A Systematic Approach to Power State Table (PST) Debugging

Bhaskar Pal Suman Nandan Kaushik De and Rajarshi Mukherjee
Synopsys India Pvt Ltd
Bangalore, India - 560016
Email: {bpal,snandan,kaushikd,rmukherj}@synopsys.com

***Abstract*—Unified Power Format (UPF), allows designers to describe low power design intent and help the way complex low power integrated circuits can be designed, verified and implemented. The power architecture intent specification include structural specification like specification of the power supply network, power switches, retention cells, etc and functional specification like specification of the supply port/net states and the Power State Table (PST). Static Low Power checkers like MVRC takes the UPF PST as the golden reference and performs most of the static checks (e.g. like isolation/level shifter requirements). With the growing low power complexities of the designs, the number of PSTs in a design can be in the order of hundreds. Therefore it is becoming very difficult to ensure correctness/completeness/consistencies among these hundred of PSTs. Tools employ certain merging principles to merge these hundred of PSTs and inconsistent/incomplete specification of PSTs can lead to a merged PST which may not satisfy the actual low power intent. Therefore, an under-constrained or over-constrained merged PST leads to incorrect verification results. Manual debugging of what went wrong between these large PSTs can be very time consuming and erroneous. In this paper, we propose a systematic approach for consistency/completeness checks of the PSTs. Moreover if erroneous PST specification leads to incorrect merged PST and thereby generates incorrect verification result, we also offer a set of debugging aids to find out the exact context (root cause) of what went wrong. We believe that the proposed approach will significantly help the PST debugging flow for large designs. ‘**

I. Introduction

This Unified Power Format (UPF) [6], allows designers to describe low-power design architecture. The power architecture specification includes structural specification like supply port/net, etc and functional specification like supply port/net states and the Power State Tables (PST). Usually a large low power integrated circuits are partitioned into multiple power blocks/islands. Each of these power blocks can have multiple power states representing different voltage levels at which the block can function. The power state table (PST) of a block defines these power states in the block level UPF in terms of a set of supply ports/nets and a set of *supply relations*. Intuitively, every row of a PST defines one or more *supply relation*. In addition to these block level PSTs, there can be a top level (global) PST which defines the chip level power states.

Figure 1 shows the power blocks of a low power design. Each of these power blocks (BLKA, BLKB, BLKC and BLKD) has its own block level PSTs. The blocks interacts

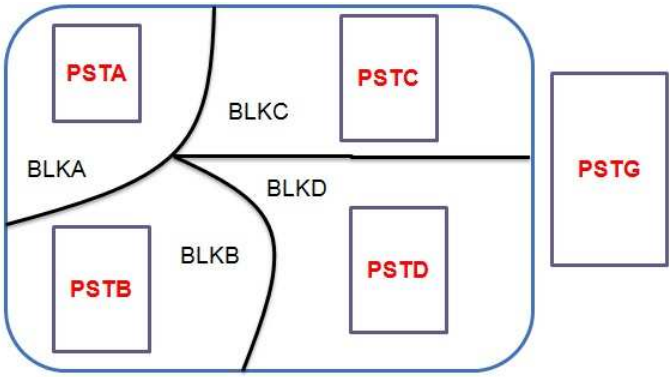

Fig. 1. Example low power design with four power blocks

with each other i.e. there are power domain crossing paths in the design.

Ideally, the global PST should represent states defined by conjunction of all the local PSTs. However, in practical scenarios, the global PST contains less number of power states, thereby constraining the power state space defined by the local block level PSTs.

On the other-hand, a local PST can constrain the global PST if the specification of local PST becomes incorrect/inconsistent with the rest. Merging of local PSTs can also produce additional new *supply relations*. Therefore if the global PST does not constrain the supply relations appropriately, the resultant PST generated by merging PSTs can have additional *supply relations* as well.

Today, the number of PSTs in a design can be in the order of hundreds. Tools [2] employ certain merging principles to merge these PSTs. As we have mentioned earlier, inconsistent/incomplete specification of these PSTs can lead to a merged PST which may not satisfy the actual low power intent. Due to these factors, enumerating the valid power states becomes difficult which in turn makes PST based debugging problem very tough.

To illustrate further, let us see what can be the impact of all these scenarios on the low power verification. Here we restrict our discussion on illustrating impact of PST debugging in the context of low power verification only. Static LP checkers like MVRC [2] takes the UPF PSTs as the golden reference and performs the static checks. Therefore, any in-

consistent/incomplete specification of these PSTs would lead to incorrect/improper verification results. Manual debugging of what went wrong between these large PSTs can be very time consuming and erroneous.

