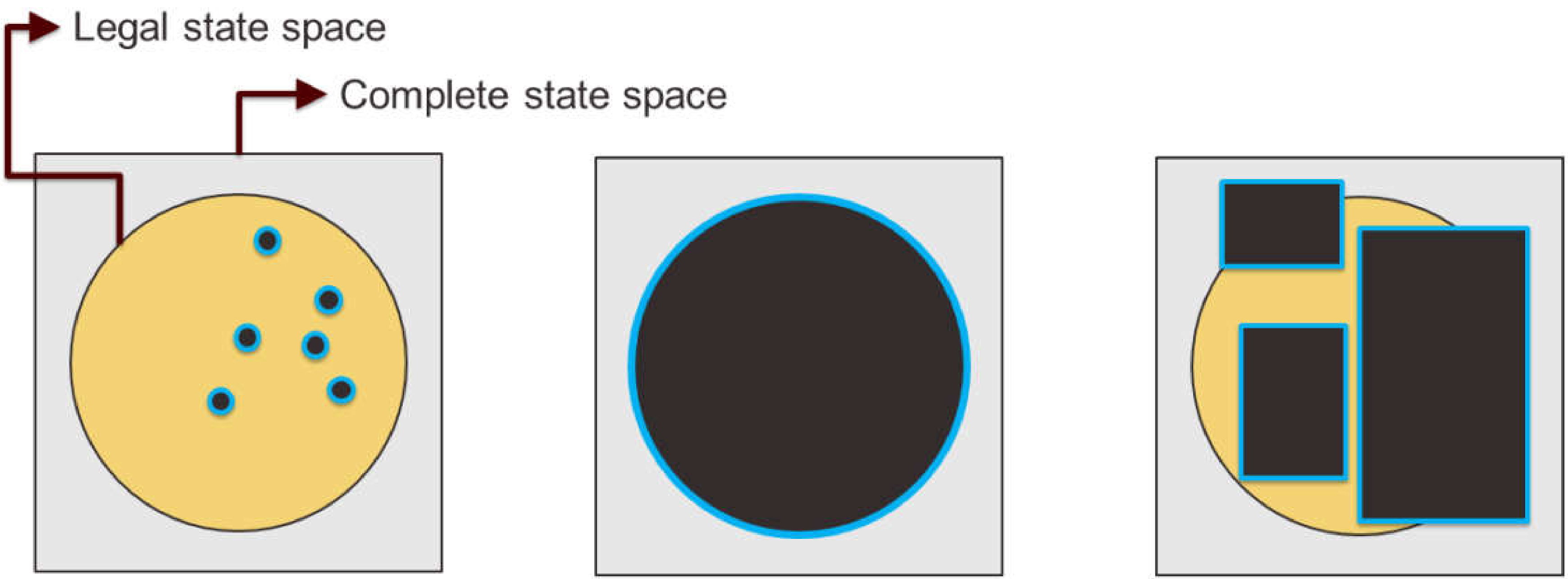
FV Note 1

Verification Completeness – Coverage

	Code Coverage	
Functional Coverage	Low	High
Low	Start of the project?	Is design complete? Try formal tools
High	Add functional coverage: include corner cases	Good coverage: check bug rate

Formal Verification – State Space



a) Simulation b) Formal (ideal) c) Formal (real)

Constraints in Formal Property Verification (FPV)

- **Constraints** play a central role in FPV.
  - They define what is **legal stimulus** to the DUT,
    - i.e., what **state space** can be reached.
- **Assertions** define the desired behavior of the DUT for the **legal stimulus**.

- **Constraints** describe how inputs to the DUT are allowed to behave, what values should be taken, and temporal relationships between inputs.
  - **Constraints** can be thought of as the stimulus in simulation.
- In constrained random simulation,
  - the **constraint solver** generates an **input vector** for the next cycle that satisfies all constraints.
- It will continue to generate cycle after cycle of stimulus
  - until the end of simulation,
  - or until it reaches a situation
    - where no legal stimulus can be generated.

➔ In contrast, **constraints** for **formal verification** can describe,
  - for example, how to <u>legally communicate</u> within a **given protocol**.

REF: https://www.edn.com/dont-over-constrain-in-formal-property-verification-fpv-flows/

Over and Under-constraining

- Writing **constraints** that exactly describe **all legal stimuli**
  - is difficult and often undesirable.
- This means that the formal environment is
  - either **under**- constrained
  - or **over**-constrained.

