

Techniques for Achieving Higher Completion in Formality

Author

Erin Hatch

Formality CAE, Synopsys, Inc.

Formality® is an equivalence-checking solution that uses formal, static techniques to determine if two versions of a design are functionally equivalent. Formality includes an intuitive, flow-based user interface to streamline the verification process. An automated setup file in Formality sets the commands and variables in Formality to match the setup used by Design Compiler® (DC) or Fusion Compiler™ (FC), eliminating manual, error-prone scripts. In addition, Formality can verify sophisticated DC or FC optimizations leading to a high rate of the first-pass completion.

Verification Flow with DC or FC

The Formality guidance file, known as the automated setup file (.svf file extension), will dramatically increase verification performance and completion, especially when verifying data-path designs that are highly optimized. The automated setup file contains hints regarding both guidance and setup that will help in verification. This file provides a way for Formality to automatically detect, and independently verify, many of the design optimizations that take place during synthesis.

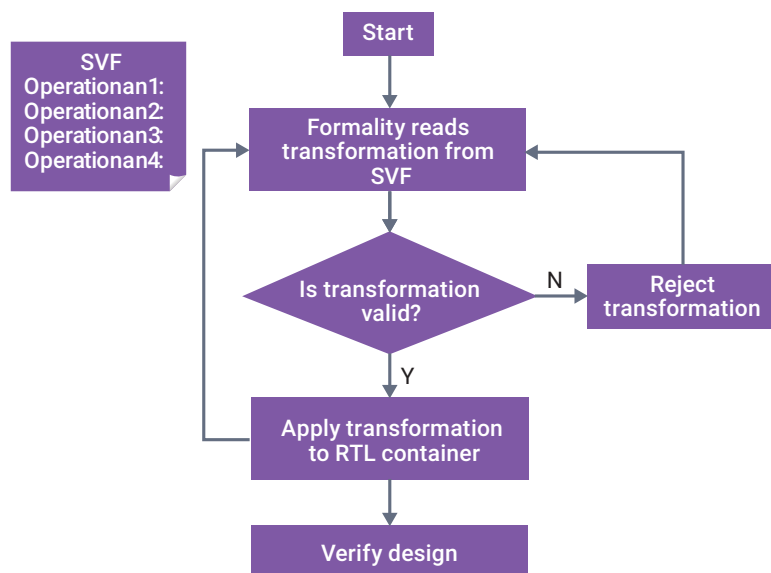


Figure 1: Formality uses the automated setup file to transform its view of the RTL to resemble the operator structure used by Design Compiler or Fusion Compiler

Design Compiler or Fusion Compiler automatically generates an automated setup file during synthesis. It contains important information about object name changes, constant register optimizations, duplicated and merged registers, FSM re-encoding, and information about advanced data-path optimizations. All the data that is on- by-default in this flow will either be implicitly or explicitly proven before or during verification, or Formality will not use it. The automated setup file is an absolute necessity when dealing with register phase inversions, FSM re-encoding and retiming. For detailed information about the automated setup file, refer to the automated setup file whitepaper on SolvNet:

https://www.synopsys.com/cgi-bin/verification/pdfr1.cgi?file=guided_simpli_equivcheck_wp.pdf

Formality has a feature called the auto-setup mode. You can start this mode by setting the `synopsys_ auto_setup` variable to `true`. In doing this, you are telling Formality to make the same assumptions that were made in synthesis. Using auto-setup mode will help increase your out-of-the-box verification success.

Auto setup mode will work either with or without, using the automated setup file; including it, however, means that the auto-setup mode will do more setup for you. Setting this variable will change how undriven signals are handled and RTL interpretation such as recognizing synthesis directives like `synopsys_full_case` and `synopsys_parallel_case`. This mode will also do the setup for you to include clock gating and automatic disabling of the scan. The Formality transcript will show which variables were changed from their defaults. You can override any variable. Formality will take the last value that was set.

Hard Verifications

A hard verification is a verification session in which Formality cannot completely verify all compare points. The Formality transcript or log file either shows no apparent progress for several hours or the verification aborts (stops) due to design complexity.

