  
  verif.formal @testFoo(k = 500) {  
    %s_in = verif.symbolic_input : i32  
    %foo.0 = verif.symbolic_input : i32  
    %c0_i32 = hw.constant 0 : i32  
    %prec0 = comb.icmp bin ugt %in, %c0_i32 : i32  
    verif.assert %prec0 : i1  
    %c1000_i32 = hw.constant 1000 : i32  
    %prec1 = comb.icmp bin ult %in, %c1000_i32 : i32  
    verif.assert %prec1 : i1  
    %post = ...  
    verif.assume %post : i1  
    %spec = ...  
    verif.assert %spec : i1  
  }  
}
```

```
verif.formal @Bar(k = 500) {  
  %s_in = verif.symbolic_input : i32  
  %s_sign = verif.symbolic_input : i1  
  %bar.0 = verif.symbolic_input : i32  
  // body logic  
  %c0_i32 = hw.constant 0 : i32  
  %ingt0 = comb.icmp bin ugt %s_in, %c0_i32 : i32  
  %prec0 = ltl.implication %s_sign, %ingt0 : !ltl.property  
  verif.assume %prec0 : !ltl.property  
  %inlt0 = comb.icmp bin ult %s_in, %c0_i32 : i32  
  %true = hw.constant 1 : i1  
  %ns = comb.xor %s_sign, %true : i1  
  %prec1 = ltl.implication %ns, %inlt0 : !ltl.property  
  verif.assume %prec1 : !ltl.property  
  verif.assert ...  
  // foo 1 instance  
  %foo1.0 = verif.symbolic_input : i32  
  %c0_i32 = hw.constant 0 : i32  
  %prec0 = comb.icmp bin ugt %..., %c0_i32 : i32  
  verif.assert %prec0 : i1  
  %c1000_i32 = hw.constant 1000 : i32  
  %prec1 = comb.icmp bin ult %..., %c1000_i32 :  
  verif.assert %prec1 : i1  
  %post = ...  
  verif.assume %post : i1  
  // foo 2 instance  
  %foo2.0 = verif.symbolic_input : i32  
  %prec0_0 = comb.icmp bin ugt %..., %c0_i32 :  
  verif.assert %prec0_0 : i1  
  %prec1_0 = comb.icmp bin ult %..., %c1000_i32  
  verif.assert %prec1_0 : i1  
  %post_0 = ...  
  verif.assume %post_0 : i1  
  %out = ...  
}
```

```
verif.formal @testBar(k = 500) {  
  %s_in = verif.symbolic_input : i32  
  %s_sign = verif.symbolic_input : i1  
  %bar.0 = verif.symbolic_input : i32  
  %c0_i32 = hw.constant 0 : i32  
  %ingt0 = comb.icmp bin ugt %s_in, %c0_i32 : i32  
  %prec0 = ltl.implication %s_sign, %ingt0 : !ltl.property  
  verif.assert %prec0 : !ltl.property  
  %inlt0 = comb.icmp bin ult %s_in, %c0_i32 : i32  
  %true = hw.constant 1 : i1  
  %ns = comb.xor %s_sign, %true : i1  
  %prec1 = ltl.implication %ns, %inlt0 : !ltl.property  
  verif.assert %prec1 : !ltl.property  
  verif.assume ...  
  %spec = ...  
  verif.assert %spec : i1  
}
```

End-to-End Verification Flow

Formal

Core



PrepareForFormal

```
module {  
  // k can be a pass argument for modules  
  verif.formal @Foo(k = 500) {  
    %s_in = verif.symbolic_input : i32  
    %foo.0 = verif.symbolic_input : i32  
    %c0_i32 = hw.constant 0 : i32  
    %prec0 = comb.icmp bin ugt %in, %c0_i32 : i32  
    verif.assume %prec0 : i1  
    %c1000_i32 = hw.constant 1000 : i32  
    %prec1 = comb.icmp bin ult %in, c1000_i32 : i32  
    verif.assume %prec1 : i1  
    %post = ...  
    verif.assert %post : i1  
    // rest of the module  
  }  
  
  verif.formal @testFoo(k = 500) {  
    %s_in = verif.symbolic_input : i32  
    %foo.0 = verif.symbolic_input : i32  
    %c0_i32 = hw.constant 0 : i32  
    %prec0 = comb.icmp bin ugt %in, %c0_i32 : i32  
    verif.assert %prec0 : i1  
    %c1000_i32 = hw.constant 1000 : i32  
    %prec1 = comb.icmp bin ult %in, c1000_i32 : i32  
    verif.assert %prec1 : i1  
    %post = ...  