To elaborate further, every block level power verification is performed assuming that the *supply relations* encoded in the PST are correct and complete. However, when a block gets connected with its environment (chip level), if some of these *supply relations* get eliminated (either by the constraining global PST or while merging with other block level PSTs), the block-level verification results get changed. The change is very difficult to debug as, for the designer, the block level PST remains the same.

In this paper, we first elaborate different scenarios in which the PST specification becomes incorrect/inconsistent. Then, we propose a systematic approach for consistency/correctness checks of the PSTs. We further link the PST specification errors with verification results, leading to a root cause detection mechanism. The specific contributions are given as follows:

1) Block level checks: We check whether supply constraints are properly specified in the PSTs. We also detect potential partial specification of PST entries.
2) Chip level checks:
   - We propose connectivity based checks to see if the block level supply constraints hold at the chip level.
   - We detect partial specification of PSTs that can lead to potential design bugs.
   - We propose algorithms that can find out the exact context (root cause) for which supply assumptions gets broken leading to different verification results.
   - We propose a set of debug routines which can be used to debug/query different *supply relations*.

We describe an overview of the proposed PST debugging approach with an example. We demonstrate the proposed approach on some industry level designs. The results show that the proposed approach can catch potential PST errors and provide sufficient debug information to the designer/verification engineer to analyze the violations emitted by the static checkers. This approach can also be used by simulation based tools like MVSIM [3] for better coverage analysis.

The paper is organized as follows. We elaborate the PST debugging problem with the example design (see Figure 1) in Section II. A formal model and detailed algorithms are presented in Section III-E. The tool flow is presented in Section IV. The experimental results are shown in Section V. Section VI presents other works in this area and how this approach differs from those. Finally, the concluding remarks are presented in Section VII.

II. A Motivating Example

In this section we elaborate the PST debugging problem and the related issues/impacts with the example design (see Figure 1). There are four power blocks in the design. The block level PSTs (PSTA, PSTB, PSTC and PSTD) and the global PST (PSTG) are shown in Figure 2. There are overall five supplies in the design S1, S2, S3, S4 and S5. The port/power states of the supplies in the design are shown in Figure 3.

PSTA

	S1	S2
A11	off	ON
A12	ON	ON
A13	off	off

PSTB

	S2	S3
B11	off	ON
B12	ON	ON1
B13	off	off

PSTC

	S3	S4
C11	ON	ON1
C12	ON1	ON
C13	off	off

PSTD

	S4	S5
D11	ON	ON
D12	off	off

PSTG

	S1	S2	S3	S4	S5
G11	ON	ON	ON1	ON	ON
G12	off	off	off	off	ON
G13	off	off	ON	off	ON
G14	off	off	off	off	off

Fig. 2. Block level PSTs and global PST

```
add_port_state S1 –state {ON 0.9}
                  –state {OFF off}
add_port_state S2 –state {ON 0.9}
                  –state {OFF off}
add_port_state S3 –state {ON 0.9}
     –state {ON1 1.1} –state {OFF off}
add_port_state S4 –state {ON 0.9}
     –state {ON1 1.1} –state {OFF off}
add_port_state S5 –state {ON 0.9}
                  –state {OFF off}
```

Fig. 3. Supply port states

In this example, we have considered single voltage value port/power states. However, this approach is also scalable to multi-voltage values port/power states (which uses min-nom-max format). In these cases we convert the multi-voltage port/power states into single voltage value port/power states.

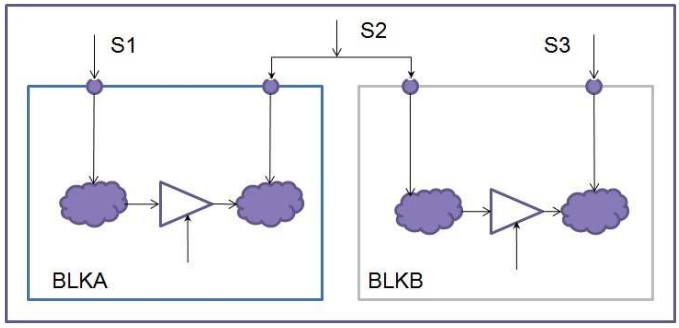

Fig. 4. Isolation devices inserted in BLKA and BLKB

PSTA		
	S1	S2
A11	off	ON
A12	ON	ON
A13	off	off

PSTG					
	S1	S2	S3	S4	S5
G11	ON	ON	ON1	ON	ON
G12	off	off	off	off	ON
G13	off	off	ON	off	ON
G14	off	off	off	off	off

Fig. 5. PSTA and PSTG to clarify the example

Note that PST for BLKA, PSTA, has a *supply relation* {(S1, off), (S2, ON)} in row A11. Therefore if there exists power domain crossing paths from S1 to S2, an *isolation* device needs to be inserted in each of these paths to stop the X-value propagation. BLKA has these *isolation* devices inserted in these paths and therefore block level static checks give no errors/warnings. Same *supply relation* holds between supply S2 and S3 in PSTB. BLKB also has proper *isolation* devices inserted in it. Figure 4 shows BLKA and BLKB with the devices.

However, once the chip level integration and static checks take place, these block level verification results change in the following manner We only elaborate the example in this section. The formal definition and algorithms are presented in Section III-E.

- *Supply relation in Block level PST gets overridden by global PST:* In our example, the *supply relation* {(S1, off), (S2, ON)} in PST state A11 in PSTA is not maintained by PSTG. PSTG represents stricter *supply relations* between supplies S1 and S2 (see Figure 5). Therefore this supply relation does not appear in the final merged PST. As a result, in the chip level static checks, all the isolation devices inserted between power domain crossing paths between S1 and S2 would be reported as *redundant*. This verification result may surprise the designer/verification engineer as the block level PSTs remain the same. Some may argue that a closer investigation of the global PST would clarify this scenario. Surely it does, but in a large design, with hundreds of PSTs and thousands of such violations, this debugging can be very difficult. As these violations appear due to inconsistencies between PSTs, the verification tool should consider these violations in a separate category and link these to the exact *root cause* to ease the debugging.
- The previous scenario represents the simplest way in which a block level PST loses one of its *supply relations*. However, there can be scenarios where a *supply relation* gets eliminated by other block level PSTs and this elimination may not be caused by a single block level PST. Inconsistent PST specification can eliminate a *supply relation* in multiple steps. Figure 6 shows another example (different from our example) in which a *supply relation* in one PST (PST1) gets eliminated/constrained by the other PST (PST2). Although this is also a direct elimination, debugging this eliminating scenario among hundreds of block level PSTs becomes very difficult.
- *Supply relations in a block level PST can get eliminated by other block level PSTs in* **multiple steps**. For example, the *supply relation* {(S2, off), (S3, ON)} represented by PST state B11 in PSTB does not get eliminated by the global PST PSTG (see Figure 2) as PSTG also holds this *supply relation* in PST state/row G13. This *supply relation* does not get eliminated directly by any other block level PSTs. So visually it is not possible to find out the eliminating PST. However, we will see (in the next section) how this *supply relation* gets eliminated in multiple steps by multiple PSTs during PST merging. Debugging such scenarios is very hard and non-intuitive. Therefore it needs hints from the checker tool to analyze such scenarios and link violations resulting from these to their exact *root cause*.

PST1		
	S1	S2
P11	off	ON
P12	ON	ON
P13	off	off

PST2		
	S1	S2
P21	ON	ON
P22	off	off

Fig. 6. Another set of PSTs PST1 and PST2 to clarify the examples

PSTA		
	S1	S2
A11	off	ON
A12	ON	ON
A13	off	off

PSTB		
	S2	S3
B11	off	ON
B12	ON	ON1
B13	off	off

PSTC		
	S3	S4
C11	ON	ON1
C12	ON1	ON
C13	off	off

PSTD		
	S4	S5
D11	ON	ON
D12	off	off

PSTG					
	S1	S2	S3	S4	S5
G11	ON	ON	ON1	ON	ON
G12	off	off	off	off	ON
G13	off	off	ON	off	ON
G14	off	off	off	off	off

Fig. 7. Indirect elimination of supply relation

Algorithm *PstMerging (PST1, PST2, PSTM)*

Input: PST1, PST2

Output: PSTM

1. Let S1 is the set of supplies of PST1
2. Let S2 is the set of supplies of PST2
3. The set of supplies of PSTM = S1 U S2
4. For each state (row) St1 of PST1
    1. For each state (row) St2 of PST2
        1. For each *supplyVoltPair (S, V)* of St1
            a. If supply S is present in St2 with different value V1, discard merging St1 and St2
        2. The set of *supplyVoltPair*s in the new PST row St1St2 of the merged PST PSTM contains union of all the *supplyVoltPair*s in St1 and St2.

Fig. 8. Merging of two PSTs

III. Formal models and Algorithms

In this section we first define some of the concepts that we use throughout the paper. We illustrate the concepts with examples. In Section III-A, we illustrate how the PST merging takes place for a UPF with multiple PSTs. In Section III-B, we illustrate the concept of eliminator PSTs which is a key notion used in this paper. Section III-C elaborates how different PST entries (we refer as *supplyVoltPair* below) gets associated with each other leading to a PST entry with eliminator PST. In Section III-D, we illustrate the approach to find eliminating PST sequence for a *supply relation*. Next we present outline of the algorithms to find out eliminating PST sequence responsible for elimination of a *supply relation* in Section III-E. Finally in Section III-G, we illustrate how violations get linked to their root cause.

To begin with, lets first define the *PST entries* formally. We call a PST entry a *supplyVoltPair*.

- **supplyVoltPair:** This is a pair (S, Volt) where S is the supply port/net and Volt is a voltage value specified by one of the *add_port_states*/*add_power_states* in the UPF.

Typically every PST entry can be represented as a *supplyVoltPair*. For example in PSTB (see Figure 7), PST state B11 has two *supplyVoltPair*s – (S2, off) and (S3, ON). The specification of *supplyVoltPairs* in a PST play a significant role in PST merging process leading to influence the outcome of the checkers. In this paper we use this concept of *supplyVoltPairs* to illustrate how incorrect/incomplete usage of *supplyVoltPairs* in PSTs are closely related to each other and influence the final outcome from the checkers. Next we use this concept to formalize our algorithms.

A. PST merging - how it happens

Given a set of PSTs for a design which includes block level as well as global PST, static checkers use a set of steps/algorithms to find out the resultant PST. This resultant PST is called the merged PST and act as a *golden reference*. In [1], discussions related to different aspects of PST merging have been presented.

PST merging is a process which merges the PSTs in the UPF taking a pair at a time (it can be any order) and in each step generates a new merged PST. The outline of the algorithm is given in Figure III-A.

To illustrate further, if we try to merge PSTA and PSTB (see Figure 7), we see that the common supply in this case is S2. We start with PST state A11 of PSTA. As (S2, ON) is also

PSTA		
	S1	S2
A11	off	ON
A12	ON	ON
A13	off	off

PSTB		
	S2	S3
B11	off	ON
B12	ON	ON1
B13	off	off

PSTAB			
	S1	S2	S3
AB11	off	ON	ON1
AB12	ON	ON	ON1
AB13	off	off	ON
AB14	off	off	off

Fig. 9. Merging of PSTA and PSTB

PSTC		
	S3	S4
C11	ON	ON1
C12	ON1	ON
C13	off	off

PSTD		
	S4	S5
D11	ON	ON
D12	off	off

PSTCD			
	S3	S4	S5
CD11	ON1	ON	ON
CD12	off	off	off

Fig. 10. Merging of PSTC and PSTD

present in PST state B12 of PSTB, PST state/row A11 can be merged with state/row B12. Similarly, A12 can be merged with B12 and A13 can be merged with B11 and B13. The ultimate merged PST is shown in Figure 9.

However, if we try to merge PSTC and PSTD (see Figure 10), we see that the common supply in both the PSTs is S4. However, the *supplyVoltPair* (S4, ON1) in PST row/state C11 of PSTC does not appear in any states/rows of PSTD. This is an example of *incomplete specification* of PST. For this, during merging, PST states C11 cannot be merged with any states/rows of PSTD and therefore this PST state/row *gets eliminated* in the merged PST. Figure 10 shows this merging.

Now we formally define how to find out the eliminating PST trace/sequence of a *supply relation*.

B. Eliminator PST

We first define what is an *eliminator PST* for a *supplyVoltPair*.

- **eliminator PST (supply, supply_state):** A PST P is called an eliminator PST for a *supplyVoltPair* (S, St) if supply S appears in P but the port/power state St of supply S has not been used in P. Therefore if St of S has been used in any other PST, say P1, merging of P with P1 will remove all the PST rows from P1 where port/power state St of S has been used.

For example, both in PSTC and PSTD (see Figure 10), supply S4 has been used. However, in PSTD, *supplyVoltPair* (S4, ON1) does not appear. In this case, PSTD becomes the *eliminator PST* for *supplyVoltPair* (S4, ON1). This is because, whenever PSTC gets merged with PSTD, PST row/state C11 gets eliminated.

Therefore, any *supplyVoltPair* does not appear in the final merged PST if it has an *eliminator PST*.

C. Association of supplyVoltPairs

A supply value pair, which does not have an *eliminator PST*, can disappear from the final merged PST if during the merging process it gets associated with another *supplyVoltPair* which has an *eliminator PST*.

We now illustrate how any *supplyVoltPair* gets associated to another *supplyVoltPair*. A *supplyVoltPair* which is used in some rows/states in a PST P, gets immediately associated with all the other *supplyVoltPairs* used in those rows in P. For example, in row C11 of PSTC, (S3, ON) gets immediately associated with (S4, ON1).

A *supplyVoltPair* can also get indirectly associated with a *supplyVoltPair* during the PST merge process. Suppose, a PST P1 gets merged with another PST P2, and in that process a row r1 in P1 gets successfully merged with row r2 in P2. The merging of r1 and r2 will give birth to a row r1r2 in merged PST P1P2. Once this happens, every *supplyVoltPair* in r1 and r2, gets indirectly associated with each other in r1r2 of P1P2.

In the above example, when row B11 of PSTB gets merged with C11 of PSTC, *supplyVoltPair* (S2, off) gets indirectly associated with *supplyVoltPair* (S4, ON1).

Now we illustrate how a *supplyVoltPair*, which does not have an eliminator PST, can be eliminated from the final merged PST. *supplyVoltPair* (S3, ON) of PSTC does not have any eliminator PST, but it is associated with *supplyVoltPair* (S4, ON1) in PSTC which has an *eliminator PST* (which is PSTD). Therefore, (S3, ON) gets eliminated in an indirect manner and does not appear in the final merged PST.

Interestingly, (S2, off) also gets associated with (S4, ON1) when PSTB gets merged with PSTC. However, (S2, off) will survive in the merged process because (S2, off) is also present in PST row/state B13 which does not get associated with any *supplyVoltPair* which has an *eliminator PST*.

Therefore, a *supplyVoltPair* gets eliminated from a PST altogether if it has an *eliminator PST* or in all the PST rows/states in that it appears gets associated with a *supplyVoltPair* with an *eliminator PST* during PST merge process. If

this *supplyVoltPair* (in any of these PST row/states) does not get associated with a *supplyVoltPair* with an *eliminator PST*, it survives in the merge process.

Now we define how a *supply relation* gets eliminated during the PST merging process.

A *supply relation* $\{(S1,\ V1),\ (S2,\ V2),\ ....,\ (S_N,\ V_N)\}$ which exists in some of the block/global PST does not survive in the final merge PST if any of the *supplyVoltPair* in it gets eliminated in the during PST merging i.e. it has an direct/indirect *eliminator PST*.

For example, the *supply relation* {(S2, off), (S3, ON)} gets eliminated because (S3, ON) does not survive the PST merging steps. However, as shows earlier, (S2, off) would be there in the merged PST.

D. Summary of approach to find eliminating PST sequence

Interestingly, the elimination of the PST rows that represents some *supply relation* may not occur at the same step. It will depend on how the PST merge algorithm works for a specific design. As PST merge algorithm continues in multiple iterations, the elimination of the rows can also be distributed over these iterations.

When some debug queries like debug_rail_order(S1, S2) (e.g. debugging whether there is any *supply relation* where S1 is off and S2 is ON) or debug_voltage_diff(S1, S2, voltage-value) (e.g. debugging *supply relation* where the voltage difference between S1 and S2 is same as voltage-value) referring a specific *supply relation* comes, the most simple way to trace why that *supply relation* has been eliminated, is to continue doing the PST merging and whenever a PST row (containing that *supply relation*) gets eliminated, store that information to the database and finally show the user all the ways in which that *supply relation* got eliminated. A trace can be defined as a cross product of a number of PST rows.

Although this approach is simple, it *does not* take into account the *supplyVoltPairs* with *eliminator PSTs*. As we have already shown that *supplyVoltPair* with *eliminator PSTs* are the root causes for elimination of various *supply relations*, the debug algorithm should perform the PST merging steps more *intelligently*. Therefore, to show how a *supplyVoltPair* gets disappeared in the merging process, the algorithm needs to cleverly choose the candidate PSTs with which the next merge happens.

For a *supplyVoltPair* that appears in row r1 of PST P and has association with N other *supplyVoltPair*, if there are M other PSTs with which the next merge can be done, give higher weight age to those PSTs where associates of the (N + 1) supplies has indirect or direct eliminator PST.

E. Outline of the algorithm

The objective of this algorithm is that - *Given a supply relation to the query engine, like debug_rail_order(S1, S2) or debug_voltage_diff(S1, S2, voltage-value), the algorithm finds out the minimum sequence of PST merging steps by which the relation gets eliminated.*

There can be two variants of this algorithm.

1) We build an association Graph from the PSTs in the UPF a priori and try to answer queries from that graph. This may have memory overhead.
2) We build some data base related to *supplyVoltPairs* which have eliminator PSTs or which are associated in a direct/indirect manner with the *supplyVoltPairs* which have *eliminator PSTs*. This database associates weights to each of the *supplyVoltPair* in the PSTs that indicates its distance from one of the *supplyVoltPair* which has an *eliminator PST*.
   After we have this we take a debug query and try to build a trace (sequence of PST row merges) for this specific query. This is the dynamic variant of the algorithm.

In this paper we present this dynamic variant only. The algorithm is given in Figure 11.

An example illustrating the steps in Figure 11 can be found in Section III-F. The associated *associationGraph* is also given in Fig 12.

Note that the above algorithm can be slightly modified to form an association Graph which actually depicts how each *supplyVoltPair* in the PSTs are connected to *supplyVoltPairs* which have eliminator PSTs. Whenever step 6 happens, we have a transition of row of one PST to row of another PST in the graph. The definition of such association graph is given below. But whether such a graph should be retained in memory or not is left to the implementation.

associationGraph (supplyVoltPair(S, St))

1) A vertex on this graph is a *supplyVoltPair* (S, State).
2) An edge is a connection (undirected) between two vertices labeled by a PST pair (P1, P2) which shows a possible merging by which these two vertices can be associated.
3) An edge can have weights indicating the distance to a *supplyVoltPair* with an eliminator PST.

Once the above database (distance database) is complete, we can start implementing the queries. Given a debug query related to a *supply relation*, we first see whether it gets eliminated in the final merged PST. If yes, we first try to find out whether any *supplyVoltPair* in this *supply relation* has an immediate *eliminator PST*. If yes, then we get the answer. Otherwise, we next find out the sequence of PST merge operations by which this *supply relation* gets killed.

To start this searching, we need to select the *starter PST* which initiates this sequence/trace. If the *supply relation* exists only in one PST then that becomes the *starter PST*. Otherwise, we follow procedure *selectStarterPST* to select the *starter PST*.

For example, if the query is debug_rail_order(S1, S2), then first find out is there any PST where this occurs. If yes take the set of those PSTs.

Procedure selectStarterPST (*supply relation*).

1) If {(S1, off), (S2, ON)} is there in the merged PST, we do not proceed, we send the states where this appears.
2) If (S1, off) or (S1, ON) has an *eliminator PST* then we return that *eliminator PST*.

Algorithm *buildDistanceDatabase*

1. Find out all the supplyVoltPairs which has *eliminator PSTs* in the UPF. Let the set be E
2. For each supplyVoltPair, (S, V), in the PSTs that doesn't appear in E
   1. If in all the rows in which (S, V) occurs, (S, V) has been associated with at least one supplyVoltPair which has an *eliminator PST*, them include (S, V) to E
3. For each supplyVoltPair, (S, V), in the PSTs that doesn't belong to E
   1. For each row in some PST where (S, V) appears.
      1. Create a storage called N where N indicates the shortest distance of (S, V) to a member supplyVoltPair in E.
      2. Assign N (S, V) = 100000 (some big number)
4. For each supplyVoltPair, (S, V), in the PSTs that belongs to E
   1. Create a storage called N and assign N to 0
5. Create another set L = E
6. For each supplyVoltPair, (S, V), in the PSTs that appear in L
   1. In each *incomplete row* R of the PSTs where (S, V) appears
      1. For each associated supplyVoltPair (S1, V1) in row R
      2. If N (S, V) >= N (S1, V1) - 1, *continue*
      3. N (S1, V1) = N (S, V) + 1
      4. Insert (S1, V1) to L'
   2. Mark row R as *complete*
7. Set L = L' and goto 6

Fig. 11. Algorithm to find out eliminating PST sequence/trace

3) For each row R of the PSTs P where {(S1, off), (S2, ON)} (i.e. any *supply relation*) occurs
   - Weight (P) = min (N(S1, off), N(S2, ON))
   - For all the associated supplies (S, V) in row R
     a) Take the minimum N(S, V)
     b) If N(S, V) $<$ Weight (P), Weight (P) = N(S, V)
4) Start with PST with minimum Weight.

The above algorithm (step 3 to step 4) is generic and holds for any supply relation. Also note that a supply relation can be as simple a *supplyVoltPair* itself.

The weight of a PST is necessarily the indication of how quickly we get to the point where {(S1, off), (S2, ON)} *supply relation* gets eliminated.

Once the *starter PST* is computed, we search for the best candidate PST with which this can be merged leading to elimination of the *supply relation*. However, as we have already mentioned, it may take several PST merging steps to ultimately find out the PST sequence/trace that kills a *supply relation*. Therefore once the *starter PST* is found, to compute this sequence/trace, we need to find out a PST with which this can be merged. Once this merging is done and we get a merged PST, if we still need to continue tracing, we again need to select a PST from the remaining PSTs to merge with this merged PST. This process continues till we find out the entire set of sequences. Therefore to select the best candidate PST at each such iteration (for PST merging), we follow the following steps.

The following procedure *selectBestCandidatePST* takes two arguments:

- The merged PST, PSTM which represents a PST trace (which initially consists of only the *starter PST*.
- The set of candidate PSTs (PSTs which are not used to compute PSTM), with which PSTM can be merged next.

Procedure selectBestCandidatePST

1) For each PST row/state R in PSTM in which the *supply relation* exists.
   a) Let L be the set of *supplyVoltPairs* in row R.
   b) For each row R1 of each candidate PST, P_{next} which can be merged with row R of PSTM.
      - Weight (R1) = Minimum weight of the *supplyVoltPairs* in R1 (distance database helps in this step).
      - Weight (P_{next}) = Minimum (Weight (P_{next}, Weight (R1)).
2) Return the PST P_{next} which has minimum weight.

F. Example

To illustrate further, lets look at the example design in Figure 1. Lets assume we need to debug why *supply relation* {(S2, off), (S3, ON)} gets eliminated.

To illustrate the steps, first we need to build the distance database. Note that there is only one *supplyVoltPair* (S4, ON1) which has an *eliminator PST*. There step by step execution of *buildDistanceDatabase* is given as follows:

1) N(S4, ON1) = 0. Therefore, L = {(S4, ON1)}
2) N(S3, ON) = 1 and L = {(S3, ON)}
3) N(S2, off) = 2 and L = {(S2, off)}
4) N (S1, off) = 3 and L = {(S1, off)}

An example *associationGraph* for this example is shown in Figure 12.

The *supply relation* exists in only one PST, PSTB, therefore this becomes the *starter PST*. Searching the PST trace sequence is executed in the following manner.

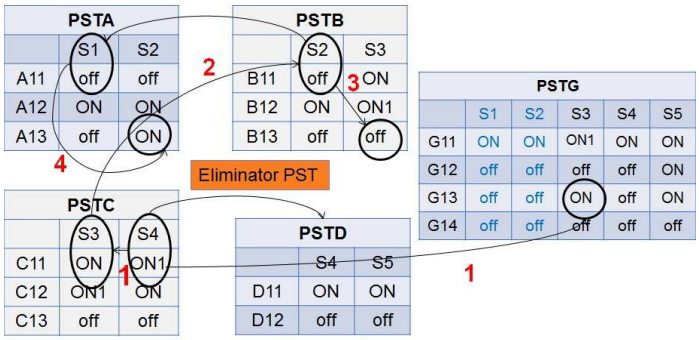

Fig. 12. An example Association Graph

1) In step 1 the candidate PSTs are PSTC and PSTG which can be merged with PST row/state B11 of PSTB. The candidate PST rows are C11 and G13 respectively.
2) As the Weight of (S3, ON) = 1 and (S4, ON1) = 0, PST row C11 of PSTC becomes the winner and therefore, PSTC becomes the next candidate PST to merge with PSTB.
3) Once PSTC gets merged with PSTB, *supply relation* gets eliminated and therefore the PST trace becomes (PSTB → PSTC). This becomes the *root cause* for eliminating the *supply relation*.

G. Linking violations with their root cause due to PST elimination

Usually any violation which gets generated based on PST computation like *isolation redundancy/missing* scenarios or *level-shifter redundancy/missing* scenario etc, are candidates for this approach. Once a violation gets generated, it stores the *supply relation* condition, for which it gets generated. For example, in an *isolation redundant* violation, the supplies (that drives the source/sink of the power domain crossing path), say S1 and S2 does not have a *supply relation* when S1 is Off and S2 is ON in the merged PST. Therefore for all such violations, we internally compute debug queries like *debug_rail_order* (S1, S2) and try to extract the PST trace/sequence that killed it. Once we have such trace, we report that as *root cause* for such violations. Figure IV shows this flow.

IV. Tool Flow

Figure 13 shows the usage of the proposed approach in the low power static checking process. The proposed approach works in different phases. The different phases are as follows:

- Once the UPF database gets populated, there is a *UPF consistency checker* which checks for simple scenarios like whether the block level port constraints are honored by the chip-level actual connections. It also checks scenarios like incomplete specification of PST, where a supply has been used in a PST but its port/power states are not used completely in the PST. This is an important check as this can lead to *eliminator PSTs*.