Usually, hard verification involves verifying data-path designs. DC or FC produce very complex, highly optimized, data-path blocks. These blocks are structurally very different from the unoptimized RTL and are difficult to validate. Occasionally, hard verifications are due to causes found with CRC, parity generators, or chains of XOR trees.

```
Status: Matching...

***** Matching Results *****
32 Compare points matched by name
0 Compare points matched by signature analysis
0 Compare points matched by topology
96 Matched primary inputs, black-box outputs
0(0) Unmatched reference(implementation) compare points
0(0) Unmatched reference(implementation) primary inputs, black-box outputs
15(0) Unmatched reference(implementation) unread points
*****

Status: Verifying...

Status: Matching hierarchy...

Status: Pre-verifying recovered datapath block vs reference r:/WORK/test/mult_5 (/WORK/M_RTL_MULT_UN5_16_16).
Pre-verification of recovered datapath block SUCCEEDED
Number of unmatched datapath blocks found: 1
Use 'report unmatched_points -datapath' to report unmatched datapath blocks.
Use 'set_user_match' to match unmatched datapath blocks.

Status: Verifying...
..... 0F/0A/3P/29U (9%) 02/22/07 23:27 159MB/1747.60sec
..... 0F/0A/3P/29U (9%) 02/22/07 23:57 200MB/3536.84sec
..... 0F/0A/3P/29U (9%) 02/23/07 00:33 200MB/5665.90sec
..... 0F/0A/3P/29U (9%) 02/23/07 01:03 266MB/7379.06sec
..... 0F/0A/3P/29U (9%) 02/23/07 01:33 266MB/9136.15sec
..... 0F/0A/3P/29U (9%) 02/23/07 02:03 266MB/10921.82sec
..... 0F/0A/3P/29U (9%) 02/23/07 02:33 266MB/12664.95sec
..... 0F/0A/3P/29U (9%) 02/23/07 03:06 266MB/14637.13sec
..... 0F/0A/3P/29U (9%) 02/23/07 03:36 266MB/16426.55sec
..... 0F/0A/3P/29U (9%) 02/23/07 04:06 266MB/18138.65sec
..... 0F/0A/3P/29U (9%) 02/23/07 04:36 266MB/19889.62sec
```

Figure 2: Example of Hard Verification Appearing Hung or Stuck

```

Compare point mix[16] is aborted
Compare point mix[17] is aborted
Compare point mix[18] is aborted
Compare point mix[19] is aborted
Compare point mix[20] is aborted
Compare point mix[21] is aborted
Compare point mix[22] is aborted
Compare point mix[23] is aborted
Compare point mix[24] is aborted
Compare point mix[25] is aborted
Compare point mix[26] is aborted
Compare point mix[27] is aborted
Compare point mix[28] is aborted
Compare point mix[29] is aborted
Compare point mix[30] is aborted
Compare point mix[31] is aborted

***** Verification Results *****
Verification INCONCLUSIVE
(Equivalence checking aborted due to complexity)
-----
Reference design: r:/WORK/test
Implementation design: i:/WORK/test
3 Passing compare points
29 Aborted compare points
0 Unverified compare points

Matched Compare Points      BBPin    Loop    BBNNet    Cut    Port    DFF    LAT    TOTAL
-----
Passing (equivalent)        0        0        0        0        3        0        0        3
Failing (not equivalent)    0        0        0        0        0        0        0        0
Aborted                      0        0        0        0       29        0        0       29
Hard (too complex)
*****

```

Figure 3: Example of Hard Verification Aborted Due to Complexity

Let's discuss what makes a design hard to verify. Large designs are not necessarily a problem to verify if they are somewhat structurally similar. Verifying dissimilar designs is not really a problem if the designs are of moderate size. Datapath optimization, however, combines the two difficult problems. You get very large blocks of logic that are structurally different from the original design. Verifying optimized datapath is the most difficult equivalency checking completion issue.

Formality has three techniques that it uses for datapath verification.

- **Restructuring:** Formality will make its representation of the RTL look more like the gate-level netlist. Formality uses the information inside the automated setup file to accomplish this. The automated setup file shows Formality what transformations Design Compiler used. Formality will first validate these transformations, and if valid, will perform a similar transformation.
- **Using special datapath solver:** Formality has a special solver that finds specific operators in the netlist and matches them up with the RTL, like a multiplier for example. Formality proves that the implementation design contains all of the pieces of this operator, for every input and output associated with the component. It can then be removed from further verification, making it much easier to verify the remaining portions of the design.



Figure 4: Formality Datapath Solver

- **Other solver technology:** Formality deploys several other solvers. With these, design similarity plays a key role in verification completion.

Analyzing Hard Points

There are a number of different techniques for analyzing a given set of hard points, which is the first step in getting a resolution. The most efficient methodology is to use the Formality command `analyze_points` which will recommend what can be done in Design Compiler to make designs more verifiable.

Formality Command `analyze_points`

The Formality command `analyze_points` runs a set of heuristic analyses on the most recent verification to determine if there are potential causes for the failing or hard verification other than logical differences. This command produces a recommendation for hard points to use with the DC/FC `set_verification_priority` command such that a re-synthesis may provide a more verifiable netlist.

The DC/FC command `set_verification_priority` sets the `verification_priority` attribute on the specified objects so that different levels of optimization can be applied during `compile_ultra`. The `compile` command does not ungroup cells or designs which have this `verification_priority` attribute. More details on this command can be obtained by viewing the man page in DC or FC and is also elaborated on later in this document.

