PDR without a PhD

Michael Zhang · 2026-11-09 · formal verification, model checking, rust

Table of Contents

1. What this is 1
2. What is PDR? 1
3. PDR in action 2
4. The pieces 5
   4.1. Transition system 6
   4.2. Cubes and clauses 7
   4.3. Frames 8
   4.4. Relative induction 11
5. The algorithm 13
   5.1. Main loop 14
      5.1.1. Recursive cube blocking 14
   5.2. Clause propagation 16
6. Implementation tips and tricks 17
7. Optimizing PDR 18
8. Testing PDR 19
9. Acknowledgements 19
Bibliography 19

1. What this is

This post came about while I was implementing the *Property-Directed Reachability* (PDR) model-checking algorithm in the Patronus hardware verification toolkit. As most people probably do, I tried to first understand the algorithm by reading papers ([1], [2]). However, most papers (even the implementation-focused ones) discuss PDR from a high-level, mathematical perspective that was hard to translate into working code. While I initially got a general idea of the algorithm, I spent an embarrassing amount of time trying to understand the subtle concepts that formed that gap between theory and implementation.

The papers did nothing wrong; they just assumed a lot of background that I didn't have then. My goal here is to make a post that I would've wanted then: the data structures, loops, and solver queries in code that run PDR; the little bits of advice on where PDR implementations often go wrong. I hope this post enables the PDR algorithm, a foundation for many state-of-the-art hardware model checkers, to be more accessible for the programmer.

2. What is PDR?

PDR is an algorithm that can prove that some hardware **never** violates a property. Not "never in the first 50 cycles," but **never**—for all executions of unbounded length.

This sounds complicated (and many papers make it look this way!). In reality, the algorithm is quite short. If you've ever implemented a worklist algorithm (such as a graph traversal), the structure should be particularly familiar.

However, before we move on, I have three things to clear up front:

1. **PDR requires an incremental SAT/SMT solver**: PDR requires lots of SAT queries. Making the solver start on a clean slate every query will waste a lot of time. I recommend using an incremental solver such as Bitwuzla, Z3, cvc5, or Yices 2. Personally, I prefer Bitwuzla for its strong performance on bitvector and array theories that show up a lot in hardware verification.
2. **The vanilla PDR implementation here is correct, but slow**: Most modern PDR implementations rely on a plethora of optimizations to work effectively on large industrial designs. We'll cover some common optimizations (but not implement them) to show the potential improvements, but this post will mostly focus on getting the skeleton of PDR down pat.
3. **This post provides examples in Rust and Verilog**: While knowing Rust and Verilog might be helpful, readers with some programming and/or hardware design experience should be able to follow along.

3. PDR in action

Let's say we have the following circuit:

```
module Pipe(input clk, output bad);
  reg a = 1'b0;
  reg b = 1'b0;
  reg c = 1'b0;

  always @(posedge clk) begin
    a <= 1'b1;
    b <= a;
    c <= b;
  end

  assign bad = (c && !b);
endmodule
```

which is simply a three-cycle register pipeline that eventually sets register c to 1.

The property to prove here: register c set to 1 implies b is set to 1 (i.e. $c = 1 \Rightarrow b = 1$, the negation of `bad`). This property may seem trivial by inspection (since 1 is propagated across $a \rightarrow b \rightarrow c$). However, we will see how PDR discovers this fact on its own.

First, PDR will keep track of a finite sequence of *frames*, called a *frame trace*. A frame (notated as $\mathbf{R}_k$) is a **logical formula** that includes states that could *possibly* be reached in k steps or less. *Possibly* is the key here: the frame itself is almost always an over-approximation, so it could be the case that some states in $\mathbf{R}_k$ are actually **unreachable** in k steps or less. We say that these states are *spurious*. The main job of PDR is to tighten this approximation (by excluding spurious property violations) to eventually prove the property, or find a true counterexample (that is, an execution path from the initial states to a property violation).

We start out the algorithm with the first frame, $\mathbf{R}_0$, that only contains the initial reset states. We notate this as

$$\mathbf{R}_0 : a = 0 \wedge b = 0 \wedge c = 0$$

Notice that $\mathbf{R}_0$ is special since there are no spurious states.

We then test for property violations in $\mathbf{R}_0$: there are clearly none since $c = 0$. Next, we add a blank frame $\mathbf{R}_1 : \top$ to the frame trace. Since $\mathbf{R}_1$ is blank, there are no restrictions on our approximation of reachable states in one cycle or less. As far as PDR is concerned, *any state* can be reached in one cycle or less.

This is where we run into trouble. The solver will claim that there's a property violation $a = 0 \wedge b = 0 \wedge c = 1$ in $\mathbf{R}_1$ (i.e. there's a property violation in the initial reset states or cycle 1). Clearly, this is not the case since registers a and b should have been set to 1 before c was set to 1. The main problem is that $\mathbf{R}_1$ is too loose and falsely admits this property violation. To fix this, we need to prove that $a = 0 \wedge b = 0 \wedge c = 1$ **cannot** be reached from the initial reset states $\mathbf{R}_0$. PDR does this by *blocking* $a = 0 \wedge b = 0 \wedge c = 1$ at $\mathbf{R}_1$ (in general, PDR blocks at the target frame and all lower frames, except $\mathbf{R}_0$, which is not an over-approximation by construction and requires no refinement). For now, assume that PDR has some reachability checker (more details on how this works later) that proves $a = 0 \wedge b = 0 \wedge c = 1$ is unreachable from $\mathbf{R}_0$. Then, we can explicitly block the spurious property violation by conjuncting its negation to $\mathbf{R}_1$. $\mathbf{R}_1$ then becomes

$$\mathbf{R}_1 : a \neq 0 \vee b \neq 0 \vee c \neq 1$$

We then try to find more property violations at $\mathbf{R}_1$. Unfortunately, $a = 1 \wedge b = 0 \wedge c = 1$ is another violation! We follow the same steps as before and find that this property violation is also spurious in one cycle or less. $\mathbf{R}_1$ then becomes

$$\mathbf{R}_1 : (a \neq 0 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 1 \vee b \neq 0 \vee c \neq 1)$$