    verif.assume %post : i1  
    %spec = ...  
    verif.assert %spec : i1  
  }  
}
```

```
verif.formal @Bar(k = 500) {  
  %s_in = verif.symbolic_input : i32  
  %s_sign = verif.symbolic_input : i1  
  %bar.0 = verif.symbolic_input : i32  
  // body logic  
  %c0_i32 = hw.constant 0 : i32  
  %ingt0 = comb.icmp bin ugt %s_in, %c0_i32 : i32  
  %prec0 = ltl.implication %s_sign, %ingt0 : !ltl.property  
  verif.assume %prec0 : !ltl.property  
  %inlt0 = comb.icmp bin ult %s_in, %c0_i32 : i32  
  %true = hw.constant 1 : i1  
  %ns = comb.xor %s_sign, %true : i1  
  %prec1 = ltl.implication %ns, %inlt0 : !ltl.property  
  verif.assume %prec1 : !ltl.property  
  verif.assert ...  
  // foo 1 instance  
  %foo1.0 = verif.symbolic_input : i32  
  %c0_i32 = hw.constant 0 : i32  
  %prec0 = comb.icmp bin ugt %..., %c0_i32 : i32  
  verif.assert %prec0 : i1  
  %c1000_i32 = hw.constant 1000 : i32  
  %prec1 = comb.icmp bin ult %..., c1000_i32 :  
  verif.assert %prec1 : i1  
  %post = ...  
  verif.assume %post : i1  
  // foo 2 instance  