- **Under**-constrained means that
  - there are **fewer** constraints than required to exactly model the stimulus.
  - ➔ This means that some potentially **illegal** inputs will be driven to the DUT.

- **Over**-constrained means that there are **more** constraints than required,
  - and not **all** legal behaviors will be allowed.

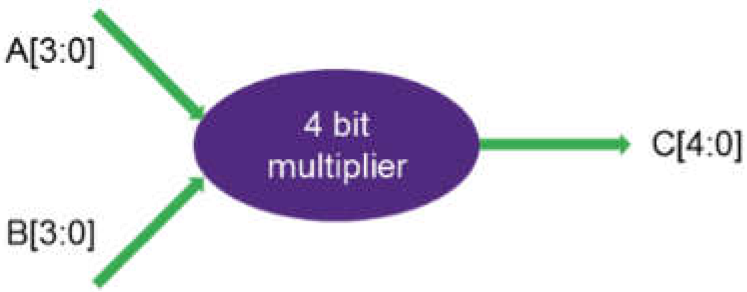
REF: https://www.edn.com/dont-over-constrain-in-formal-property-verification-fpv-flows/

An Example of Under-constrained

- Having a **slightly under-constrained** environment
  - is usually the **best approach**.

- Many designs can handle inputs and behaviors *not defined* in the specification,
  - and a *larger* **state space** in the design will be verified
    - if *fewer* constraints are used.
- An **under**-constrained environment may lead to failing **assertions**,
  - and if this is the case, **additional** constraints *need to be added*.

- For example, let's say we have a 4-bit multiplier to verify:



An Example of Under-constrained (2)

- The specification says it can multiply positive integers A and B > 0,
  - but the verification engineer assumes A and B >= 0.
- The **constraints** and the **assertion** to check the multiplier is simply:

```
assume property (@posedge clk) A >= 0;
assume property (@posedge clk) B >= 0;
assert property (@posedge clk) C == A * B;
```

- If the **property** is proven in this case –
  - for either or both A and B being zero
  - as well as for *positive* integers – then obviously
    - it will hold for A and B only being *greater than zero*.
- The **constraints** allowed for additional behaviors, which means the environment is **under**-constrained.
- Having *fewer* constraints usually also improves run time of **formal tools**.
- If the **properties** **pass**, we don't have to worry about the **under** constraining case anymore.

An Example of Over-constrained

- **Over**-constraining the formal environment is a much *bigger problem*
  - as it may hide bugs in the design.
- In effect you are not verifying as much as you think you are.

- For example,
  - assume the case of a multiplier that can multiply both *positive* and *negative* numbers,
  - but the verification engineer
    - misinterprets the specification
    - and writes **constraints** to restrict A and B to >= 0.
- Assuming the multiplier works,
  - the property above will pass,
    - and you think verification is done because all properties pass.

An Example of Over-constrained (2)

- **Over**-constraining is only a problem when it is un-intentional.

- Intentional **over**-constraining is a useful method to break up verification of a design into cases.
- One example is verification of a **memory controller**.
  - First constrain the stimulus to do **only write** transactions,
    - and then constrain it to do **only read** transactions.
  - Each of these cases is clearly over-constrained.
  - In the first case, read transactions, which are legal transactions, are not allowed,
    - and in the second case, write transactions are not allowed.
  - This is not a problem because the two cases together cover all legal stimulus.

- It is the case where only one of the cases is exercised and not the other,
  - leading the verification engineer to think that verification is done.
- A risk with intentionally **over**-constraining is that legal input values are missed,
  - and sequences such as read followed by write (in the case of the **memory controller**) are not verified.