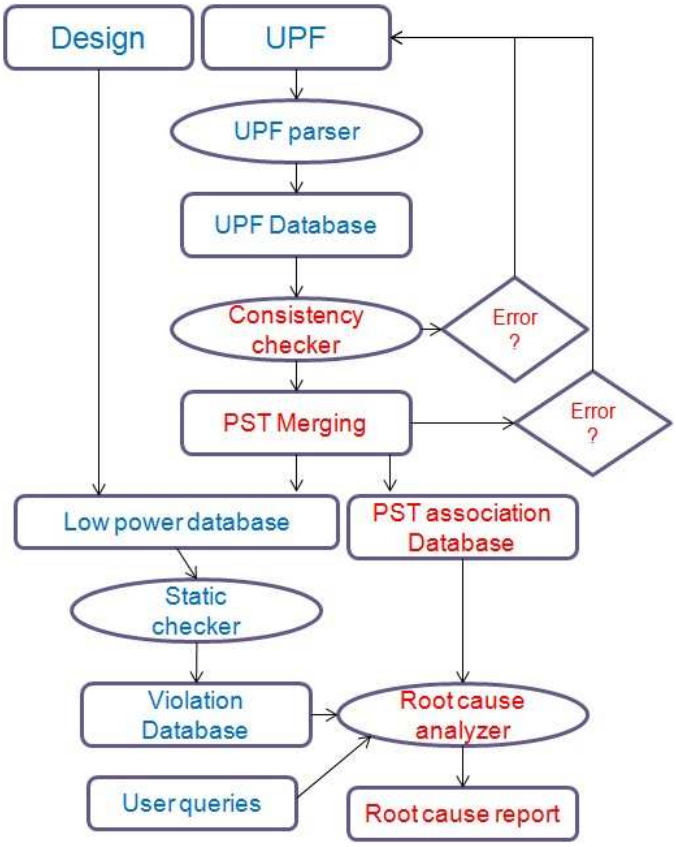

Fig. 13. Tool flow

- Once the PST merging happens, the PST association database gets populated. This database also contains information regarding *eliminator PSTs* for *supplyVoltPairs*.
- Once the violation database gets generated, the root cause analyzer gets kicked in and this uses the PST association database to link the violations to its actual root case (if any such scenario is there).
  In addition to this, the user level debug queries are also handled by this layer.

V. Results

We have implemented a prototype tool for the proposed approaches. The tool flow has been presented in Section IV. We have run the proposed flow on some industrial designs. The experimental results are shown in Figure 14.

In all of these test cases, we perform the checks in the following phases.

- We first check whether the top level ports (in block/chip level) have proper supply constraints. This is an important criteria that models the environment of the block/chip properly.
- We next check whether any intermediate boundary supply constraint is honored by its load/driver. This is required to ensure that the block level port constraints are maintained by the chip level connections.

Design	Size	Block level Issues	Chip level issues	Runtime overhead
Design-1	10 K	0	1	1 second
Design-2	2 M	2	2	< 5 seconds
Design-3	13.7 M	1	2	< 10 seconds
Design-4	45 M	1	5	< 1 min

Fig. 14. Experimental results

- We check whether there is inconsistencies in the PST specification and whether these cause extra violations.

In some of the designs we have observed that either the block port constraints were missing or not properly specified in the PSTs. In few designs, block level constraints get eliminated resulting to extra violations. In these designs, the global PST represents stricter *supply relations* resulting in killing of *block level supply relations*. Though redundant violations do not contribute to critical failures, it is always good to inform the user such situations to prevent critical bugs to creep in. In some of the designs, we have intentionally changed the UPF and injected errors to see whether the tool catches these.

In all of these runs, the runtime and memory overhead are negligible.

VI. Related Work

In [1], authors have presented scenarios that lead to incorrect/inconsistent PST. However, first, their approaches are dependent on some inputs from the designer which may not be available. Second, the detailed root cause detection mechanism and linking the root cause with resulting violation are not presented.

Some other works that discuss PST and low power verification issues can be found in [4], [5].

VII. Conclusion

In this paper we have presented an approach to systematically debug the PST. In addition to the simple checks like absence of port constraints, it includes involved checks that works on the PST merge process. The checkers are distributed throughout the static checker work flow. Therefore it enables early issue detection and correction. Ultimately we have proposed approaches to link the violations with their *root cause*. This root cause analysis is not generic, it only detects scenarios created by inconsistent specification of PSTs and PST merge process.

There are some future scopes which includes - (1) Displaying the culprit PST trace/sequence in GUI for ease of debugging, (2) Debug procedures for scenarios where additional *supply relations* gets added to the merged PST. Simple additions can easily be detected but there can be scenarios which are more involved and needs separate discussion.

References

[1] H. Vardhan, A. Bagotra and N. Bajaj, *Is Power State Table Golden*, DVCon, 2012.

[2] Synopsys-MVRC *www.synopsys.com/tools/veri_cation/lowpowerveri_cation/pages/mvrc.aspx*

[3] VCS with MVSIM *www.synopsys.com/Tools/Verifi cation/.../Pages/MVSIM.aspx*

[4] Power State Table for Low Power UPF and VP *www.scribd.com/doc/43157041/40/Power-State-Table*

[5] Low Power Methodology Manual *www.synopsys.com/community/partners/arm/pages/lpmm.aspx*

[6] *Unifi ed Power Format (UPF 2.0) Standard [Draft Version]*, IEEE P1801/D18,23rd October, 2008.