  %foo2.0 = verif.symbolic_input : i32  
  %prec0_0 = comb.icmp bin ugt %..., %c0_i32 :  
  verif.assert %prec0_0 : i1  
  %prec1_0 = comb.icmp bin ult %..., c1000_i32  
  verif.assert %prec1_0 : i1  
  %post_0 = ...  
  verif.assume %post_0 : i1  
  %out = ...  
}
```

```
verif.formal @testBar(k = 500) {  
  %s_in = verif.symbolic_input : i32  
  %s_sign = verif.symbolic_input : i1  
  %bar.0 = verif.symbolic_input : i32  
  %c0_i32 = hw.constant 0 : i32  
  %ingt0 = comb.icmp bin ugt %s_in, %c0_i32 : i32  
  %prec0 = ltl.implication %s_sign, %ingt0 : !ltl.property  
  verif.assert %prec0 : !ltl.property  
  %inlt0 = comb.icmp bin ult %s_in, %c0_i32 : i32  
  %true = hw.constant 1 : i1  
  %ns = comb.xor %s_sign, %true : i1  
  %prec1 = ltl.implication %ns, %inlt0 : !ltl.property  
  verif.assert %prec1 : !ltl.property  
  verif.assume ...  
  %spec = ...  
  verif.assert %spec : i1  
}
```

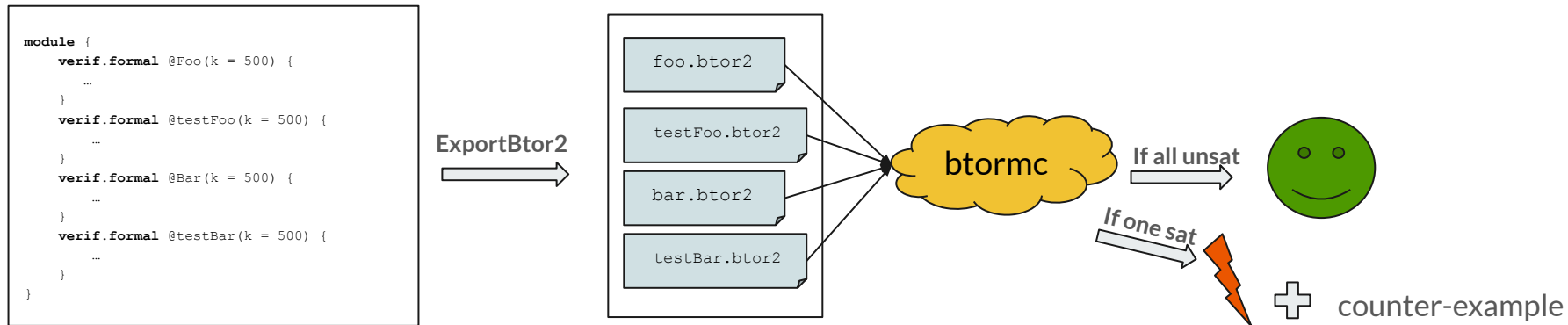


PrepareForFormal

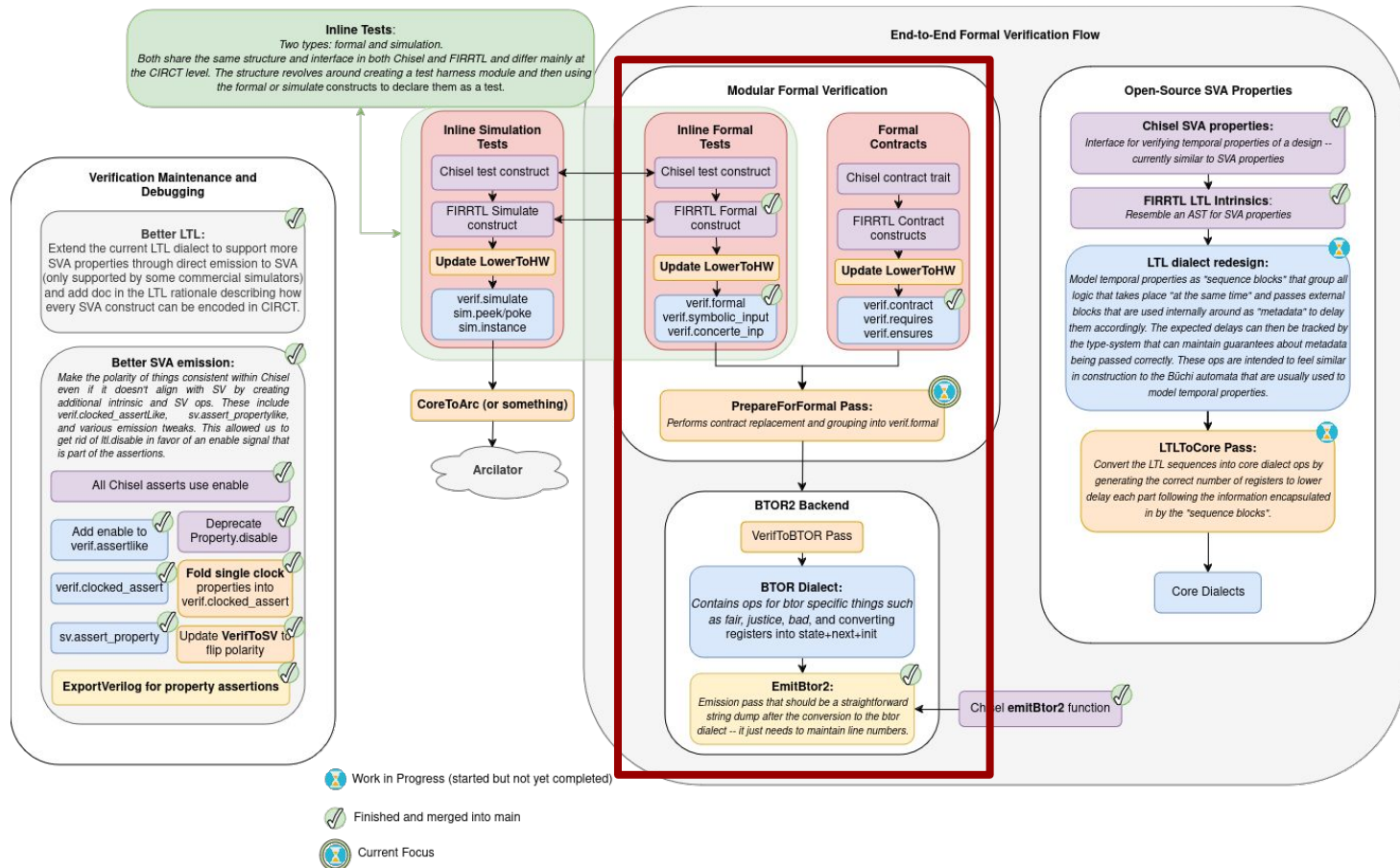
- Perform Contract replacement.
