PURSE: Property Ordering Using Runtime Statistics for Efficient Multi-Property Verification

Sourav Das[1], Aritra Hazra[1], Pallab Dasgupta[2], Sudipta Kundu[2], Himanshu Jain[2]
[1]Indian Institute of Technology Kharagpur, Kharagpur, India, [2]Synopsys Inc., USA
{s_das, aritrah}@cse.iitkgp.ac.in, {pallabd, sudiptak, hjain}@synopsys.com

***Abstract*—Multi-property verification has emerged as a contemporary challenge in the chip design industry. With designs now encompassing hundreds of properties, conventional sequential verification without information sharing is no longer preferred. Past attempts towards grouping or ordering properties based on cone-of-influence (COI) are typically ineffective for complex designs. This paper introduces PURSE, a novel approach that addresses this challenge by dynamically reordering properties for sequential and incremental solving. By identifying and prioritizing simpler properties, the process accelerates convergence. This article presents two dynamic reordering techniques guided by statistical data gathered from the IC3/Property Directed Reachability (PDR) proof engine. The study compares dynamic ordering strategies against static ordering and a default ordering based on design structure. Empirical results from various industrial designs demonstrate that our proposed methodology performs better in most cases, with up to 25% improvements in convergence.**

***Index Terms*—Property Directed Reachability, Multi-property Verification, Dynamic Property Ordering, Statistical Analysis**

I. Introduction

Verification encompasses simulation-based testing [1], [2] and formal methods [3]. While simulation is effective for functional testing, it might miss critical corner cases. Conversely, formal methods offer exhaustive verification support but often demand significant computational resources and time, especially for "hard-solving properties" involving intricate temporally deep operations. Typically, verification involves a multitude of formal properties. The effort required for solving each property is unknown a priori and can widely vary compared to another property. Handling multiple hard properties complicates the verification flow and time/resource deployment, often leading to unpredictable delays.

Multi-property verification has, therefore, emerged as a contemporary challenge in the chip design industry, and there is no organized way to do it. Researchers have tried different approaches to solve this problem, such as proving a property while assuming the other properties to be true [4], grouping properties [5]–[7], or organizing them in a certain order [8], [9] based on the structure such as textual overlap or their relative cone-of-influence (COI). However, there are notable limitations in these approaches. In the former methodology [4], the authors assume all other properties to be true while proving an assert property. However, for an assert property proven with assume guarantees, the assert statement must be validated within all paths where the assumption holds. Situations may arise where assertion fails before the assumption does, resulting in false counterexamples. The effectiveness of ordering and grouping properties based on the structure of the design is limited by the fact that they are static decisions. Moreover, some of the naïve algorithms for analyzing design structure require quadratic comparisons, which do not scale for practical designs. Although near-linear time algorithms [5], [6] exist for grouping properties, computing COI for large designs with many properties is still cumbersome, and not a suitable differentiator as often the COI includes all design space variables, or the COI sizes are almost identical. In [5], [6], the authors addressed multi-property verification through concurrent verification of high-affinity groups where each group can be verified in parallel, and in [7], the authors explored property grouping in high-level languages like System Verilog. However, our approach emphasizes the aspect of property ordering. Some prior works [8], [9] use property ordering, but these are based on the size of COI, which denotes the number of latches present in the respective COI. Such strategies are very naïve and fail to scale up in practice. For example, the COI sizes for 8 properties, $\langle 0, 1, 10, 11, 16, 17, 18, 19\rangle$, in one of our designs are $\langle 201, 201, 195, 207, 199, 206, 199, 195\rangle$. Here, Property 11 has the largest COI and properties, 10 and 19, have the smallest COI. On the other hand, properties $\langle 10, 11, 17, 19\rangle$ were proven easily, and $\langle 0, 1, 16, 18\rangle$ turned out to be harder, with Property 0 being the hardest.

In this paper, we present PURSE, the first tool that dynamically reorders the properties on the fly by looking at the runtime statistics obtained from the model checker. Given a design and a structurally ordered set of properties at the bit level, we identify the key features of a property by statistically analyzing the data dumps from the IC3/PDR [10], [11] algorithm. We use IC3/PDR as the baseline proof engine as it is a very scalable and popular hardware model-checking engine. Throughout the paper, we use PDR as the short form for IC3/PDR. By nature, PDR is sensitive to property ordering, as it can inherently use the already proven properties to solve the other properties more effectively. The significant contribution of this article lies in identifying the easier properties and solving them faster by dynamically reordering them and then using them to solve the other properties. The key contributions of the paper are:

- Identifying relevant computable features by analyzing the statistical data dumps of the PDR algorithm across many

The authors acknowledge Synopsys Inc. for supporting this work.

979-8-3503-4859-0/DATE24/© 2024 EDAA

properties
- Devising two justifiable metrics for dynamically reordering the properties without imposing significant overhead on the total runtime of the process
- Experimental results with different baseline ordering strategies and demonstrating the efficacy of dynamic ordering on real industrial test cases.

The rest of the paper is organized as follows. Section II outlines the PDR model-checking algorithm and the different terminologies. Section III presents the proposed methodology. Section IV discusses the results of the property ordering algorithm in industrial designs and compares it with other baseline ordering strategies. Section V discusses conclusions.

II. Background and Preliminaries

This section presents the formal modelling of the problem, provides a brief overview of the PDR algorithm, and describes terminologies relevant to the PDR algorithm. We also outline two existing multi-property verification approaches using PDR.

A. Formal Modelling

A Finite State Machine (FSM), $\mathcal{M} = \langle \mathcal{I}, \mathcal{O}, \mathcal{S}, S_0, \mathcal{T} \rangle$, is a tuple where $\mathcal{I}$ is the set of primary inputs, $\mathcal{O}$ is the set of primary outputs, $\mathcal{S}$ is the set of internal state variables, and $S_0 \subseteq \mathcal{S}$ is the set of initial states. $\mathcal{T}$ is the transition relation, $\mathcal{T} \subseteq \mathcal{I} \times \mathcal{S} \times \mathcal{S}$, that is, $(i, s, s') \in \mathcal{T}$ iff there is a transition from state s to s' on input i.

A state of the system is an assignment of Boolean values to all the variables in $\mathcal{S}$ and is described as a *cube*, that is, a conjunction of *literals*, where each literal is a state variable or its negation. The negation of a cube is called a *clause*, which is a disjunction of literals.

Given a FSM, $\mathcal{M}$, and a set of properties, $\{\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_N\}$, the model checking tasks are to find whether $\mathcal{M} \models \mathtt{P}_1$, $\mathcal{M} \models \mathtt{P}_2, \cdots$, and $\mathcal{M} \models \mathtt{P}_N$. Here $\mathcal{M} \models \mathtt{P}_i$ when the property $\mathtt{P}_i$ is proven on $\mathcal{M}$. If not, then there exists a sequence of states from the initial state to the state falsifying the property, called the counter-example (CEX) trace of the property. Formal verification *converges* on a property when it proves it or reports a CEX.

B. Property Directed Reachability (PDR)

The PDR algorithm tries to solve a property by incrementally generating inductive clauses relative to stepwise approximation. The algorithm always performs a one-step inductive check for a property or its derivatives, which is scalable as compared to other model checking algorithms [12], [13] that unroll the transition system, thereby choking the SAT solver on large designs. PDR checks the property by gradually refining it and eventually producing an inductive strengthening or a counter-example trace. Algorithm 1 outlines the vanilla version of the PDR algorithm for single property verification with minor modifications (highlighted in blue) for introducing timeouts. Here $\mathtt{CB}(cb)$ and $\mathtt{PB}(pb)$ represent the conflict and propagation clause budgets, respectively. It

Algorithm 1 PDR for Single Property Verification

Input: $\mathcal{M}$, P, CB, PB
Output: SAT, UNSAT, UNDECIDEABLE

```
1:  F ← {F_0}, k ← 0                      ▷ F: PDR Trace, k : PDR depth
2:  cb ← 0, pb ← 0                        ▷ Current clause budgets
3:  while true do
4:      F, cex ← REC_BLOCK_CUBE(M, P, k)
5:      cb, pb ← Update current clause budgets
6:      if cex ≠ φ then
7:          return SAT                    ▷ Found a real CEX
8:      else if cb > CB || pb > PB then
9:          return UNDECIDEABLE
10:     else
11:         k ← k + 1
12:         F ← F ∪ {F_{k+1}}             ▷ Open a new frame
13:         F ← PROPAGATE_BLOCKED_CUBES(M, F)
14:         if F reaches a fixed point then
15:             return UNSAT
```

is assumed that the reader is familiar with the PDR algorithm. The details of the procedures REC_BLOCK_CUBE and PROPAGATE_BLOCKED_CUBES can be found in [11].

In short, the execution of PDR algorithm involves finding a bad cube (s) in the last frame and invoking REC_BLOCK_CUBE to block the cube recursively. Once blocked, the cube is generalized (its size is reduced by eliminating literals) before adding it to the frame. During the generalization phase, a new cube g is derived from s, which can subsume multiple older cubes. So, if a property is hard and performs poorer generalization, it needs to block more cubes, increasing the SAT solver calls. Moreover, properties performing poorer generalization have larger literals count and average frame clause size. Finally, when a property is proven in the last frame, the PROPAGATE_BLOCKED_CUBES method pushes cubes forward while performing subsumption.

C. Terminologies Associated with PDR

From the description of the PDR algorithm provided above, we figured that a formal characterization of the following terminologies would be relevant to explain our proposed ordering approach, which will follow in the subsequent section.

Definition 1: **[Frames ($\mathcal{F}$)]** The PDR algorithm maintains a set of frames $\mathcal{F} = \{\mathtt{F}_0, \mathtt{F}_1, \ldots, \mathtt{F}_k\}$, where $\mathtt{F}_0 = S_0$, and each $\mathtt{F}_i$ $(0 < i \leq k)$ is an over-approximation of the set of states that are reachable in $1, 2, \ldots, \mathtt{k}$ steps. The frames are maintained in such a way that each $\mathtt{F}_k$ is contained within $\mathtt{F}_{k+1}$; in other words, the relation is syntactic: that is, $clauses(\mathtt{F}_{k+1}) \subseteq clauses(\mathtt{F}_k)$.

Definition 2: **[Proof Obligations** $(\mathtt{s}_j, \mathtt{k}_j)$**]** Along with the set of frames, $\mathcal{F}$, PDR maintains a queue of proof-obligations, $\mathcal{Q} = \{(\mathtt{s}_1, \mathtt{k}_1), (\mathtt{s}_2, \mathtt{k}_2) \cdots (\mathtt{s}_i, \mathtt{k}_i)\}$, where $(\mathtt{s}_j, \mathtt{k}_j)$ indicates that the state (cube) $\mathtt{s}_j$ in frame $\mathtt{k}_j$ needs to be blocked. We also use $|\mathtt{PO}|$ to denote the total number of proof obligations created.

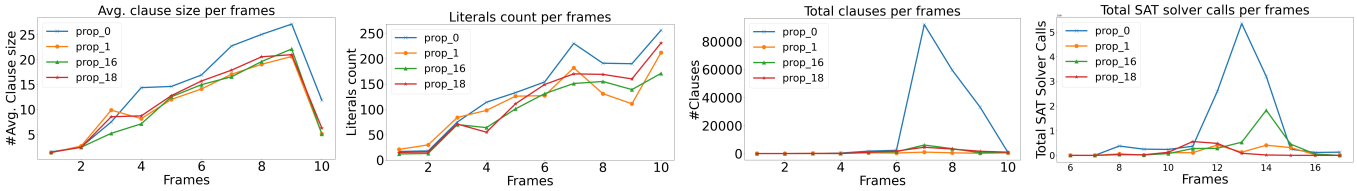

Fig. 1: Change of Avg. Clause Size, Literals Count, Total Clauses, and SAT Solver Calls per Frame for Different Properties

Definition 3: **[Frame Clauses (ϕ)]** In PDR, each frame $\mathtt{F_k}$ is a set of clauses. The algorithm checks whether a property P is inductively true, that is, whether $(\mathtt{P} \wedge \mathcal{T}) \rightarrow \mathtt{P'}$, where $\mathtt{P'}$ denotes P in the next state. If this check fails, then PDR finds a bad state s and refines the check to $(\mathtt{P} \wedge \neg \mathtt{s} \wedge \mathcal{T}) \rightarrow \mathtt{P'}$, and separately aims to prove the unreachability of s. In other words P gets progressively modified to $(\mathtt{P} \wedge \phi)$, where $\phi = (\neg \mathtt{s_1} \cdots \wedge \neg \mathtt{s_k})$. Here, $(\mathtt{P} \wedge \phi)$ represents a frame $\mathtt{F_k}$, ϕ denotes the frame clauses, and $|\phi|$ denotes the number of frame clauses.

Definition 4: **[Literals Count (TL)]** The number of literals in a clause $\mathtt{c_i}$ is denoted by $|\mathtt{c_i}|$. The total literal count is the sum of all the literals (each literal counted once) present in the frame clauses. For example, if there are three frame clauses, $[(\mathtt{x_1} \vee \mathtt{x_3}), (\mathtt{x_2} \vee \neg \mathtt{x_3} \vee \mathtt{x_1}), (\neg \mathtt{x_1} \vee \mathtt{x_2} \vee \neg \mathtt{x_3})]$, then $\mathtt{TL} = 5$.

Definition 5: **[Average Literals per Frame Clause (AL)]** We define AL as the sum of the number of literals across all the frame clauses divided by the number of frame clauses. For example, if there are three frame clauses, $[(\mathtt{x_2} \vee \neg \mathtt{x_3} \vee \mathtt{x_1}), (\neg \mathtt{x_1} \vee \mathtt{x_2} \vee \neg \mathtt{x_3}), (\mathtt{x_1} \vee \mathtt{x_3})]$, then $\mathtt{AL} = (3 + 3 + 2)/3 = 2.6$

Definition 6: **[SAT Solver Calls (TC)]** The PDR algorithm makes numerous calls to the SAT solver. We define TC as the number of calls the PDR algorithm makes to the SAT solver for solving a property.

Figure 1 shows how the above metrics change over frames for four different properties. Higher values of these metrics, such as for Property *prop_0*, typically indicate that the property is harder to prove.

D. Multi-property Verification using PDR

Multi-property verification is commonly performed sequentially, using a portfolio-based approach using multiple model-checking algorithms. PDR is one of the most scalable model-checking approaches. Typically PDR is applied in two ways, namely *depth first* or *breadth first*.

- **Depth First PDR (DF-PDR):** DF-PDR tries to solve properties sequentially, one property at a time. The traditional PDR engine for a single property (Algorithm 1) is sequentially invoked N times separately for the N properties. The ordering is important, as invariants discovered in each proof are carried over to the next proof.
- **Breadth First PDR (BF-PDR):** In this approach, all the properties are processed, frame by frame. This is how it is implemented in the ABC [14] tool. Since the PDR algorithm maintains a set of frames for each property, BF-PDR explores Frame 1 for all properties, followed by Frame 2, until we reach a fix-point where $\mathtt{F_k} = \mathtt{F_{k+1}}$.

For large designs, taking all the properties together typically leads to poor design abstraction, and therefore, it makes sense to work on individual properties, one at a time. The major drawback of the DF-PDR approach is that it may get stuck on a hard goal, thereby failing to make progress on easier goals. It is, therefore, important to order the properties so that easier properties are not affected by the harder goals. This is the focus of this paper.

Notably, regardless of the goal order, BF-PDR will fail to make progress if there is a mixed bag of hard and easy goals. This justifies our focus on DF-PDR, specifically trying to improve its efficacy by finding the suitable property ordering.

III. Proposed Methodology

The main challenge in finding an ordering of the goals for DF-PDR, that ensures that easier goals are not affected by the harder ones is that the structure of the property and the design do not reveal the hardness of the properties. On the other hand, as PDR starts working on the properties, it creates footprints that contain relevant information to infer the hardness of the goals. In order to take advantage of these footprints, we start solving properties sequentially using the traditional PDR algorithm with low clause budgets. Although the clause budgets are low, they are not static across designs and depend on the design size. This ensures two things: 1) we are not deciding too early about the goal order; 2) the algorithm never gets stuck on a single goal. The statistical footprints dumped during this phase are:

$$\langle \mathtt{GOAL\_ID}, |\mathtt{PO}|, |\phi|, \mathtt{TL}, \mathtt{AL}, \mathtt{TC}, |\mathcal{F}|, \mathtt{T} \rangle$$

where GOAL_ID represents the property id, T represents the time (in secs) for which the goal was run in the given iteration, and the rest of the members are defined in Section II-C. Table I shows the data extracted from PDR after the first iteration. In the proposed framework, such footprints are used to create a new goal order for the next round. Additionally, all goals proven till the current iteration are used as assumptions (helper properties) in all future iterations. The process continues till all the properties are solved, or a specified timeout is reached. The following subsection presents the property ordering heuristics based on the footprints of the previous round.

A. PURSE Framework

Given a design ($\mathcal{M}$) in the bit level and a pre-defined goal order ($\mathcal{L}_\mathcal{P}$) for multiple properties, the algorithm tries to

TABLE I: Footprint of Round-1 of PDR for Design $D1$

Goal_ID	$\lvert$PO$\rvert$	$\lvert\phi\rvert$	TL	AL	TC	$\lvert\mathcal{F}\rvert$	T
0	2717	466	2189	4.70	7295	13	5
1	2328	388	1586	4.10	6477	13	4
16	915	235	841	3.58	2767	9	2
18	1126	223	804	3.61	3113	13	2

statistically find a better ordering by reordering the unsolved goals. The algorithm also receives a maximum time bound $\mathscr{T}$, and two clause budgets cb_val and pb_val. Initially, the algorithm starts with a low budget but increases exponentially if no properties are solved in a given iteration. At the end, the algorithm returns a list indicating the status of the properties. Algorithm 2 outlines the property ordering algorithm, and the steps are described below.

Initialization: Initialize conflict clause budget (CB) and propagation clause budget (PB) to low values (Lines 4 $\cdots$ 10 of Algorithm 2). These starting values may be chosen based on design size because choosing too small initial clause budgets for large designs will result in early and, thereby, premature choice of ordering. Likewise, choosing a large initial clause budget will marginalise the effect of ordering in small circuits.

Sequential Runs With Goal Ordering: Following the initial clause budget setup, the goals are attempted sequentially. If the result of the vanilla PDR is SAT (refuted) or UNSAT (proven), the Property_Status list is updated to 0 or 1 accordingly. However, if the result is UNDECIDEABLE, the unk_goals list is updated. The current property status and related statistics are also recorded for later analysis (Line 30 of Algorithm 2). Finally, the statistics of the unsolved goals are extracted and ordered based on certain criteria (Lines 38 and 39 of Algorithm 2). We use two heuristics for ordering the unsolved goals, namely:

[Variant-1] Typically, hard properties create a very large number of proof obligations in PDR, resulting in a large number of calls to the SAT solver. Therefore, our first metric is the average number of calls to the SAT solver per frame. The inconclusive goals are sorted in ascending order of $(\mathtt{TC}/\lvert\mathcal{F}\rvert)$. Following the example from Table I, the $(\mathtt{TC}/\lvert\mathcal{F}\rvert)$ values of $\langle 0, 1, 16, 18\rangle$ are $\langle 561.15, 495.92, 307.44, 239.46\rangle$ respectively. So, the goal order for the next iteration is $\langle 18, 16, 1, 0\rangle$.

[Variant-2] An important metric for progress in PDR is the number of frames explored. In the same resource budget, PDR will typically explore more frames for easy properties than for hard properties. In our second metric, the goals are sorted based on the descending order of frames explored, that is, $\lvert\mathcal{F}\rvert$. Properties having the same $\lvert\mathcal{F}\rvert$ are then sorted in ascending order of a combined score that adds up the cost components, TC, $\lvert\phi\rvert$, $\lvert$PO$\rvert$, TL, AL, after normalization.

The combined normalized score is computed in two steps. The *first step* normalizes each metric, namely TC, $\lvert\phi\rvert$, $\lvert$PO$\rvert$, TL, AL, by dividing it by its maximum value. For example, the normalized score for $\lvert$PO$\rvert$ of property i is calculated as:

$$\mathtt{NS}(\mathtt{PO_i}) = \lvert\mathtt{PO_i}\rvert / \mathtt{Max}\big(\lvert\mathtt{PO_i}\rvert\big)_{i=1}^{i=N}$$

Algorithm 2 PURSE for Multi-property Verification

Input: $\mathcal{M}$, $\mathcal{L}_\mathcal{P}$, $\mathscr{T}$, cb_val, pb_val
Output: Property_Status

1: Property_Status$[\lvert\mathcal{L}_\mathtt{P}\rvert] = \{-1, \ldots, -1\}$
2: gates $\leftarrow$ **#SOME VALUE** ▷ Designer's Choice
3: size $\leftarrow$ **GET #AND GATES OF DESIGN**
4: **if** size $<$ gates **then**
5: CB $\leftarrow$ cb_val, PB $\leftarrow$ pb_val
6: **else if** size $<$ gates $\times$ 10 **then**
7: CB $\leftarrow$ cb_val $\times$ ((size%gates) + 1)
8: PB $\leftarrow$ pb_val $\times$ ((size%gates) + 1)
9: **else**
10: CB $\leftarrow$ cb_val $\times$ 11, PB $\leftarrow$ pb_val $\times$ 11
11: j $\leftarrow$ 0
12: **while** *true* **do** ▷ Incremental Loop
13: solved $\leftarrow$ 0
14: unk_goals $\leftarrow$ [ ]
15: **for** each $\mathcal{P}_i \in \mathcal{L}_\mathcal{P}$ **do** ▷ Sequential Loop
16: $start\_time \leftarrow clock()$
17: **if** j $\neq$ 0 **then**
18: $\mathcal{P}_i \leftarrow \mathcal{P}_i^{j-1}$ ▷ Restore Property Status
19: $val \leftarrow$ PDR$(\mathcal{M}, \mathcal{P}_\mathtt{i}, \mathtt{CB}, \mathtt{PB})$ ▷ refer Algo. 1
20: **if** $val ==$ SAT **then** ▷ Real CEX found
21: Property_Status$[\mathcal{P}_\mathtt{i}] \leftarrow 0$
22: solved $\leftarrow$ solved + 1
23: **else if** $val ==$ UNSAT **then** ▷ Property Proved
24: Property_Status$[\mathcal{P}_\mathtt{i}] \leftarrow 1$
25: solved $\leftarrow$ solved + 1
26: **else if** $val ==$ UNDECIDEABLE **then**
27: unk_goals $\leftarrow$ unk_goals $\cup\, \mathcal{P}_i$
28: **else**
29: **return** -1
30: **Save current status**
31: $end\_time \leftarrow clock()$
32: T $\leftarrow$ T + $(end\_time - start\_time)$
33: **if** T $> \mathscr{T}$ **then**
34: **break**
35: $\mathcal{L}_\mathcal{P} \leftarrow$ unk_goals
36: j $\leftarrow$ j + 1
37: **if** (T $< \mathscr{T}$) $\wedge$ ($\lvert\mathcal{L}_\mathcal{P}\rvert \neq 0$) **then**
38: **Extract relevant statistics**
39: **Order unsolved goals** ▷ Two variants
40: **else**
41: **break**
42: **if** solved $== 0$ **then** ▷ Increase clause budgets
43: CB $\leftarrow$ CB $\times\, mf$ ▷ mf: Multiplicative Factor
44: PB $\leftarrow$ PB $\times\, mf$
45: **return** Property_Status

TABLE II: Comparative Study of Different Property Ordering Techniques with the Baseline over Various Industrial Designs

Design	#Prop	#POI	Default (Static)[1]	PURSEv1 (Static)[2]	PURSEv2 (Static)[3]	PURSEv1 (Dynamic)[4]	PURSEv2 (Dynamic)[5]	[4] vs [1] (% Gain)	[5] vs [1] (% Gain)
D1	21	8	37276	26913	28528	32139	51238	13.78	-37.46
D2	1941	676	71099591	73264162	72819275	72496586	72180243	-1.96	-1.52
D3	8641	2873	17044150	12865153	14922747	15823496	14730866	7.16	13.57
D4	6574	83	6271336	6269116	6264583	6274929	6266387	-0.06	0.08
D5	313	140	16608615	16730077	16631718	16684047	16586187	-0.45	0.14
D6	243	73	13189030	12398094	12606198	12102366	11955580	8.24	9.35
D7	250	250	46554837	46625961	47608075	46022850	46367765	1.14	0.40
D8	1025	294	11228810	11235879	10927121	11030329	10935091	1.77	2.62
D9	14	14	479518	424719	395706	359841	385870	24.96	19.53
D10	427	41	1971213	1949918	1982767	1931754	1817641	2	7.79
D11	10	5	33644	28440	27339	26187	25658	22.16	23.74
D12	591	463	71193600	73267200	70156800	72576000	64972800	-1.94	8.74
D13	172	172	13647823	13267296	13217101	13373447	12939278	2.01	5.19
D14	459	458	6005515	6388454	5923007	6255418	6001638	-4.16	0.06
D15	1602	643	45273600	45619200	45273600	45619200	45273600	-0.76	0
D16	908	75	283628	294355	287077	253743	255163	10.54	10.04
D17	1283	1126	49932634	52305889	43504405	50853099	47658667	-1.84	4.55
D18	1032	969	164848603	168404074	170415353	166824773	164341158	-1.2	0.31
D19	3976	3976	382762329	421209950	328663526	401248581	373129770	-4.83	2.52
D20	182	177	27132583	28161905	28180261	27782135	27769577	-2.39	-2.35
D21	3005	1262	176332538	177726277	185026669	182146484	177301564	-3.30	-0.55
D22	1163	985	226517231	228649807	228205789	226445347	226310441	0.03	0.09
D23	168	168	63817	57641	65005	56155	57872	12.01	9.32
D24	298	298	13397164	12765750	13875152	12102308	12511252	9.67	6.61
D25	247	73	16915034	16546478	14811855	15420542	14900776	8.84	11.91
D26	3781	958	76992831	71216976	85441974	81181641	75683898	-5.44	1.70
D27	2937	500	32670999	32788726	33146598	32617611	33314505	0.16	-1.97
D28	1941	676	60821240	61672375	63447643	60878542	59434436	-0.09	2.28
D29	682	682	8575152	7108371	7904161	8230457	7637232	4.02	10.92

where $\mathtt{N}$ is the current number of unsolved properties. From Table I, the maximum value of $|\mathtt{PO}|$ for the unsolved goals is 2717 and the normalized values of $\mathtt{NS(PO)}$ for the unsolved goals are $\langle 1, 0.86, 0.34, 0.41\rangle$, respectively. Similarly, the normalized score for $|\phi|, \mathtt{TL}, \mathtt{AL}, \mathtt{TC}$ for goal id 0 are all 1's. The *second step* computes the average normalized score by taking the average of all the normalized values computed above.

$$\begin{aligned}\mathtt{AvgNS_i} = (\mathtt{NS}(|\mathtt{PO_i}|) + \mathtt{NS}(|\phi_i|) + \mathtt{NS}(\mathtt{TC_i}) + \\ = \mathtt{NS}(\mathtt{TL_i}) + \mathtt{NS}(\mathtt{AL_i}))/5\end{aligned}$$

For instance, from Table I, the average normalized score for all the goals is $\langle 1, 0.835, 0.47, 0.49\rangle$ respectively. Since the goals are first sorted based on the descending order of the frames, goal id 16 comes in last. However, goals $\langle 0, 1, 18\rangle$ share the same frame depth and are sorted based on their $\mathtt{AvgNS}$. Consequently, the final goal order for the next round is $\langle 18, 1, 0, 16\rangle$.

Incremental Updates: The PDR algorithm is, by default, incremental. However, in this paper, the term incremental refers to two aspects: (i) properties are solved incrementally, that is, we use the already solved properties as assume properties to solve the unsolved ones, and (ii) the clause budgets are updated incrementally to provide more resources to the harder properties. Moreover, the clause budget is doubled if the number of goals remains unchanged in a given iteration (Lines 42 to 44 of Algorithm 2). The process continues as long as there are unsolved goals and the total time limit is not exceeded.

IV. Experimental Results

The experiments were performed on all industrial test cases on a 8-core, 16 × 2800 MHz AMD EPYC processor having 236 GB RAM. Table II offers a comprehensive evaluation of the PURSE ordering strategies, including the default ordering (based on design structure), and presents a detailed analysis across all 29 designs. We also compared the ordering algorithms with randomly chosen and COI-based orderings and demonstrated empirically that dynamic property ordering works better.

In Table II, Column 1 denotes design tags, Column 2 represents the total number of properties in the design, and Column 3 represents the total number of properties of interest (POI). The properties of interest are those left after removing properties of vacuity, witness, covers and liveness from the total property set. Columns 4 to 8 represent a score for each ordering criterion based on the number of properties it solved in a maximum time bound of 48 hours. The score is calculated by summing the runtime of solved goals and adding twice the total runtime multiplied by the total number of unsolved goals.

For example, suppose we have four goals and 2 of them get solved in 120 secs and 1000 secs, respectively, and the remaining two do not get solved in an hour (3600 secs). Then the score is calculated as $(120{+}1000{+}(2{\times}3600{\times}2))$, resulting in a total score of 15520. The score here is calculated after

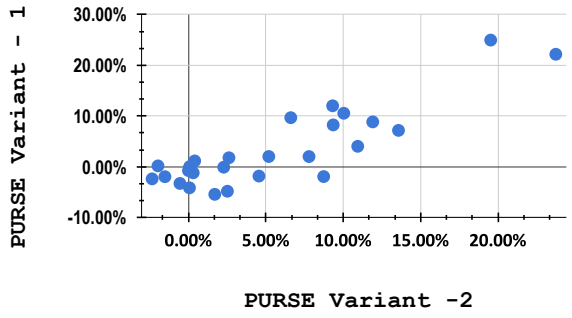

Fig. 2: Comparison of PURSE Variants with Default Ordering

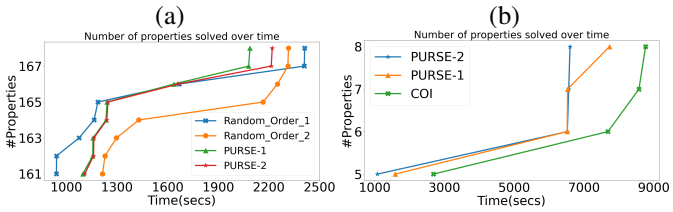

Fig. 3: Comparison of PURSE with (a) Random Ordering ($D23$) and (b) COI based Ordering ($D1$)

running the designs for 48 hours. We use such a notion of score to penalize the algorithm for unsolved goals and lost computation effort.