Conflicting Constraints

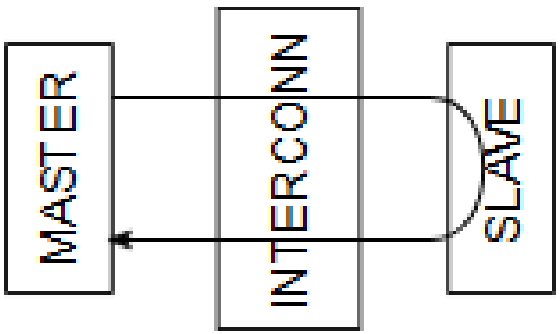
- **Constraints** limit the set of **inputs** and the **state space** explored in formal property verification (FPV).
- If the verification environment has **constraints** that conflict with each other or with statements in the design,
  - **no legal inputs** may be possible, and **none** of the **state space** in the design will be reachable.
- For example,
  - the two **constraints** below can be satisfied individually,
  - but together they produce a conflict:

equal: assume property (@ posedge clk) (wb_stb_i == wb_cyc_i)

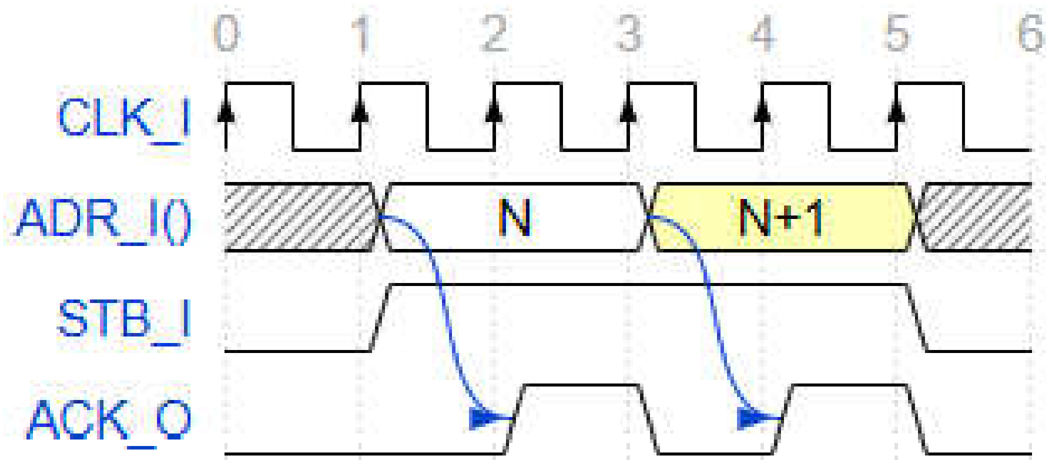
not_equal: assume property (@ posedge clk) (wb_stb_i != wb_cyc_i)

- **Conflicting constraints** can be seen as the most extreme form of an **over**-constrained environment that is so constrained that there are no legal inputs.
  - This means that no **assertions** can fail, in effect because no checking is done.
- It is analogous to saying that none of my test cases fail in simulation when the reason is that you have not executed any test cases.
- The statement is true, but it is misleading in terms of verification completeness.

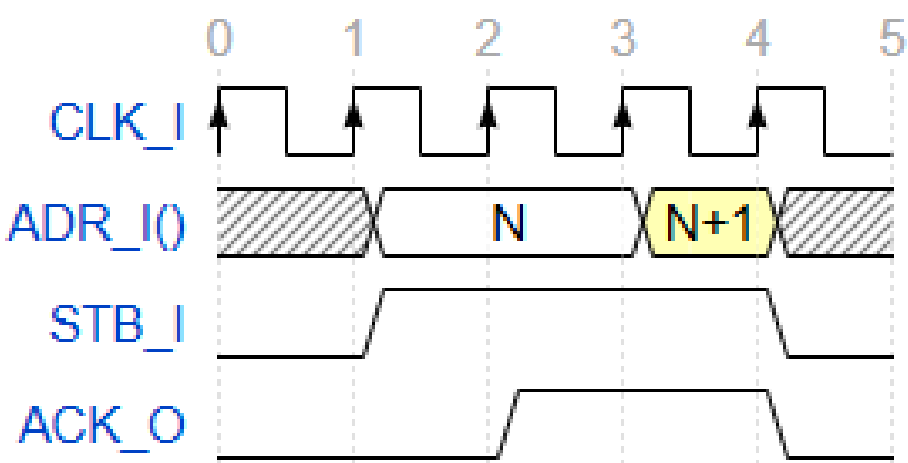
SoC Wishbone Bus



Asynchronous cycle termination path



WISHBONE Classic synchronous cycle terminated burst



Advanced synchronous terminated burst

Conflicting Constraints (2)

- Most formal tools detect **conflicting constraints** before attempting to run proofs of **properties**.
- Manually determining which properties conflict is difficult, and it is better to rely on tools to report it.

