SYNOPSYS®

WHITE PAPER

Automated Constraint Management for Faster Designer Productivity

Author

Rimpy Chugh
Product Management
Manager, Synopsys

Overview

Constraints management helps shorten the designer's manual constraints transformation effort across the design cycle with automated constraints management flows. The management of constraints refers to tasks that are not associated with verifying the correctness of constraints, nor associated with the generation of constraints, but with the transformation of constraints from one form to another and the subsequent verification of the integrity of the transformation. Constraint management includes the following flows:

- The mapping of constraints from one design representation to another, also known as constraints mapping
- The promotion of block-level constraints to the top-level, also known as constraints promotion
- The demotion of top-level constraints to the block-level, also known as constraints demotion
- The verification that two designs with two different sets of constraints are equivalently constrained, also known as constraints equivalency

Constraint Mapping

The constraint mapping flow deals with the issue of mapping constraints written for one design representation to another logically equivalent representation of the same design. For example, constraints written for logic synthesis, meant to apply to an RTL description of a design, need to be transformed before they can be used for place-and-route on a post-DFT inserted netlist or for final static-timing signoff on a post-route netlist. Various other transformations occur to a design as it goes through implementation: hierarchy is flattened and register names change as RTL is mapped to a specific technology. So, a constraint file that is written for RTL cannot directly be applied to its logically equivalent netlist. Engineers currently deal with these various constraints translation issues in one of two approaches.

Approach 1:

Designers use a tool that generates a new view of the design in order to generate SDC for the newly generated view. For example, a logic synthesis tool like Synopsys Design Compiler® or Synopsys Fusion Compiler™ writes out an SDC file that is later used for place-and-route (P&R) using Synopsys IC Compiler™ II. IC Compiler II can also be leveraged to write out an SDC for static timing analysis (STA) using Synopsys PrimeTime®.

The problem with this approach is that constraints change as implementation progresses. As timing issues are flagged, timing exceptions are added, leading to multiple versions of the SDC file for the same design. Managing each of these versions and ensuring that a change made in one version is mirrored in another is quite a serious challenge. Also, this approach relies on each technology used in the implementation flow to have robust SDC generation capabilities–which is often unrealistic and problematic if different vendors are used for synthesis, P&R, and STA within the design development flow.

Approach 2:

Designers ensure that SDC constraints are written so that the constraints are applicable to all design representations in the implementation flow. This can be challenging because it requires anticipating what the names of design objects will be as implementation progresses. This requires using wildcards, and complex queries that filter collections of pins and cells based on attributes. To manage such complexity, expert designers spend a lot of time writing constraints: mistakes are made, and inefficiencies are introduced. These mistakes can be chip-killing or, at the very least result in higher turnaround time.

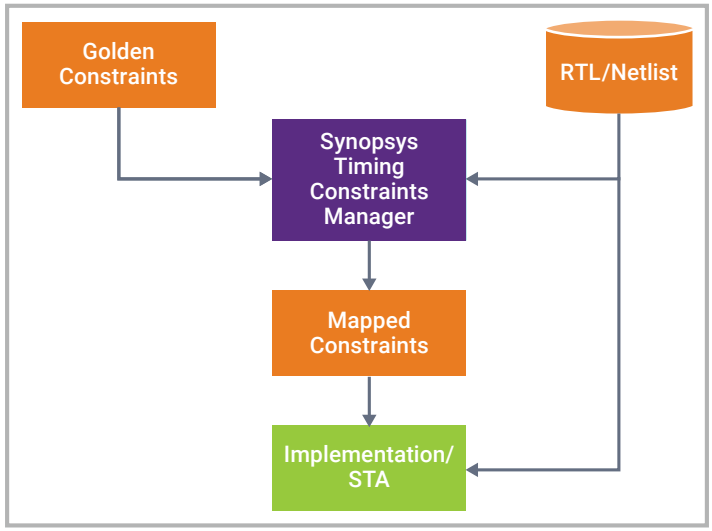

Figure 1: Constraint Mapping

Synopsys Timing Constraints Manager constraint mapping flow, shown in Figure 1, takes constraints that are written for one design representation as input, and a description of the corresponding design that they are supposed to apply to. Optionally, an engineer can also provide a mapping file generated by a logical equivalence check (LEC) that provides a mapping between registers in the two design representations. Given this input, Timing Constraints Manager generates an SDC file for the target design representation. The generated SDC file is meant to be consumed by the step in the implementation flow that needs it (P&R, STA, etc.) and not maintained or manually modified. The Timing Constraints Manager constraint equivalence-verification flow verifies the integrity of the mapped constraints by checking that the golden and target design representations are constrained in the same way using the golden and mapped design constraints, respectively. Any updates to constraints are made to the golden-constraint file that is input to Timing Constraints Manager, not to the mapped SDC file generated by Timing Constraints Manager.

As an example of the mapping performed by Timing Constraints Manager, consider the constraints below that are meant for application to synthesis:

```
set_multicycle_path -setup 2 -from u_arbiter/ready_reg[3]/clocked_on

set_false_path -through genloop[0].u_adder/sum
```

Timing Constraints Manager maps these constraints to a netlist, without a mapping file from LEC, to generate the output shown below:

```
#From user.sdc line 1

set_multicycle_path -setup 2 -from u_arbiter_ready_reg_3/CP

# From user.sdc line 2

set_false_path -through genloop_0_u_adder/sum
```

Timing Constraints Manager transforms the RTL names to netlist names and handles hierarchy removal automatically. Comments in the Timing Constraints Managers generated SDC points back to the file and line number, considering which SDC constraint was introduced in the golden constraint file. Timing Constraints Manager maps a reference to a design object in the golden representation to a unique object in the target representation. If a unique, unambiguous match does not exist, then Timing Constraints Manager does not map the object, and instead issues a warning message such as the one shown below:

Warning: Design object 'ifu/dec/const_cpuid[0]' does not exist (found 'ifu/dec/'). (RTL-002)

The mapped SDC file in this situation looks as follows:

```
# Unmapped: port_pin_list ifu/dec/const_cpuid[0] (found 'ifu/dec/')

# From user.sdc line 1

# set_case_analysis \

# 0 \

# ** { }
```

The above issue is resolved by either fixing the way the constraint is written or by providing a mapping file (one that is automatically generated by a LEC tool or by creating one manually).

With Timing Constraints Manager constraint mapping flow, engineers can avoid spending long cycles writing complex constraints via TCL that must act as a golden constraint file applicable to all versions of a design as it goes through implementation. Using the same constraint mapping flow, engineers can also avoid runtime issues that are typically faced when manually written conflicting constraints are read into implementation tools such as Design Compiler. Using this constraints mapping flow, engineers can escape relying on or qualify the SDC writing capabilities of technologies used in the later design stages. Instead, engineers can rely on a single tool to automatically map their golden design constraints to a target design representation and leverage comments that make clear the source for each line in the golden constraint file. The integrity of the constraint transformation is further verified using SDC equivalence verification.

Constraint Promotion

Block-level constraints need to routinely be promoted up to the chip level. Often, the promotion of a block-level constraint file simply requires the addition of a hierarchy prefix to a block-level pin/cell reference, and the removal of clock and I/O definitions on the ports of a block. However, when generated clocks are defined inside a block, or when the timing exceptions refer to the clocks on a block, then the promotion of these constraints is more complex.

For example, consider the following block-level SDC:

```
create_clock clk -period 6 -name clk

set_false_path -from B -to Z

set_multicycle_path -from clk -to U2_reg -setup 2
```

Promoting 'set_false_path' only requires adding the hierarchy prefix for each instance of the block. The multi-cycle path (MCP), on the other hand, states that all registers inside the block that are clocked by `'clk'` have a 2-cycle MCP to `'U2_reg'`. Promoting this MCP requires establishing which clocks propagate to the block. Multiple clocks may propagate to the clk port on the block. Also, different clocks may propagate to the block for different instantiations of the block. For the engineers, even after establishing the top-level clocks that propagate to each instance of the block, the MCP cannot be specified relative to these top-level clocks. If it were, then the MCP would apply to paths that start at registers clocked by top-level clocks outside the block and end at `'uIP/U2_reg'`. This would be incorrect as the engineer never specify an MCP for paths that originate outside the block. The scope of the MCP needs to be maintained inside the block.

Engineers today either manually promote their IP constraints to the top level or ask their IP teams to write their constraints so they can be applied to both a standalone block and to the block in the context of the top-level design. The manual promotion of constraints is cumbersome, takes time, and is error-prone. For engineers to write IP constraints so they can be applied to both block and top is not always practical and requires conditional TCL that introduces uncertainty on whether the block is constrained the same way standalone versus in the context of a larger design.

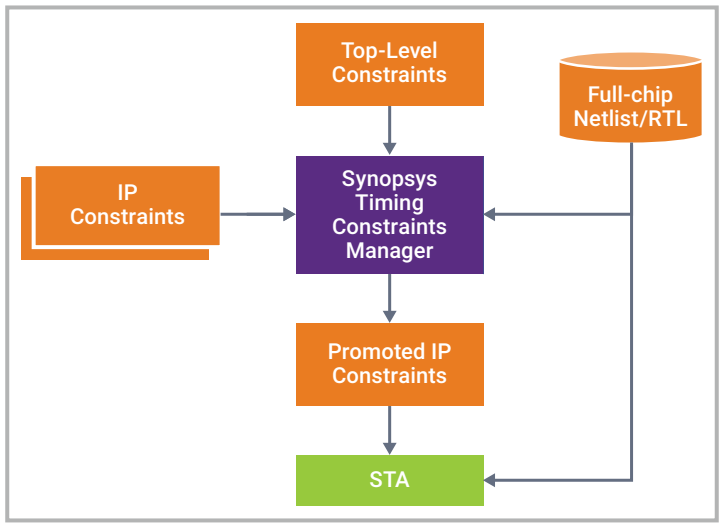

Figure 2: Constraint Promotion

Timing Constraints Manager constraint promotion flow shown in Figure 2 takes as input top-level constraints and top-level netlist or RTL, along with the block-level constraints that need to be promoted. Timing Constraints Manager allows constraints for multiple blocks to be promoted in a single run. An engineer can specify different SDCs for different instances of a block. Timing Constraints Manager takes the provided collateral and propagates top and block-level clocks (considering design constants, case analysis, and other SDC constraints that impact clock propagation) to establish which clocks propagate to the block-level `create_clock` commands. This information is used to promote the block-level constraints (generated clocks, timing exceptions, clock groups, etc.). Once constraint promotion is done using Timing Constraints Manager, the promoted constraints are compared to the block-level constraints using the SDC equivalence verification flow.

As an illustration of Timing Constraints Manager constraint promotion flow, consider the promotion of the block-level SDC, shown earlier in this section, using the following top-level constraints and for the top-level design shown in Figure 3:

```
create_clock sysclk -period 6

create_clock testclk -period 6

set_case_analysis 0 scanmode

set_clock_groups -async -group sysclk -group testclk
```

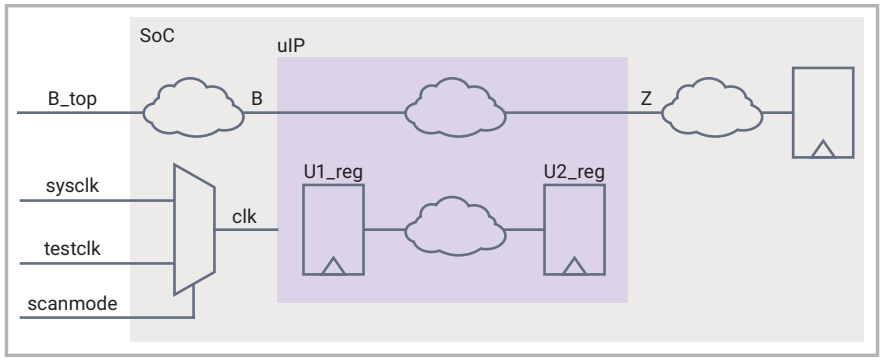

Figure 3: Example top-level design

The promoted SDC generated by Timing Constraints Manager is shown below:

```
# From IP.sdc line 2 for instance uIP

set_false_path -through uIP/B -through uIP/Z

# From IP.sdc line 1 for instance uIP

create_generated_clock -div 1 -master sysclk -name sysclk_1 uIP/clk

# From IP.sdc line 3 for instance uIP

set_multicycle_path -from sysclk_1 -to uIP/U2_reg -setup 2

# From top.sdc line 4 for instance uIP

set_clock_groups -async -group {sysclk_1} -group {testclk}
```

Timing Constraints Manager establishes that the clock propagating to `uIP/clk` is `sysclk`. A divide-by 1 generated clock, `sysclk_1`, is created corresponding to this top-level clock, and the MCP is written out relative to `sysclk_1`. This is done to ensure that the scope of the MCP stays within the block. Finally, `sysclk_1` is specified to be asynchronous to testclk, because the top-level constraints specify that `sysclk` and `testclk` are asynchronous to each other. Further, Timing Constraints Manager SDC equivalence verification flow between the promoted SDC applied to the top-level design, and the standalone block-level SDC indicates that all paths within the block are equivalently constrained.

Constraint Demotion

In a top-down design flow, it is necessary to take top-level design constraints and push them down to the block level to create block-level constraints from top-level constraints. This ensures that block-implementation is consistent with top-level constraints and ensures that block-level timing paths are optimized to meet top-level timing requirements. Constraint demotion reduces block-level implementation iterations and the risk that while block-level static-timing signoff is clean, top-level timing requirements are still not met.

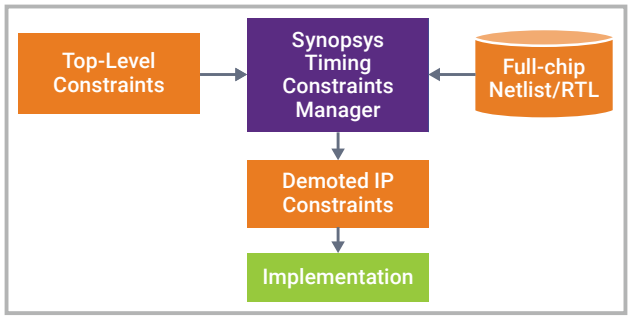

Figure 4: Constraint demotion

The Timing Constraints Manager constraint demotion flow is shown in Figure 4. Engineers provide Timing Constraints Manager top-level RTL or netlist along with top-level constraints. Engineers also specify the blocks for which SDC should be generated. When blocks are multiply instantiated, engineers have the flexibility of asking Timing Constraints Manager to generate one SDC file for each instance of the block, or a single SDC file that covers all block instantiations. Given this collateral, Timing Constraints Manager establishes the clocks and constants that propagate to each instance of a block. For each port on a block, Timing Constraints Manager establishes the registers that drive or are driven by the port and the clocks that clock these register. This information is used to establish the I/O delays for the port. Timing Constraints Manager does not attempt to allocate or budget I/O delays, but correctly identifies the clocks and clock edges relative to which a port must have its I/O delays specified. Timing exceptions and clock groups specified at the top-level are pushed down to the block-level. Once constraints have been generated for each block, the SDC equivalence verification flow is used to check whether each timing path within a block is constrained the same way as it is at the top-level.

As an illustration of constraint demotion, consider the top-level SDC shown below and the top-level design shown in Figure 3.

```
create_clock sysclk -period 6

create_clock testclk -period 12

set_case_analysis 0 scanmode

set_clock_groups -async -group sysclk -group testclk

set_false_path -from B_top
```

The Timing Constraints Manager demoted SDC for IP is the following:

```
# From top.sdc line 1

create_clock -name sysclk clk


# From top.sdc line 5

set_false_path -from B set_output_delay 0.0 Z -clock sysclk
```

Timing Constraints Manager establishes that the top-level clock that propagates to `uIP/clk` is sysclk. Since there is no I/O delay specified on `B_top` and that is the only top-level startpoint that drives `uIP/B`, no I/O delay is specified for port `B` on IP. Timing Constraints Manager establishes that `uIP/Z` is clocked by `sysclk`, resulting in the output delay on port `Z`. Finally, the false path specified at the top level is pushed down to the block level. Timing Constraints Manager SDC equivalence verification illustrates that the block-level timing paths on IP are constrained the same way as the paths within `uIP` are constrained at the top level.

SDC Equivalence Verification

There are several situations where constraints are transformed in various ways. During each such transformation, engineers need to ensure that the design is equivalently constrained. For example:

- When RTL constraints are mapped to a netlist, the intent is that both the RTL and netlist will be equivalently constrained.
- When block-level constraints are promoted up to the top level, the intent is that the promoted constraints constrain block-level paths at the top level the same way that block-level paths are constrained standalone.
- When top-level constraints are pushed down to the block level, the intent is that the pushed-down block-level constraints constrain the standalone block the same way as the block is constrained in the context of the top-level design.
- When multi-mode constraints are merged into a super-mode, the intent is that the super-mode constrains the design the same way as the multi-mode constraints.
- When a .lib model is extracted from RTL/netlist, the expectation is that the timing arcs on the .lib model match the I/O timing paths seen in the RTL/netlist.

In each of these situations, Timing Constraints Manager SDC equivalence verification should be used to establish whether a design with one set of constraints is equivalently constrained relative to another, or the same, logically equivalent design with a different set of constraints. The Timing Constraints Manager SDC equivalence verification flow is shown in Figure 5.

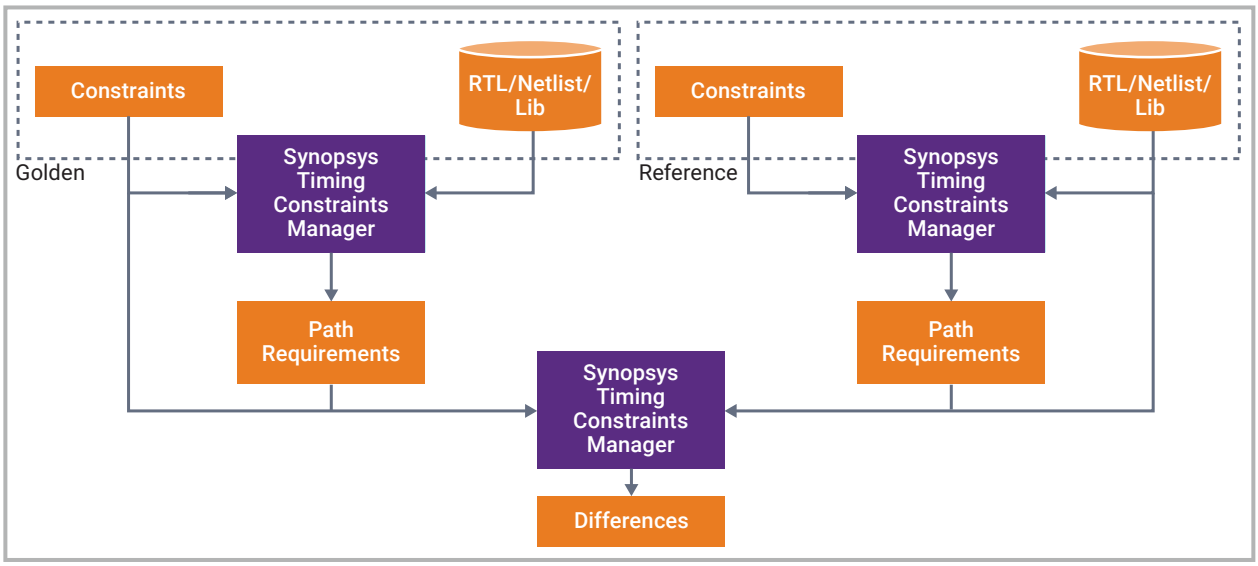

Figure 5: SDC equivalence verification

The Timing Constraints Manager SDC verification flow takes as input a design description that the engineer considers to be golden, and the constraints associated with this design description. In an RTL versus netlist flow, the RTL would be considered to be golden. In a constraint promotion situation, the block-level constraints would be considered golden. Any number of blocks and their associated constraints can be provided as input to the flow. An engineer has the flexibility to provide different SDCs for different instances of the block. In constraint demotion, the top-level constraints would be considered golden. In merged-mode equivalence verification, the multi-mode constraints would be considered golden. Any number of modes and their associated constraints can be provided as input to the flow. In a .lib equivalence checking flow the RTL/netlist from which the .lib model was extracted would be considered golden.

The Timing Constraints Manager SDC verification flow also takes as input a reference design description that is being compared to the golden constraints. In an RTL versus netlist flow, the netlist and its associated constraints would be the reference design. Optionally, users can provide a mapping file generated by an LEC tool that establishes a mapping between RTL and netlist registers. For constraint promotion, the top-level design and the promoted constraints would be the reference. For constraint demotion, the block-level SDCs would be considered a reference. For merged-mode equivalence verification, the merged-mode SDC would be considered a reference. For .lib equivalence verification, the .lib model would be considered a reference.

For both the golden and the reference design descriptions, Timing Constraints Manager is first used to establish the tightest timing requirement that needs to be satisfied by each path on the design. A path is a unique start and endpoint pair (where a start or end point may be a flop or a port). Establishing the tightest timing requirement for a path requires propagating clocks, and constants, applying timing exceptions, and then examining all the launch and capture clocks that propagate to the start and end points to compute the tightest timing requirement that needs to be satisfied by the path. This is much like what is done by STA, except that Timing Constraints Manager is not computing the timing slack for a path - it has no delay information - just the tightest timing requirement that the path needs to be met.

Once the tightest timing requirements have been computed for each path on both golden and reference designs, these designs are compared to establish if they are the same. For RTL versus netlist comparison, this SDC equivalence verification requires mapping RTL registers to netlist registers and then comparing the path requirements. For merged-mode constraint equivalence verification, this requires computing the tightest timing requirement for a path across all modes and comparing those requirements to the one established by the merged mode. For block versus top equivalence verification (used to verify constraint promotion or constraint demotion), this requires breaking top-level paths into sub-paths that are contained within the block being compared.

For example, consider the design shown in Figure 6, where the constraints on block IP are being compared standalone versus in the context of the top-level design that instantiates `uIP`. The top-level design has paths from registers `A_reg` and `B_reg` that go through `uIP/IN` to `uIP/X_reg`. The tightest timing requirement on these paths at the top level is compared to the requirement on the `IN→X_reg` path at the block level. Similarly, the tightest requirement on the paths from `uIP/Y_reg`, through `uIP/OUT` to `C_reg` and `D_reg` are compared to the requirement on the `Y_reg→OUT` path at the block level. Finally, the timing requirement on the path from `uIP/X_reg` to `uIP/Y_reg` is compared to the same path `(X_reg→Y_reg)` when the block is viewed standalone.

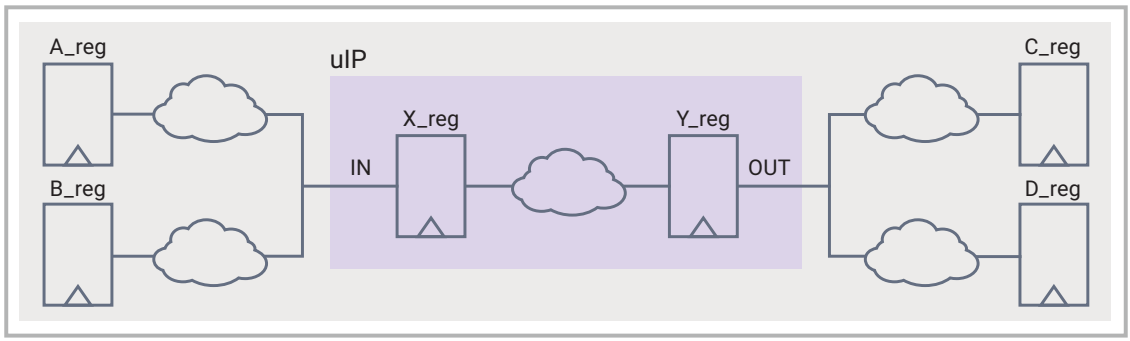

Figure 6: illustration of block-versus-top SDC equivalence verification

Once the comparison of path timing requirements is complete, differences are grouped into buckets, for easier review. For any path with a difference in timing requirements, an engineer is able to root cause the issue leveraging reports regarding how the path is constrained in the golden and reference design-whether there are differences in timing exceptions, differences in clock, or constant propagation that result in a difference in the timing requirements.

The Synopsys Timing Constraints Manager SDC equivalence verification flow is complete (all paths are compared), not noisy (when a difference is flagged, it is legitimate), fast (equivalence checking on a multi-million gate SoC is an overnight run), and accompanied by a powerful debug environment that brings up both the golden and reference designs simultaneously with links to interactive queries that help root cause differences for engineers.

10/18/23.CS1225611614-timing-constraints-manager-WP.
Pub: Oct 2023