This means that our approximation of reachable states in one cycle or less has excluded all cases where $c = 1 \wedge b = 0$ (i.e. these states have been proven to be spurious). Therefore, $\mathbf{R}_1$ has no property violations and we have proven that the hardware satisfies the property in the initial reset states and cycle 1.

There's one more step that tries to prove that the $\mathbf{R}_1$ formula holds in *all cycles*. First, just as before, we add a blank frame $\mathbf{R}_2 : \top$ to the frame trace. However, instead of finding property violations at $\mathbf{R}_2$ first, we check if any of the states blocked by $\mathbf{R}_1$ can also be blocked in $\mathbf{R}_2$.

We first check the original state that we blocked in $\mathbf{R}_1$: $a = 0 \wedge b = 0 \wedge c = 1$. Since $a = 1$ after every transition, we cannot reach $a = 0 \wedge b = 0 \wedge c = 1$, so it stays blocked in $\mathbf{R}_2$.

In contrast, when we check the other spurious state blocked in $\mathbf{R}_1$, $a = 1 \wedge b = 0 \wedge c = 1$, we find the predecessor $a = 0 \wedge b = 1 \wedge c = 0$ in $\mathbf{R}_1$, which transitions into our spurious state $a = 1 \wedge b = 0 \wedge c = 1$ in one cycle. Therefore, we *cannot conclude* whether $a = 1 \wedge b = 0 \wedge c = 1$ is spurious in $\mathbf{R}_2$.

Therefore, the only state we can block in $\mathbf{R}_2$ so far is $a = 0 \wedge b = 0 \wedge c = 1$. We can then add its negation to $\mathbf{R}_2$:

$$\mathbf{R}_2 : a \neq 0 \vee b \neq 0 \vee c \neq 1$$

Once again, we check for property violations at the last frame $\mathbf{R}_2$. In this case, $a = 1 \wedge b = 0 \wedge c = 1$ is also a property violation. Clearly, we know that this state is

spurious because b must be set to 1 after a is set to 1, and c is set to 1 after b is set to 1.

To prove that this property violation is spurious, PDR will follow a similar procedure from before. However, instead of testing whether $a = 1 \wedge b = 0 \wedge c = 1$ is reachable from $\mathbf{R}_0$, we first see if it's reachable from $\mathbf{R}_1$. This is clearly the case (since we previously demonstrated the state $a = 0 \wedge b = 1 \wedge c = 0$ is in $\mathbf{R}_1$ and can transition to $a = 1 \wedge b = 0 \wedge c = 1$ in $\mathbf{R}_2$).

This hiccup doesn't mean we failed, however. Recall that $\mathbf{R}_1$ is an *over-approximation* of reachable states at step 1 and less. Therefore, we can conjecture that $a = 0 \wedge b = 1 \wedge c = 0$ is a spurious state at $\mathbf{R}_1$ itself. To concretize this fact, we can *recursively* run our procedure on $a = 0 \wedge b = 1 \wedge c = 0$ with respect to $\mathbf{R}_0$. In this case, we clearly cannot reach $a = 0 \wedge b = 1 \wedge c = 0$ in the first step from the initial states $\mathbf{R}_0$. Therefore, we can successfully refine $\mathbf{R}_1$ to block the new spurious state:

$$\mathbf{R}_1 : (a \neq 0 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 1 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 0 \vee b \neq 1 \vee c \neq 0)$$

However, $a = 1 \wedge b = 0 \wedge c = 1$ has another predecessor $a = 0 \wedge b = 1 \wedge c = 1$. By a similar process, we can also conclude that $a = 0 \wedge b = 1 \wedge c = 1$ is spurious in $\mathbf{R}_1$. Then, $\mathbf{R}_1$ becomes

$$\mathbf{R}_1 : (a \neq 0 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 1 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 0 \vee b \neq 1 \vee c \neq 0) \wedge (a \neq 0 \vee b \neq 1 \vee c \neq 1)$$

This ensures that we can safely assert that $a = 1 \wedge b = 0 \wedge c = 1$ is spurious in $\mathbf{R}_2$. We can then update $\mathbf{R}_2$ to get:

$$\mathbf{R}_2 : (a \neq 0 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 1 \vee b \neq 0 \vee c \neq 1)$$

Notice that $\mathbf{R}_2$ is now safe since we blocked all states where $c = 1 \wedge b = 0$. We can then add a new blank frame $\mathbf{R}_3$.

Then, we once again try to test if states blocked in lower frames can hold in higher frames. Eliding the exact details, PDR will find that all states blocked in $\mathbf{R}_1$ are also blocked in $\mathbf{R}_2$ (with the help of the reachability checker). We then get

$$\mathbf{R}_2 : (a \neq 0 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 1 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 0 \vee b \neq 1 \vee c \neq 0) \wedge (a \neq 0 \vee b \neq 1 \vee c \neq 1)$$
$$\mathbf{R}_2 = \mathbf{R}_1$$

This means that $\mathbf{R}_2$ has not changed from $\mathbf{R}_1$, reaching a *fixpoint*. Since $\mathbf{R}_1$ contains the initial states, has no property violations, and is preserved under a transition, the property will hold for all cycles. In other words, $\mathbf{R}_1$ is an *inductive invariant* (more on this later!) that proves the property. We can now assert that $(a \neq 0 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 1 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 0 \vee b \neq 1 \vee c \neq 0) \wedge (a \neq 0 \vee b \neq 1 \vee c \neq 1)$ (simplified to $(c = 1 \Rightarrow b = 1) \wedge (b = 1 \Rightarrow a = 1)$) holds in all cycles of our `Pipe` circuit! Therefore, since $(c = 1 \Rightarrow b = 1) \wedge (b = 1 \Rightarrow a = 1) \Rightarrow (c = 1 \Rightarrow b = 1)$, we successfully prove the property.

Overall the PDR algorithm performs the following high-level steps:

Algorithm 1: Property-Directed Reachability (PDR)

1: ▷ Recursively refine ft if s is spurious at frame k and return whether s is actually spurious at frame k
2: **procedure** BLOCK(ft, s, k)

```
    if k = 0 then
        ▷ s is in R_0, so it must be a true counterexample
        return FALSE
    else
        while s has a predecessor t in R_{k-1} ∧ ¬s do
            if ¬BLOCK(ft, t, k − 1) then
                return FALSE
            end
        end
        conjunct ¬s to R_1, ..., R_k
        return TRUE
    end
end
▷ S is the target hardware design, P is the property we want to prove about S
procedure PDR(S, P)
    ▷ Frame trace initialized with R_0 (where initial_states extracts the initial
    reset states of S)
    ft ← [initial_states(S)]

    while TRUE do
        while there is a state s of S in the frontier (last) frame R_N that violates
        P do
            if ¬BLOCK(ft, s, N) then
                ▷ If s is not spurious, then property violation can truly be reached
                from initial states
                return FAIL
            end
        end

        ▷ Add blank frame (i.e. ⊤) to ft; N now refers to this new frontier frame
        ft.append(⊤)

        ▷ Check if states can be blocked at higher frames
        for 0 < i < N do
            for state s blocked in R_i do
                if s is unreachable in one step from R_i ∧ ¬s then
                    conjunct ¬s to R_{i+1}
                end
            end
        end

        if ∃0 < i < N such that R_i = R_{i+1} then
            ▷ Frames reach fixpoint: inductive invariant proving P found
            return SUCCESS
        end
    end
end
```

4. The pieces

Next, we'll go over the foundations of PDR and implement them in Rust.

4.1. Transition system

PDR can't work directly on Verilog. Instead, PDR works on an FSM-like abstraction called a *finite-state transition system*, which in practice is like the original RTL represented as logical formulas.

```
/// sign = polarity, magnitude = id
type Literal = i64;

// [`Formula`] is a bog-standard Boolean AST: True, False, Lit,
And, Or, Not
#[derive(Debug, Clone)]
enum Formula {
  /// Constant Boolean true
  True,

  /// Constant Boolean false
  False,

  /// Boolean variable
  Lit(Literal),

  /// ANDed formulas
  And(Box<Formula>, Box<Formula>),

  /// ORed formulas
  Or(Box<Formula>, Box<Formula>),

  /// NOT formula
  Not(Box<Formula>),
}

struct TransitionSystem {
  /// State literals (think registers, latches, and other
stateful elements)
  states: Vec<Literal>,

  /// Free inputs chosen by the environment (think non-clock and
non-reset `input`s in Verilog)
  inputs: Vec<Literal>,

  /// Boolean formula over state literals that asserts the
initial states
  init: Formula,

  /// Boolean formula over state literals, input literals,
  /// and **next-state** copies of state literals; asserts
  /// that second state follows first state in one cycle
  trans: Formula,
}
```

Here, a `Literal` is a *literal*: an identifier to a Boolean variable, or its negation. Since the sign encodes polarity, variable IDs should start at 1 (0 cannot be negated). A

`Formula`* is a Boolean formula composed of literals and constants connected by `AND`s, `OR`s, and `NOT`s.

In our `Pipe` circuit,

- `states` (notated as X) would be $\{a, b, c\}$, the three internal registers
- `inputs` (notated as Y) would be $\{\}$ (since we don't count the clock as a traditional input)
- `init` (notated as $\mathbf{I}(X)$) would be $a = 0 \wedge b = 0 \wedge c = 0$
- `trans` (notated as $\mathbf{T}(X, Y, X')$) would be $a' = 1 \wedge b' = a \wedge c' = b$, where $a', b', c' \in X'$ are the *primed* versions (next-state copies) of $a, b, c \in X$, respectively

We'll usually shorten $\mathbf{I}(X)$ to $\mathbf{I}$ and $\mathbf{T}(X, Y, X')$ to $\mathbf{T}$.

To get this representation, the frontend of the verification pipeline would need to compile high-level Verilog into some IR (most commonly AIGER for bit-level designs like the Boolean one here, or BTOR2 for word-level designs, which must be bit-blasted to fit our Boolean `TransitionSystem`). This flattens the combinational logic into $\mathbf{T}$ and abstracts all stateful elements and inputs as variables that can be directly mapped in our `TransitionSystem`.