Detection of an Over-constrained Environment

Detection of an Over-constrained Environment

- Having an unintentionally over-constrained environment may hide bugs in the DUT,
  - so it is important to have a **methodology to detect** it.
- Methods described below to detect **over**-constraint include:
  1. Cover properties
  2. Vacuity checks
  3. Formal coverage analysis

Cover Properties

- **Cover properties** describe expected behaviors in the design.
  - If they fail, it may indicate an **over**-constrained environment.
- **Cover properties** are unique to each design
  - and are written by **design** or **verification engineers**
    - to ensure sequences of events can happen.
- For example: (exercise on an FIFO RTL design)

  cover property @(posedge clk) empty ##4 full;

  - The **cover property** ensures that it is possible to fill a **four-deep FIFO** in the design.
  - This means that some write logic must be working.
  - If the property fails,
    - it could be caused by
      - a design bug
      - or a **constraint** that prevents four data values from being written to the FIFO.
- **Cover properties** are a useful tool to detect an **over**-constrained environment,
  - but the main drawback is that the **designer** must write them.
    - If a **cover property** is missing, nothing is detected.

Vacuity Checks

- **Vacuity checks** are a type of **cover property** automatically generated by **formal tools**.
- **The tool** creates a check on the antecedent expression of an **assertion** with an **implication** – where the left-hand side of an assertion implies that
  - the right-hand side must hold true at some point.
- For example:

  assert property @(posedge clk) REQ && ACTIVE |-> ##2 DONE;

  - The vacuity check for the property above would be equivalent to:

    cover property @(posedge clk) REQ && ACTIVE;

  - If the vacuity check (cover property above) fails,
    - it means that the property can never fail because the antecedent is never true.
- The vacuity check may fail because of a design bug or because of a constraint.
- For example:

  assume property @(posedge clk) ACTIVE |-> !REQ;

Vacuity Checks (2)

- As we have seen in the example above,
  - **vacuity checks** can also be useful in detecting an **over**-constrained environment.
- Their main advantage over **cover properties** is that
  - they are generated automatically by **formal tools**,
  - but they rely on the existence of **properties** with implications
    - and do not check areas of the design where **properties** are sparse or missing.

Formal Coverage Analysis (FCA)

- The most effective method to detect an **over**-constrained environment
  - is to use formal coverage analysis (FCA).
- FCA uses **simulation-type** coverage goals
  - such as *line*, *expression*, *toggle*, and *FSM coverage*
    - to determine which parts of the design are reachable.

- This is similar to the use of coverage in **simulation**,
  - where you track and measure which lines and expressions have been executed or reached by the set of **test cases**.
  - If a line of code is not reached in **simulation**,
    - it may be because
      - a test has not yet been written,
      - or constrained-random **simulation** didn't run long enough to reach it.
  - Or, it may simply be impossible to reach a particular line of code.

Formal Coverage Analysis (2)

- Some parts of the design may be unreachable
  - regardless of the **constraints**.
- The structure of the RTL may prevent some lines from being reached.
- For example:
  - **if (A && B) data**
  - **if (!B) data**
- The assignment **data** will never be executed in **simulation** and is unreachable in **formal analysis** because it is impossible to reach it.
- If (A && B) is true, !B cannot be true at the same time.
- This means that it is necessary to first find the coverage goals that are structurally unreachable
  - so they can be distinguished from goals that are unreachable because of **constraints**.
- If a design is over-constrained, it means part of it is structurally reachable, yet still unreachable because of the constraints.
  - This can be detected by FCA.

Formal Coverage Analysis (3)

- Any line, expression, toggle, or FSM coverage goal that is unreachable in the presence of constraints
  - but is reachable when no constraints are applied is a potential problem due to over-constraining.
- The designer needs to determine if the over-constraining in each case is intended or harmless, or if it needs to be addressed and constraints changed.
- For example, if we have the constraint and finite-state machine:

```
assume always @(posedge clk) B

case (state)
    IDLE: begin if (B == 0) next_state  end
    ACTIVE: begin
          if B
         else next_state          end
    END: begin next_state          end
  endcase
```

- The state END will never be reached because of the constraint,
  - and we have an over-constrained environment.
- The FSM coverage and line coverage goal will fail for the state END.

Thank You