Columns 9 and 10 compare the percentage gain of the two ordering variants of PURSE with the default ordering.

Figure 2 plots the percentage gain of both the ordering variants with the baseline ordering. It is evident from Table II and Figure 2 that the second variant of PURSE performs best among all the ordering strategies with a gain of 0 – 10% for 18 designs and 10% – 25% for 6 designs.

A. PURSE *vs Static Ordering*

The property ordering strategy introduced in Section III was further classified into two categories; one is static ordering, and the other refers to dynamic ordering. In static ordering, the inconclusive goals are ordered only once after the first iteration, where an iteration refers to the resource budgets concerning the timeout strategy of the PDR algorithm. In the dynamic ordering strategy, the inconclusive goals are ordered after every iteration.

Figure 3(b) compares PURSE with a static COI-based ordering for design $D1$. Although we demonstrate that the variants of PURSE work better for $D1$, we have also verified for other designs.

B. PURSE *vs Random Ordering*

This section compares the property ordering algorithm with two randomly selected goal orders to emphasize the need for an optimal order. Moreover, for a design with N properties, a total of N! different goal orders are possible. Choosing any goal order randomly and hoping to work better is not ideal. Figure 3(a) provides a zoomed-in picture of the properties getting solved over time that compares D23 with two randomly chosen goal orders. Moreover, we chose D23 here because this design converges at the earliest, and all properties get solved. Figure 3(a) highlights that different variants of PURSE perform better than the random ordering.

V. Conclusion

This paper presents PURSE with two dynamic property ordering strategies that work by analyzing the statistical data dumped from the PDR-proof engine. Empirical results show that the PURSE algorithm works better than a structurally static ordered goal set and highlights that static orderings are no longer a viable approach. Moreover, as discussed, random ordering can lead to performance degradation, and a structured dynamic ordering strategy is the next step for multiple property verification. In future work, we shall look for other parameters that may produce better property orders and help converge faster. Also, we wish to inspect the overlap between the sequential structures of the properties to gain more insights into property grouping and ordering.

References

[1] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern soc design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[2] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts, "Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques," *IEEE Control Systems Magazine*, vol. 36, no. 6, pp. 45–64, 2016.

[3] P. Dasgupta, *A Roadmap for Formal Property Verification*. Berlin, Heidelberg: Springer-Verlag, 2006.

[4] E. Goldberg, M. Güdemann, D. Kroening, and R. Mukherjee, "Efficient verification of multi-property designs (the benefit of wrong assumptions)," in *Design, Automation & Test in Europe*, 2018, pp. 43–48.

[5] R. Dureja, J. Baumgartner, A. Ivrii, R. Kanzelman, and K. Y. Rozier, "Boosting verification scalability via structural grouping and semantic partitioning of properties," in *Formal Methods in Computer Aided Design, FMCAD*. IEEE, 2019, pp. 1–9.

[6] R. Dureja, J. Baumgartner, R. Kanzelman, M. Williams, and K. Y. Rozier, "Accelerating parallel verification via complementary property partitioning and strategy exploration," in *Formal Methods in Computer Aided Design, FMCAD*. IEEE, 2020, pp. 16–25.

[7] M. Chen and P. Mishra, "Functional test generation using efficient property clustering and learning techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 396–404, 2010.

[8] G. Cabodi and S. Nocco, "Optimized model checking of multiple properties," in *Design, Automation & Test in Europe*, 2011, pp. 1–4.

[9] G. Cabodi, P. Camurati, C. Loiacono, M. Palena, P. Pasini, D. Patti, and S. Quer, "To split or to group: from divide-and-conquer to sub-task sharing for verifying multiple properties in model checking," *Int. J. Softw. Tools Technol. Transf.*, vol. 20, no. 3, pp. 313–325, 2018.

[10] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation, VMCAI*, vol. 6538. Springer, 2011, pp. 70–87.

[11] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD*, 2011, pp. 125–134.

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, pp. 117–148, 2003.

[13] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Formal Methods in Computer-Aided Design*. Springer Berlin Heidelberg, 2000, pp. 127–144.

[14] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, CAV*, vol. 6174. Springer, 2010, pp. 24–40.