Primed variables are key in PDR. Almost all solver queries in the algorithm will be of the form "placing a constraint at this cycle, and here's a constraint for next cycle," glued together with $\mathbf{T}$ (e.g. `SAT`?$[\mathbf{R}_k \wedge \neg c \wedge \mathbf{T} \wedge c']$, where the literals in $\mathbf{R}_k$ and $\neg c$ are unprimed, while those in c' are primed). This will always require priming unprimed literals for the next cycle constraints, which quickly becomes confusing and can lead to many latent bugs (trust me, I've had to deal with a lot of them). Therefore, before implementing PDR, you **must** have your priming representation down pat.

To implement this in Rust, we might add a new field `prime_map : FxHashMap<Literal, Literal>` that maps from the unprimed literal to the primed version. Then, we could add a priming helper function `prime(&self, lit: Literal) -> Literal` to prime unprimed literals with `prime_map`. Be careful with negative literals: `prime` must preserve the sign (i.e. `prime(-l) = -prime(l)`), since some formulas we use will contain negated literals.

Finally, some vocab. We say a *trace* is a sequence of **states** in the transition system where the first state satisfies $\mathbf{I}$ and each consecutive pair satisfies $\mathbf{T}$ for some inputs. A state is *reachable* if there's a trace that ends with it. PDR's main goal is to prove some property $\mathbf{P}$ over **all reachable states**.

4.2. Cubes and clauses

You'll see these two words constantly in papers, but they're simply two shapes of formulas:

- A *cube* is a **conjunction** (AND) of literals. An example would be $x \wedge \neg y \wedge z$. They're used in PDR to describe a region of state (most of the time, a single state). Solver models are usually a cube of state assignments.

*For non-Rust programmers, the `Box` in the recursive constructors represents a *unique* pointer to heap memory (similar to `std::unique_ptr` in C++). This is required for recursive types in Rust to avoid infinitely-sized values in the stack.

- A *clause* is a **disjunction** (OR) of literals. An example would be $\neg x \vee y \vee \neg z$. They're usually conjuncted in non-initial frames to **block** spurious states (e.g. $(a \neq 0 \vee b \neq 0 \vee c \neq 1) \wedge (a \neq 1 \vee b \neq 0 \vee c \neq 1)$, a conjunction of clauses, blocked the spurious states $a = 0 \wedge b = 0 \wedge c = 1$ and $a = 1 \wedge b = 0 \wedge c = 1$ in the `Pipe` example).

These structures are related by negation through de Morgan's Laws: `NOT` a cube, and you get a clause, and vice versa. In our `Pipe` example, we negated spurious states (represented as cubes) to add as clauses to frames.

We can then write these structures in Rust:

```
#[derive(Debug, Clone)]
struct Cube {
  literals: Vec<Literal>,
}

#[derive(Debug, Clone)]
struct Clause {
  literals: Vec<Literal>,
}

impl Cube {
  /// # Returns
  /// [`Formula`] representing this cube
  fn to_formula(&self) -> Formula {
    self.literals
        .iter()
        .fold(
          Formula::True,
          |acc, &l| Formula::And(Box::new(Formula::Lit(l)),
Box::new(acc)),
        )
  }

  /// # Returns
  /// [`Clause`] representing the negation of this cube
  fn negate(&self) -> Clause {
    Clause {
      literals: self.literals.iter().map(|&l| -l).collect(),
    }
  }
}

/// Implement similar methods for [`Clause`], but mirror for
[`negate`] (i.e. [`negate`] should return a [`Cube`])
impl Clause { ... }
```

4.3. Frames

We discussed what a frame means in our `Pipe` example, but haven't concretized it in code yet. We try to do that here with the foundations that we've already built.

Fundamentally, a frame is a conjunction of clauses, where each clause blocks a spurious state. In Rust, we can represent this as

```
/// A frame that contains states that satisfy `clauses[0] /\
clauses[1] /\ ...`
#[derive(Debug, Default)]
struct Frame {
  /// The clauses part of this frame
  clauses: Vec<Clause>,
}
```

The only exception to this rule is $\mathbf{R}_0$, the initial frame that only contains the initial reset states. In other words, $\mathbf{R}_0$ is described directly by the initial states, rather than by blocking clauses. Implementations have different solutions to this problem: some fold $\mathbf{R}_0$ into the frame trace, or hoist it into a special field. I chose the latter option in the Patronus implementation to ensure consistency in the frame representation (since $\mathbf{R}_0$ will be represented as an arbitrary `Formula` given by $\mathbf{I}$, while other frames will be represented as sets of clauses). However, it led to an "off-by-one" indexing issue that I needed to keep at the back of my mind.

This leads to our final PDR state structure:

```
/// Frame identifier
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
enum FrameId {
  /// Init frame
  Init,

  /// Non-init frame (with non-zero index)
  Finite(NonZeroUsize),
}

impl FrameId {
  /// # Returns
  /// Decremented frame index (e.g. `Finite(5) |-> Finite(4)`,
`Finite(1) |-> Init`)
  ///
  /// # Panics
  /// If [self] is [`FrameId::Init`]
  fn decrement(&self) -> Self { ... }

  /// # Returns
  /// Incremented frame index (e.g `Finite(4) |-> Finite(5)`,
`Init |-> Finite(1)`)
  fn increment(&self) -> Self { ... }
}

struct Pdr {
  /// Initial frame R_0
  init_frame: Formula,

  /// Frame trace, with indexing off-by-one (i.e. frames[0] |->
R_1, ...)
  frames: Vec<Frame>,
```

```
}

/// Implement proper frame indexing, noting the off-by-one
situation for non-init frames
/// Indexing should panic if initial frame is referenced (since
the frame doesn't actually exist)
impl Index<FrameId> for Pdr { type Output = Frame; ... }
impl IndexMut<FrameId> for Pdr { ... }

impl Pdr {
  /// # Returns
  /// Iterator over all non-init frame identifiers
  fn iter(&self) -> impl Iterator<Item = FrameId> + '_ { ... }
}

impl Pdr {
  /// Initialize PDR with initial frame
  fn init(init_frame: Formula) -> Self {
    Self {
      init_frame,
      frames: vec![],
    }
  }

  /// # Returns
  /// Boolean formula representing state included in frame
  fn frame_assumptions(&self, idx: FrameId) -> Formula { ... }

  /// # Returns
  /// Index of frontier frame
  fn frontier(&self) -> FrameId { ... }

  /// Add blank frame to end of frame trace
  fn add_frame(&mut self) { ... }

  /// Add clause to a specific frame
  fn add_clause(&mut self, clause: Clause, idx: FrameId) { ... }
}
```

Finally, three facts hold of the frame trace at all times:

1. $\mathbf{R_0} = \mathbf{I}$ **and higher frames become weaker**: this means that $\mathbf{R}_i \Rightarrow \mathbf{R}_{i+1}$ for $i \geq 0$, so anything true in a lower frame **must** be true at a higher frame. Intuitively, higher frames must include **at least as many states** and **no more clauses** (i.e. have no more constraints) than lower frames. This holds by construction in PDR, as we will see later.
2. **Each frame is closed under one cycle into the next frame**: this means that $\mathbf{R}_i \wedge \mathbf{T} \Rightarrow \mathbf{R}'_{i+1}$ (where $\mathbf{R}'_{i+1}$ is $\mathbf{R}_{i+1}$ with all of its unprimed literals replaced with primed ones). Intuitively, any state in $\mathbf{R}_i$ will transition into a state in $\mathbf{R}_{i+1}$. This essentially enforces what it means to be a frame in the frame trace. It's also the reason why propagating all clauses from one frame to the next implies the discovery of an inductive invariant.

3. **Every frame, except the frontier, satisfies the property**: this means that $\mathbf{R}_i \Rightarrow \mathbf{P}$, where $i < N$ and N is the index of the frontier frame. The frontier frame $\mathbf{R}_N$ is exempt because PDR is actively clearing it of spurious property violations.

4.4. Relative induction

One more idea, and it's doing the real work for the reachability checker from Section 3.

Recall ordinary induction from a discrete math class. Usually you have a base case, and some inductive case where you assume that a property holds at some step and you try to prove that it holds at the next step.

This paradigm maps nicely to our finite-state transition system. For a property $\mathbf{A}$ to be *inductive*, we just need to show that $\mathbf{A}$ holds initially and survives all transitions. Mathematically,

$$\mathbf{I} \Rightarrow \mathbf{A}$$
$$\mathbf{A} \wedge \mathbf{T} \Rightarrow \mathbf{A}'$$

The first implication is the "base case," and the second implication is the "inductive case."[†] For all designs that satisfy the target property $\mathbf{P}$, PDR will find an *inductive invariant* that satisfies $\mathbf{P}$.

The main catch is that properties are rarely inductive on their own. We just know too little about where we started in the inductive case to prove anything meaningful.

How do we fix this? We can simply weaken the requirements. This is the foundation of *relative induction*. Instead of just relying on a property $\mathbf{A}$ as an inductive hypothesis, we introduce a new *strengthening lemma* $\mathbf{B}$. We then say that $\mathbf{A}$ is *inductive relative to* $\mathbf{B}$ iff

$$\mathbf{I} \Rightarrow \mathbf{A}$$
$$\mathbf{B} \wedge \mathbf{A} \wedge \mathbf{T} \Rightarrow \mathbf{A}'$$

In plain English, what relative induction asks is "can $\mathbf{A}$ hold for the initial states and survive the transition if I assume that $\mathbf{B}$ is true?" In essence, the "inductiveness" of $\mathbf{A}$ becomes "conditional" on $\mathbf{B}$.

This is exactly what PDR applies when reasoning against frames. Throughout PDR, we replace $\mathbf{A}$ with the negation of some cube c (i.e. the clause $\neg c$) that we want to block, and $\mathbf{B}$ with some frame $\mathbf{R}_k$ that we're testing against. If $\neg c$ is inductive relative to $\mathbf{R}_k$, then we know that c cannot be reached in one step from states included in $\mathbf{R}_k$ (other than c itself). Therefore, we can block c in $\mathbf{R}_{k+1}$.

Else, c is reachable from states in $\mathbf{R}_k$. If $k = 0$, then c must truly be reachable from the initial states (i.e. c is not spurious), by definition. However, since all $\mathbf{R}_k$ where $k > 0$ are over-approximations of reachable state, the predecessor to c might be spurious itself, possibly making c also spurious. Therefore, as in Section 3, we need to recursively test the relative inductiveness of c's predecessors with respect to their previous frames to check if c is actually reachable.

This process is the basis of reachability checking in PDR. We'll see it in action in Section 5.1.1.

[†]This is usually called the *consecution case* in the literature.

Next, we concretize the relative induction proof in the PDR implementation with the following SAT query:

$$\text{SAT?}[\mathbf{R}_k \wedge \neg c \wedge \mathbf{T} \wedge c']$$

In PDR, the cube c is either

1. a property violation or a state leading to a property violation that we want to block in $\mathbf{R}_{k+1}$ (i.e. trying to block property violations at the frontier frame), or
2. a state already blocked in $\mathbf{R}_k$ that we also want to block in $\mathbf{R}_{k+1}$ (i.e. trying to move blocked states to higher frames)

Note that $\neg c$ (which is a clause) is inductive relative to $\mathbf{R}_k$ if the query is **unsatisfiable** (`UNSAT`). Get used to this negated form for solver queries in PDR: `UNSAT` is almost always what you want from a query.

Since we use the relative induction query so much in PDR, I like to factor it out as a helper function.

To implement this function, assume that we have the following solver helper functions (the implementation of these is dependent on the solver interface):

```
/// # Returns
/// Whether [form] is satisfiable
///
/// # Panics
/// If satisfiability is `UNKNOWN`
fn check_sat(form: &Formula) -> bool;

/// # Returns
/// [`Cube`] describing the model of the last query, projected
onto [states]
fn get_model(states: &[Literal]) -> Cube;
```

Then,

```
/// # Returns
/// Whether the negated [cube] is inductive relative to frame
[idx]
fn rel_ind(
  pdr_state: &Pdr,
  sys: &TransitionSystem,
  cube: &Cube,
  idx: FrameId,
) -> bool {
  // Primed cube
  let primed_cube = Cube {
    literals: cube.literals.iter().map(|&l|
sys.prime(l)).collect(),
  };

  // Negated cube
  let neg_cube = cube.negate();

  // Get frame constraints
  let frame = Box::new(pdr_state.frame_assumptions(idx));
```

```
  // Run relative induction  query `SAT?[R_idx /\ neg c /\ T /\
c']`
  !check_sat(
    &Formula::And(
      frame,
      Box::new(Formula::And(
        Box::new(neg_cube.to_formula()),
        Box::new(Formula::And(
          Box::new(sys.trans.clone()),
          Box::new(primed_cube.to_formula()),
        )),
      )),
    )
  )
}
```

The careful reader may notice that we're missing the initial state check for $\neg c$ (i.e. $\mathbf{I} \Rightarrow \neg c$). In vanilla PDR, this check is implied by the structure of the algorithm itself, as we will see next.

5. The algorithm

With all the foundations out of the way, we can start writing the PDR algorithm directly in Rust:

```
/// # Returns
/// Whether [sys] satisfies [prop] for all reachable states
fn pdr(sys: &TransitionSystem, prop: &Formula) -> bool {
  let mut pdr_state = Pdr::init(sys.init.clone());

  loop {
    // Get frontier index and frontier frame
    let front_idx = pdr_state.frontier();
    let mut front_frame = pdr_state.frame_assumptions(front_idx);

    // Main loop: run query `SAT?[R_N /\ neg P]` until `UNSAT`
and try to block counterexample otherwise
    while check_sat(
      &Formula::And(
        Box::new(front_frame),
        Box::new(Formula::Not(Box::new(prop.clone()))),
      )
    ) {
      // Extract model
      let model = get_model(&sys.states);

      // Try to recursively block the model
      if !block_cube(&mut pdr_state, sys, model, front_idx) {
        // Blocking failed: real counterexample trace must have
been found
        return false;
      }
```

```
      // Update frontier frame formula after refinement
      front_frame = pdr_state.frame_assumptions(front_idx);
    }

    // Add blank frame to end of frame trace
    pdr_state.add_frame();

    // Try to propagate all clauses in frames
    if propagate_clauses(&mut pdr_state, sys) {
      // All clauses were propagated from one frame to the next:
inductive invariant found
      return true;
    }
  }
}
```

This closely mirrors what we did to prove the property about the `Pipe` circuit, but we'll go over it more concretely here.

5.1. Main loop

This is where we try to rule out property violations in the frontier frame. We saw this in the `Pipe` example when we checked the safety property at $\mathbf{R}_0$, $\mathbf{R}_1$, and $\mathbf{R}_2$. In the $\mathbf{R}_0$ case, the loop body was skipped because there were no property violations in the initial states (i.e. $\mathsf{SAT}?[\mathbf{R}_0 \wedge \neg\mathbf{P}]$ is `UNSAT`). However, $\mathbf{R}_1$ was a different story: we found two property violations and needed to use some reachability checker to prove that they were spurious. `block_cube` is that reachability checker.

5.1.1. Recursive cube blocking

Just as in the `Pipe` example, we need to first test if the cube `model` is actually reachable from the previous frame. Recall that we can use the relative induction query to do this.

However, what if `model` is actually reachable from the previous frame? This is where the "recursive" part comes in: we extract the predecessor model and then try to block this predecessor. This process continues until we actually block this predecessor (thereby possibly blocking the original cube) or reach the initial frame (which means we found a concrete counterexample trace).

For efficiency's sake, we'll not make explicit recursive calls. Instead, we introduce a min-queue containing predecessor cubes that must be blocked for the unreachability proof to go through, ordered by the frame at which they need to be blocked. In Rust, this might look like:

```
/// Wrapper for cube and frame identifier
struct TimedCube {
  cube: Cube,
  frame_id: FrameId,
}

/// Implement comparators for [`TimedCube`] that only compare
[frame_id] in reverse order (such that lower frames are "greater"
than higher frames)
```

```rust
impl PartialEq for TimedCube { ... }
impl Eq for TimedCube { }
impl PartialOrd for TimedCube { ... }
impl Ord for TimedCube { ... }

/// # Returns
/// Whether [cube] was successfully blocked at frame [idx]
fn block_cube(
  pdr_state: &mut Pdr,
  sys: &TransitionSystem,
  cube: Cube,
  idx: FrameId,
) -> bool {
  // Min-queue of (frame to block at, cube to block) pairs
  let mut worklist = BinaryHeap::from([TimedCube { cube,
frame_id: idx }]);

  while let Some(obj) = worklist.pop() {
    if obj.frame_id == FrameId::Init {
      // Reached initial frame: cannot block
      return false;
    } else if rel_ind(pdr_state, sys, &obj.cube,
obj.frame_id.decrement()) {
      // Cube could be blocked from previous frame: enrich
current frame (and all lower, non-init frames) with new clause,
preserving first frame trace fact
      let ids = pdr_state.iter()
                    .take_while(|id| id <= &obj.frame_id)
                    .collect::<Vec<_>>();
      for lower_idx in ids {
        pdr_state.add_clause(obj.cube.negate(), lower_idx);
      }
    } else {
      // Cube could not be blocked from previous frame:
recursively try to block predecessor AND original cube
      let model = get_model(&sys.states);
      worklist.push(TimedCube {
        cube: model,
        frame_id: obj.frame_id.decrement(),
      });
      worklist.push(obj);
    }
  }

  // All cubes were blocked
  true
}
```

One possibly confusing part of this implementation is the `else` branch. At that point, we know that we can't block `cube` from the previous frame, so we push a predecessor onto the min-queue (this is the recursive call). However, we also push

the original cube as well. This is because we technically haven't blocked `cube` yet; this is dependent on the fact that we can block the predecessor `model`.

Another thing to note: the initial frame check (`obj.frame_id == FrameId::Init`) is **not** the base case $\mathbf{I} \Rightarrow \neg c$ of relative induction. Instead, it detects a real counterexample: a predecessor was found in $\mathbf{R}_0$, so the chain of cubes in the min-queue traces back to an initial state. As we saw earlier, the relative induction query only accounts for the "inductive" case. Vanilla PDR gets away without an explicit base case check because every cube is a full state: the first iteration of the main loop ensures that no initial state violates the property, and if an initial state were ever a cube to block at frame $k \geq 1$, it would reach a property violation in fewer steps than the frontier index. That shorter counterexample would have already been found in an earlier iteration. However, once cubes are generalized (see Section 7), a cube may intersect the initial states, so you must add an explicit $\mathsf{SAT}?[\mathbf{I} \wedge c]$ check before blocking.

5.2. Clause propagation

After we exit the Main Loop, we know that the frontier frame must not have any property violations. Therefore, we can add a new blank frame without violating the third fact of the frame trace. As in the `Pipe` example, we then try to move all frame clauses forward. Once again, we will use the relative induction query to do this. However, instead of testing if a cube to block at a frame can be reached from the previous frame, we take a cube already blocked at a frame and check whether it can be reached in one step from that same frame (i.e. whether or not its clause will also hold in the **next** frame).

In code, this would look like:

```
/// # Returns
/// Whether inductive invariant was found
fn propagate_clauses(pdr_state: &mut Pdr, sys: &TransitionSystem)
-> bool {
  // Get identifiers for all non-init, non-frontier frames
  let front = pdr_state.frontier();
  let ids = pdr_state.iter()
                     .take_while(|id| id < &front)
                     .collect::<Vec<_>>();

  // Iterate through all non-init, non-frontier frames
  for idx in ids {
    // Number of clauses propagated to next frame
    let mut num_prop = 0usize;

    // Iterate over all clauses in the frame
    for clause_idx in 0..pdr_state[idx].clauses.len() {
      let clause = pdr_state[idx].clauses[clause_idx].clone();

      if rel_ind(pdr_state, sys, &clause.negate(), idx) {
        // Clause holds in next frame: add to next frame
        pdr_state.add_clause(clause, idx.increment());
        num_prop += 1;
      }
    }
```

```
    // If all clauses were propagated, inductive invariant was
found
    if num_prop == pdr_state[idx].clauses.len() {
      return true;
    }
  }

  // No inductive invariant found
  false
}
```

Notice that we never iterate over the initial frame. This is because it's special: it only describes the initial reset states, not excluded spurious states, so we don't need to move them forward through the frame trace.

In addition, notice that we claim that an inductive invariant is found if all clauses were propagated from one frame to the next. This is because we can assert that $\mathbf{R}_i = \mathbf{R}_{i+1}$ if all clauses were propagated from $\mathbf{R}_i$ to $\mathbf{R}_{i+1}$ (since $\mathbf{R}_{i+1}$ must have a subset of clauses in $\mathbf{R}_i$ by the first fact of the frame trace). Then, notice that by the facts of the frame trace, we get

$$\mathbf{R}_0 \Rightarrow \mathbf{R}_i$$
$$\mathbf{I} \Rightarrow \mathbf{R}_i \quad (\text{by definition})$$

$$\mathbf{R}_i \wedge \mathbf{T} \Rightarrow \mathbf{R}'_{i+1}$$
$$\mathbf{R}_i \wedge \mathbf{T} \Rightarrow \mathbf{R}'_i \quad (\text{since } \mathbf{R}_i = \mathbf{R}_{i+1})$$

$$\mathbf{R}_i \Rightarrow \mathbf{P}$$

which implies that $\mathbf{R}_i$ is an inductive invariant that satisfies the property $\mathbf{P}$.

And we're done! That's PDR!

6. Implementation tips and tricks

Here are some other things to be aware of when writing PDR:

1. Many solvers can emit `UNKNOWN` query results when they can't decide the satisfiability of a formula; make sure to handle these cases to prevent unsoundness (e.g. accepting `UNKNOWN` as `UNSAT` in a relative induction query).
2. The format to represent Boolean formulas here is only a suggestion; use the method that fits best in your workflow (many solver libraries provide ways to construct Boolean formulas!).
3. To get a counterexample trace in cases of a real property violation, you can either run another simpler model-checking algorithm (such as BMC) for the trace length (the simplest way), or track the relationships between the cubes-to-block in the `block_cube` min-queue (more robust, but needs more machinery).
4. Avoid pushing/popping contexts in the solver (since this can cause internal lemmas that the solver learns to be discarded). One solution to avoid this is by using `(check-sat-assuming)` queries and asserting *activation literals* that are handles to the formulas you're using. For example, instead of asserting $a = 0 \wedge b = 1 \wedge c = 0$ directly in the solver (polluting the context), assert act $\Rightarrow a = 0 \wedge$

$b = 1 \wedge c = 0$, where act is an activation literal. When you want to assert the formula, pass act as an assumption to the `(check-sat-assuming)` query. Else, pass $\neg$ act (this automatically resolves the implication and prevents the solver from exploring state space it shouldn't).

5. Once you add clause generalization (see Section 7), PDR often finds stronger clauses to add to frames (in vanilla PDR, every clause blocks exactly one full state, so no clause can subsume another). However, it never cleans up weaker clauses that are *subsumed* by the stronger clauses. This causes a lot of frame bloat that eventually slows down the clause propagation step. Therefore, when adding new clauses to the frame trace, always remove weaker, subsumed clauses. A cheap check to test for this is to see if the new clause has a subset of the literals of the old clause.
6. Notice that clauses are often repeated between frames. To reduce memory usage, only store clauses at the *highest* frame at which they hold. This requires changing `frame_assumptions` to account for all clauses stored at frames `idx` and above.

7. Optimizing PDR

While our vanilla PDR works for very simple hardware, it would start to struggle on even slightly non-trivial hardware. This is why a significant portion of PDR papers are focused on adding more optimizations to PDR. We'll discuss some of them at a high-level here.

One of the biggest problems in vanilla PDR is that adding a negated cube is a very weak fact. Since the solver model gives an assignment to all state in the hardware, the negated cube just excludes one state out of possibly millions or more states. The problem here is that some of the state assignments may have nothing to do with why the cube leads to a property violation. Therefore, by placing these artificial constraints in our clause, we lose the opportunity to exclude similar spurious states. This leads to PDR having to block many of these states, which takes many expensive solver calls, eventually stalling the algorithm.

The solution to this is **clause generalization**. Namely, we can increase the size of the state space excluded by the negated cube while preserving the original relative inductiveness. Some common methods for doing this are UNSAT cores and literal dropping. However, in some cases, the resulting cube can be so minimized that it intersects with the initial states, leading to unsoundness. We elide the discussion of common solutions for this problem here, but it's an important problem to keep at the back of your mind.

In a similar vein, we can also generalize **counterexamples-to-induction**, or predecessor cubes that must be blocked for the proof to go through (in our implementation, these are the cubes that end up in the min-queue in `block_cube`). Here, we recognize that some state variables are irrelevant to the failure of the relative induction query, so we remove them from the cube and instead focus on the circuit states/inputs that directly cause the issue. This analysis can be done through ternary simulation, cone-of-influence analysis, among many other methods.

In the end, functionality is most important. I recommend trying the simplest optimizations first before moving on to more complicated ones. If a simple optimization is good enough for your applications, that's great! If you need more proving power, spend more time implementing a more advanced optimization.

8. Testing PDR

A model-checking algorithm is useless if it's unsound. To avoid these problems, proper testing is key.

For Patronus, I hand-wrote a series of safe and unsafe Verilog designs, compiled them into BTOR2 with `yosys`, and ran the algorithm to check for baseline correctness. Next, I tested the algorithm with the HWMCC '25 benchmarks and ensured that the results produced by my implementation matched with those produced by Stanford Pono (at the time of writing, a state-of-the-art model checker). Since the HWMCC '25 benchmarks are created from industry problems, they are a more robust test of the implementation.

However, the HWMCC '25 benchmarks are more complex and can run for a long time. For this reason, I usually run my hand-written Verilog tests as a basic regression test after I make changes (these tests can also be added to a CI pipeline for more automation). Then, I only run the HWMCC '25 benchmark tests weekly (or nightly, if you have the available compute) to catch any other regressions and track the performance of the algorithm after optimizations.

9. Acknowledgements

This blog post would not be possible without the mentorship of Kevin Laeufer and Adrian Sampson. Their expertise and feedback have not only been invaluable in my work on Patronus, but also my growth as a student researcher. I also want to extend my thanks to Ernest Ng and Ayaka Yorihiro for their extensive and helpful feedback on this post.

Bibliography

[1] Aaron R. Bradley. 2011. SAT-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, 2011. Springer-Verlag, Austin, TX, USA, 70–87.

[2] Niklas Een, Alan Mishchenko, and Robert Brayton. 2011. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD '11)*, 2011. FMCAD Inc, Austin, Texas, 125–134.