 - Module: **assume** preconditions + **assert** postconditions & body
 - Instance: **assert** preconditions + **assume** postconditions
- Create Formal Tests for Modules.
- Yield a format that can be used in formal back-ends.

End-to-End Verification Flow

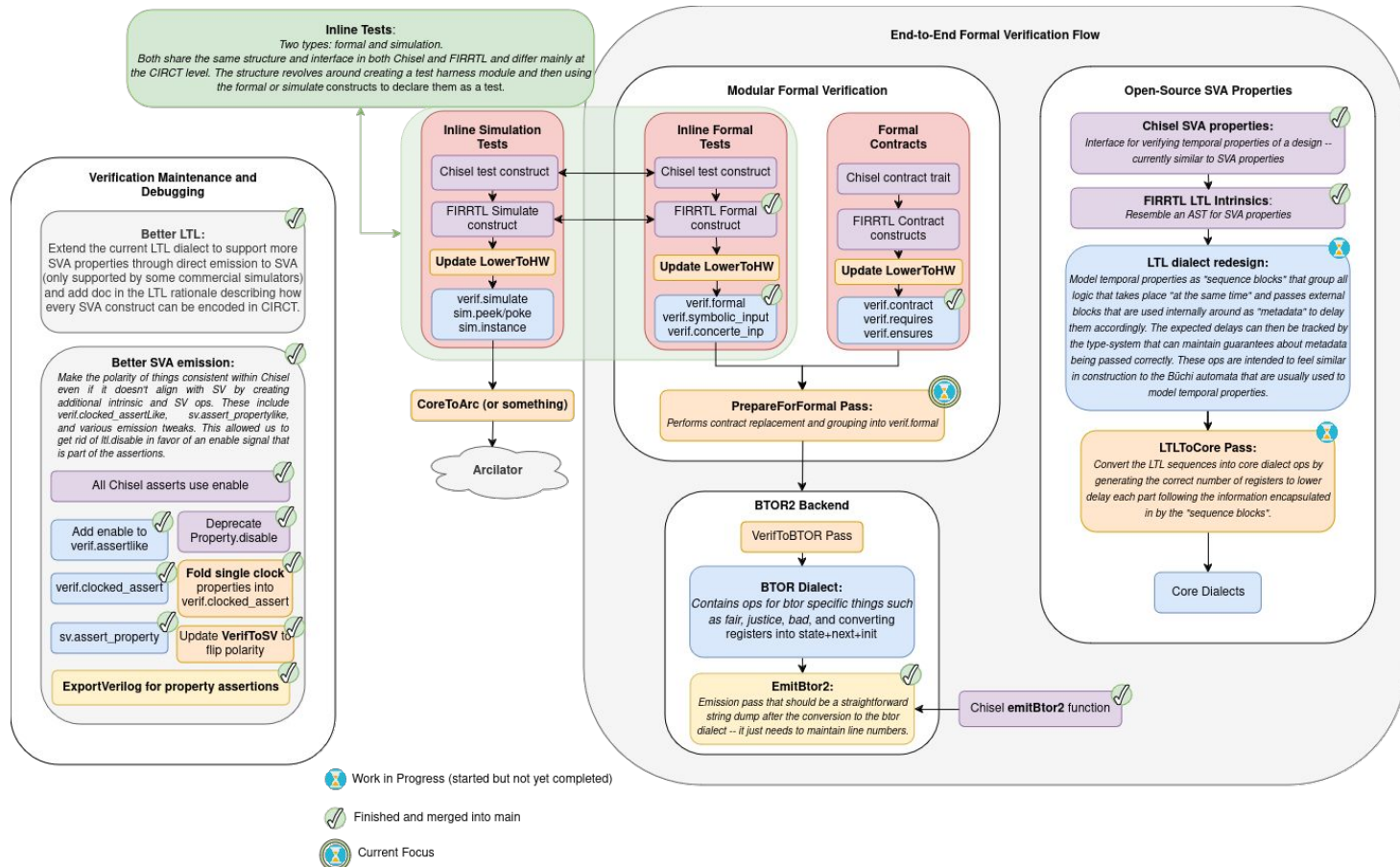
Formal



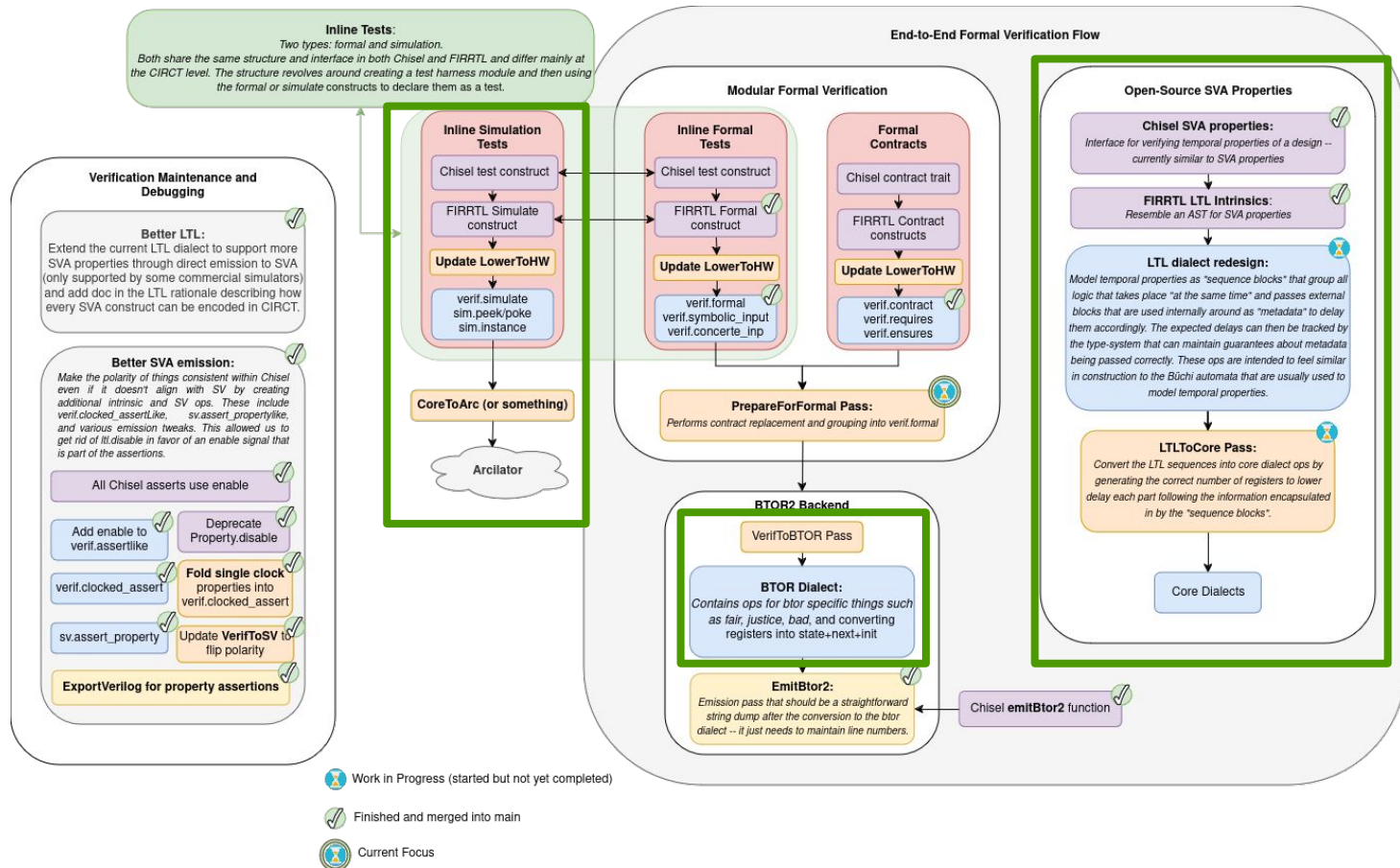
Overview:



What has been done?:



Future Work:





Conclusion

- Formally Verifying Circuits should be as **simple** and **efficient** as implementing them.
 - Verification engineers should not have to repeat work.
- Introduced a **unified interface** for writing **Formal Tests** for any back-end.
- Introduced a **formal contract** system, for retaining modularity during verification.
- Designed a compilation flow that integrates both elements in Chisel and CIRCT.

⇒ This is a WIP, the core building blocks and passes are there, still need to **provide Chisel interfaces** and **connect everything together**.

→ Please reach out if you want to help out!

What has been done?:

