Accelerating Sign-Off Cycles: Automated Scenario Extraction from Large Design Landscapes

Gaurav Pratap
Synopsys,Noida, India
gauravp@synopsys.com

Vishal Keswani
Synopsys,Noida, India
keswani@synopsys.com

Sachin Bansal
Synopsys,Noida, India
sachinba@synopsys.com

***Abstract* - For a design company, signing-off of a large SOC within the given timeline is a race against time. An important bug within a verification tool has potential to disrupt the full timeline of signoff cycle. To ensure a timely delivery of all the required bug fixes, Verification tool vendors need to be equipped with an efficient debug mechanism of user reported issues.**

In recent times, Increased size of SOC has made static verification time and memory consuming. In a SOC which contains billions of design elements, few cases of missing/false violations or large run time issues get reported by customer on any static tool. When such issues get reported at the time of final sign off stage of the chip, they become a gating issue for any static tool. In such case static tool vendors are expected to provide the fix in the tool on urgent priority. Late delivery of fix in EDA tool may lead to delay in validation cycle of design. Finding the root cause of the issue and provide quick fix in tool takes time as R&D, AE may not have direct access to design. Most of the EDA companies has created some kind of secure area where design companies provide their design access for debugging of issue. Due to various kind of restrictions shipping design to a secure network is difficult or takes time.

A verification software is a complex code and debugging the issue in the code using debug prints is a challenging task. High number of debug prints make it difficult to find root cause. In most of the case the debug print logs are so huge that engineer can't review it manually.

Due to time and memory complexity attaching debuggers on large design is cumbersome and slow. From the debug fields in violations and other reports, R&D or field only has limited knowledge about the design scenarios. It is difficult to create a unit reproducer.

Due to such kind of problems we need to have an automated method to extract the exact design scenario or reproducer from a large design. In the paper we are going to present a novel idea which will help creating a pin-pointed reproducer out of the huge design.

This paper describes an automated method to create a unit reproducer from a large design containing only minimum gates and existing constraints (UPF, others)

Keywords— Unified Power Format (UPF), System on Chip (SOC)

I. Introduction

In recent times, Increased size of SOC has made static verification time and memory consuming. In a SOC which contains billions of design elements, few cases of missing/false violations or large run time issues get reported by customer on any static tool. When such issues get reported at the time of final sign off stage of the chip, they become a gating issue for any static tool. In such case static tool vendors are expected to provide the fix in the tool on urgent priority.

To fix any issue in the tool R&D engineer need to first identify root cause of the issue. Generally, an issue occurs in any tool due to below reasons:

1. Wrong interpretation of the design attribute.
2. Incorrect annotation of auxiliary data information on the design elements
3. Incorrect/Missing handling of special design scenarios. Few examples are given below:
    a. Multi fanin/fanout paths
    b. Presence of converging path
    c. Presence of inout ports
    d. Presence of undriven/unconnected driver or load on few paths.
4. Improper handling of exception cases.
5. Other logical bugs in the software.

To know the exact root cause of issue, R&D need to understand, out of above-mentioned categories which category is leading to the reported issue.
Below methods are the traditional ways of root cause identification in a big design:

1. **Using debug prints –** Various debug dumps are added in the software which get printed during execution of software. These debug prints help understand the how the code flow worked on the given design and how software inferred various design and auxiliary information attributes.
2. **Using debugger –** A debugger or debugging tool is a computer program used to test and debug other programs (the "target" program). The main use of a debugger is to run the target program under controlled conditions that permit the programmer to track its operations in progress and monitor changes in computer resources (most often memory areas used by the target program or the computer's operating system) that may indicate malfunctioning code. Typical debugging facilities include the ability to run or halt the target program at specific points, display the contents of memory, CPU registers or storage devices (such as disk drives), and modify memory or register contents in order to enter selected test data that might be a cause of faulty program execution.
3. **Code profiling tools –** Code profiling tools like valgrind, collect, Memory recorder etc. also helps R&D engineers to understand code specific issue like extra run time taken by a specific procedure, memory corruption caused due to faulty code etc.
4. **Reducing the size of design by making unrelated portion of design as blackbox model –** Marking a portion of design as black box is a well-known method of reducing the size of design. When a block of the design is marked as black box, static tool does not expand the design elements inside that block. In this way this method reduces the runtime and memory requirement to run a design. R&D tries to understand the full design scenario related to reported issue and then tries to mark nonrelated portion of the design as black box and hence reduces the size of the design.

Each of the above-mentioned R&D debugging methods have their own limitation.

1. **Huge size of debug dump –** On the big SOC designs debug dumps are of very little help to understand the issue. As the design size increases amount of debug data generated by the tool also increase in the same proportions. Many a times even on the medium sized designs files containing debug dumps are so huge (size of several GBs) that are not easy to even open with file editing tools. Manual processing of such huge data and extracting relevant information are very difficult process.
   Shortening them dump information leads to loss of relevant information related to the reported issue.
2. **Extra runtime and memory requirement with debugging and other profiling tools –**
   Debugger and other profiling tools require an unoptimized version of tool binary to run with. Due to this, the run time and memory requirement increases significantly. Even we find a high-capacity machine to debug that particular issue, a simple bring up of design takes several hours.
3. **Reducing a design by defining few modules as black box has its own set of issues:**
   a. Sometimes the same issue does not get reproduced if we define any relevant module as black box module by mistake.
   b. Reproducing the same issue after reducing the size of design with black box requires knowledge of full design scenario.
   c. Design size reduction using black box method doesn't reduce the size significantly because the amount of design element outside the block box modules are still huge.

Due to such kind of problems we need to have an automated method to extract the exact design scenario or reproducer from a large design.

II. Steps to extract unit reproducer

This entire solution is comprised of four major components: -

A. Relevant design object identification

To extract the scenario from the design, tool needs to know at least one design object related to the concerned issue. Tool needs to know below information:

- Design object name
- Design object type (source/sink/intermediate) – this is not mandatory

This information can be provided to the tool in below manner:

- From user input - In some scenarios like missing violation or a crash or long runtime, user will have to provide a design object name related to issue. It could be source of a missing violation or sink of a missing violation or an intermediate pin in the path.
- Tool generated input - For the scenarios like false violation tool will be able to extract this information automatically using debug fields of the violation. User will be required to provide some violation marker (like violation ID).

To extract the relevant design object(s), tool will assign the priority to each debug fields of each violation flagged by the tool. The priority for each debug field will be internally decided in the tool. Refer to Fig. 1.

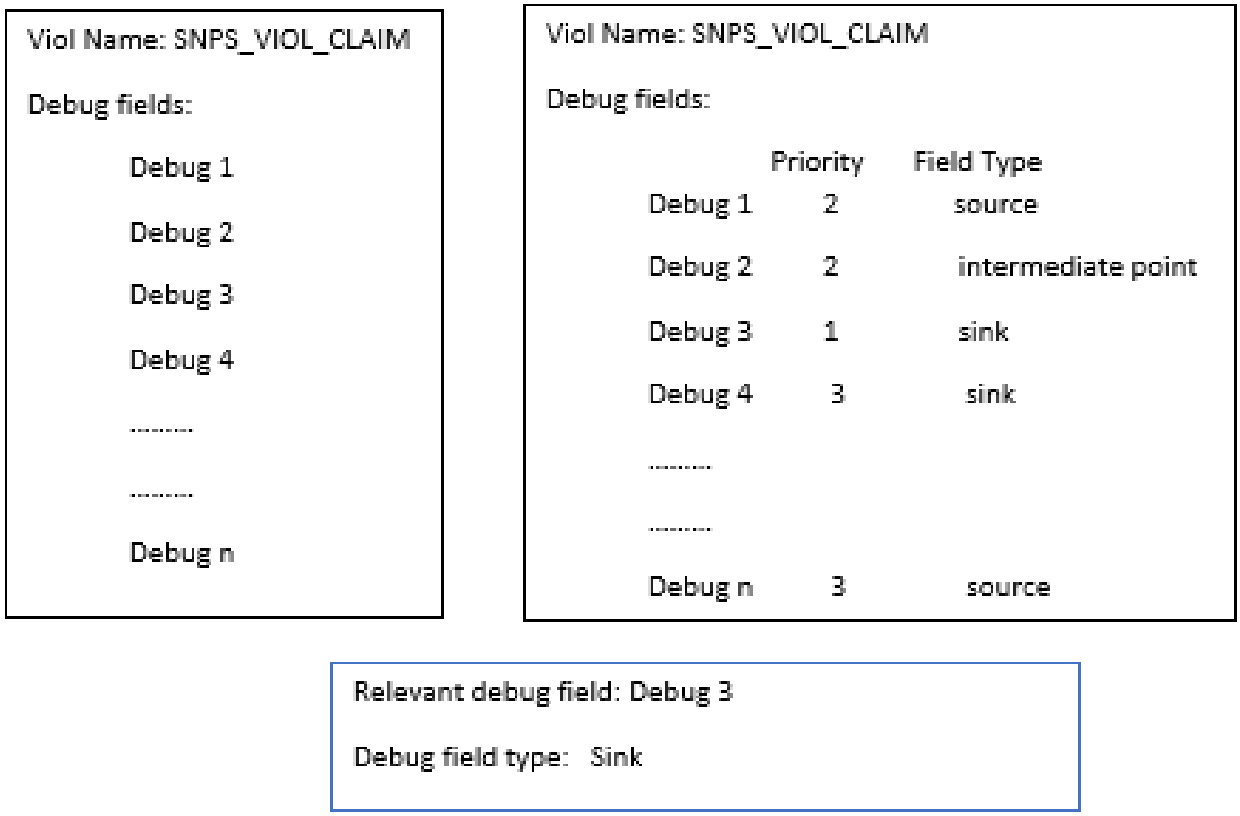

Fig 1 Tool assigned priority for debug fields

B. Design tree extraction

Based on the relevant design object and their type, this stage will identify all the design objects which need to be present in the extracted testcase. This stage will do a depth first search traversal on the complete design graph. Depth first design traversal will have below properties: -

- Traversal will start from the object identified as relevant design object in step 1
- Traversal direction (fan-in or fanout) will be decided based on object type identified in step1. Providing Object type is not mandatory, for example, if user gives an input pin, the tool understands that it can only act as a "sink" and will do a fan-in traversal. But if user specifies an intermediate net/port, then based on user input, tool will do the required traversals. Refer to Table I for traversal direction for each object type

traversal direction based on object type

TABLE I. TRAVERSAL DIRCETION BASED ON OBJECT TYPE

Object Type	Traversal Direction
Source	Fanout
Sink	Fan-in
Intermediate	Both (Fanin & Fanout)

- Design graph traversal will stop at the first sequential design object(flop/latch etc.). The traversal will pass through all the intermediate nodes. Set of the design objects (pin, port, net) reached during the design traversal will be part of extracted out design tree for the reduced testcase. These design objects will be stored in a container (Design tree container). Connectivity among the extracted design objects will be obtained from the original design graph.
- For each of the design object full hierarchy tree information will also be collected.

Sample pseudocode :-

```
foreach object (obj) in design tree container
set hier = parent Instance of obj
while (hier != Top Instance)
{
        add hier in design tree container
        set hier = parent instance of hier
}
 add Top Instance in deign tree container.
```

Power ground supply network information, related to the extracted design objects, will also be added to design tree container.

C. Dumping the design tree

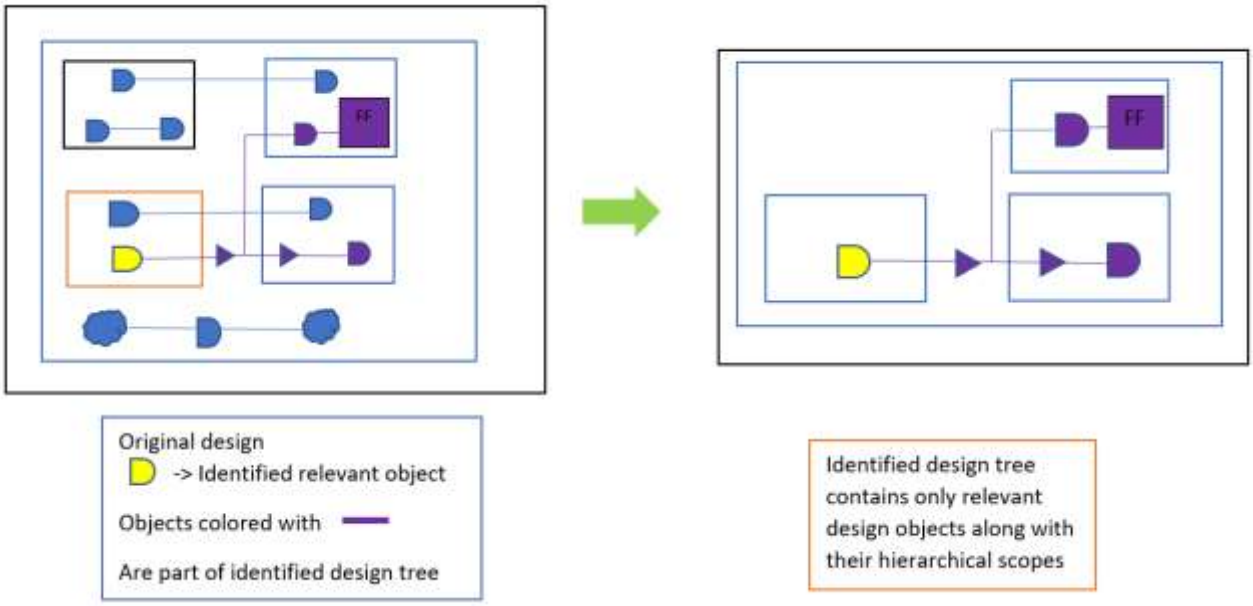

Fig 2 Extracting relevant path from design

A custom design tree dumper will dump the extracted design tree. Design tree dumper will take the list of design objects collected in the design tree extraction. Connectivity information of the design objects will be obtained from the original design graph. Based on these two inputs, a reduced design graph will be created which will be a strict subset of original design graph as shown in Fig. 2.

Sample pseudocode:-

```
for each object(obj) in design tree container
{
    get the connected object list(conlist) from original design graph
    for each object(newObj) in conlist
    {
        If(newObj is present in design tree container)
        connect the nodes of obj and newObj
    }
}
Dump this reduced design graph
```

Based on the requirement, this reduced design tree dump will be done in either of the following formats

- Plain Verilog format – Reduced design tree can be dumped in standard Verilog format. This dump can be used across tools and across versions
- Binary dump – Extracted design tree can be dumped in tool consumable encrypted binary format. This is in non-human readable format. This can be helpful in cases where some customer does not want any user to read his RTL logic.

D Intelligent readback of the extracted design

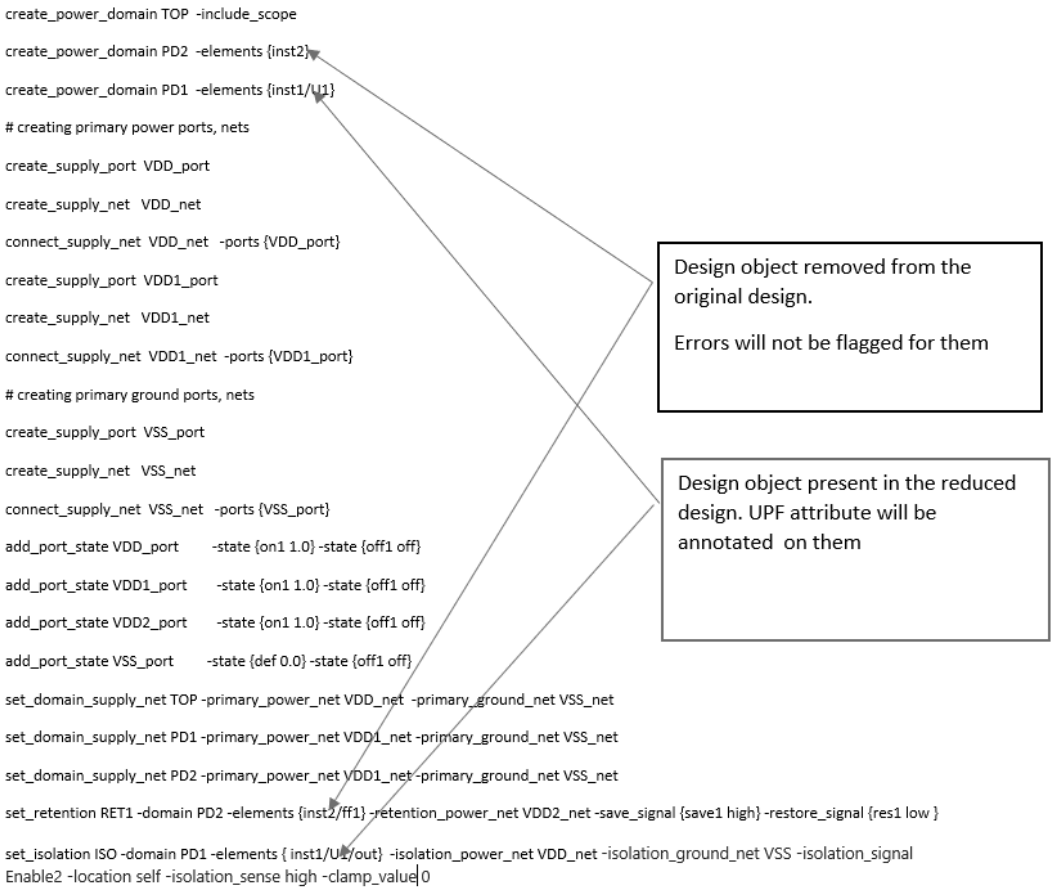

Fig. 3. Intelligent readback of UPF

A testcase comprises of the design files along with auxiliary data and liberty files. Auxiliary data (UPF/SDC etc.) contains the references of the design objects. Generally static tools flag errors while parsing of the auxiliary data files if referenced design objects are not found in the design files.

In this paper we have proposed to remove majority of the design objects in the extracted design file, but we are not reducing the auxiliary data files accordingly. Due to which in the read back run, tool needs to be intelligent enough to suppress the errors generated due to missing design objects and proceed the execution. Annotation of auxiliary data constructs on the remaining portion of design will not get impacted due to suppression of errors as shown in Fig.3.

IV PROPOSED FLOW OR METHODOLOGY

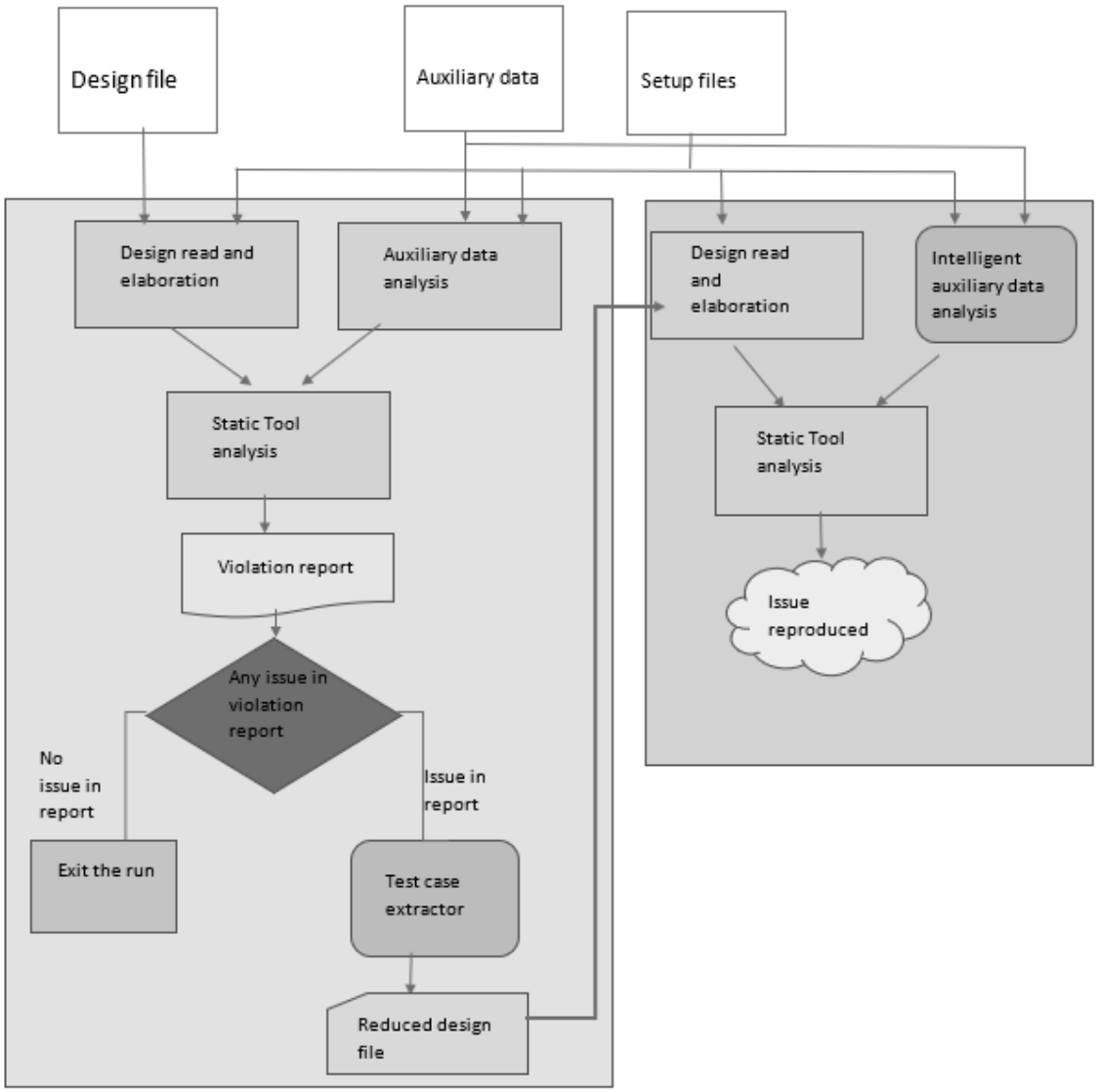

Fig.4 . 2 pass flow for issue reproduction

The proposed flow for Automated design scenario extraction related to any reported issue is as shown below (Fig. 4). This is a 2 pass flow where in Pass 1, user will identify the violation to debug and extract a reduced design for it. In Pass 2, the reduced design file is read back along with original constraints and issue is reproduced.

V RESULTS

With the use of this utility, we have successfully been able to reproduce all type of issues like electrical issues as well as functional issues. The utility has also been able to reproduce run to run issues as well which are very difficult to debug on actual design. **The design size reduction with this utility is huge which brough down the run time of reduced testcase in minutes whereas the runtime of original design was in hours.**
Gain in terms of design size reduction (~99.9% reduction in all the cases)

TABLE II. GAINS IN TERMS OF DESIGN SIZE REDUCTION

Design Name/Violation	***Flat Instance Count (Original)***	***Flat Instance Count (Reproducer)***
Design 1/ ISO_STRATEGY_REDUND	94K	16
Design 2/ ISO_INST_REDUND	270M	681
Design 3/ PSW_ACK_UNDETERMINABLE	82M	1389

TABLE III. GAINS IN TERMS OF RUNTIME

Design Name/Violation	***Runtime (s) (Original)***	***Runtime (s) (Reproducer)***
Design 1 / ISO_STRATEGY_REDUND	112	28
Design 2/ ISO_INST_REDUND	17340	196
Design 3/ PSW_ACK_UNDETERMINABLE	22610	1148

VI Future Work

- Automation can be provided so that at click of a button on a violation through GUI, a testcase is extracted out for that specific violation
- The current solution is only optimizing design. Future extensions can optimize the UPF/constraints as well

VII References

VC LP User Guide : https://spdocs.synopsys.com/dow_retrieve/qsc-u/vg/vc_lp/U-2023.03/PDFs/VC_LP_UG.pdf