Specifically, for Hard Verifications, the command `analyze_points` produces a recommendation as to what `set_verification_priority` command could be used with Design Compiler.

This report may be generated as both a command-line report and in the Graphical User Interface Debug->Analyses Window.

Example: (The text in bold is the recommendation to use with the Design Compiler command `set_verification_priority`)

```
Formality (verify)> analyze_points {r:/WORK/test/data_bl/o_reg[11]}
```

***** Analysis Results *****

Found 1 Rejected Datapath Guidance Module

These modules contain cells that may be related to rejected datapath guidance.

r:/WORK/data_bl_0 in file /remote/user/test/data_bl.v

Module with rejected datapath guidance on cell(s):

- r:/WORK/data_bl_0/add_376
- r:/WORK/data_bl_0/add_377
- r:/WORK/data_bl_0/add_378
- r:/WORK/data_bl_0/mult_404

Affecting 1 compare point(s):

- r:/WORK/test/data_bl/o_reg[11]

Use 'report_svf_operation { 1027 }' for more information.

Try adding the following command(s) to your Design Compiler or Fusion Compiler script:
`set_verification_priority -high [ get_designs { data_bl_0 } ]`

Figure 5 shows what the output in the GUI may look like.

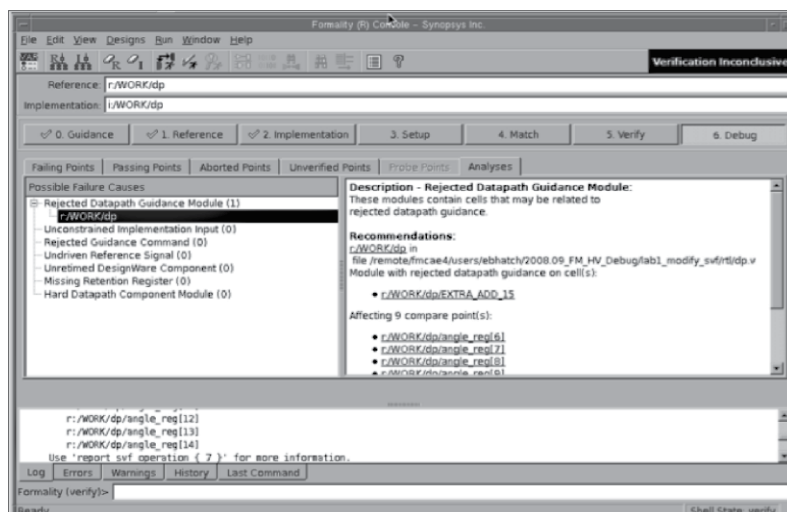


Figure 5: Using Formality Analysis Tool

The use of `set_verification_priority` is discussed later in the document under the header Resolving Hard Verifications -> `set_verification_priority`.

Other Analysis Techniques

While it is recommended to use the solution of `analyze_points` with `set_verification_ priority`, there is still benefit in understanding specific report commands associated with automated setup file guidance,

Using Guidance Summary and `report_svf_operation` Command

The Formality transcript displays the automated setup file guidance summary after running the `match` command or while running the `verify` command. This summary is available on demand by using the `report_guidance -summary` command. In the following example, note that all of the automated setup file guidance commands are rejected by Formality. This is an indication that either the wrong design or the wrong automated setup file is being used in verification.

***** Guidance Summary *****					
Command	Status				
	Accepted	Rejected	Unsupported	Unprocessed	Total
architecture_netlist:	0	1	0	0	1
datapath :	0	87	0	0	87

Figure 6: Guidance Summary Report with All Automated Setup File Guidance Commands Rejected

This next example is a more typical type of guidance summary report that you may see during a hard verification. Most of the guidance is accepted; however, some of the rejections are probably causing the hard verification.

***** Guidance Summary *****					
Command	Status				
	Accepted	Rejected	Unsupported	Unprocessed	Total
architecture_netlist:	1	0	0	0	1
datapath :	79	22	0	0	101
scan_input :	1	0	0	0	1
transformation					
map :	12	0	0	0	12
merge :	87	14	0	0	101
share :	23	2	0	0	25
tree :	13	0	0	0	23
ungroup :	4	0	0	0	4
uniquify :	2	0	0	0	2

Note: If verification succeeds you can safely ignore unaccepted guidance commands.

Figure 7: Typical Guidance Summary Report

Note that in this example there are 22 rejected `guide_datapath` commands from the automated setup file. Not all of the rejected guidance commands will contribute to a hard verification. So, there is no reason to investigate all 22 rejections, but only those rejections that contribute to hard-to-verify compare points.

To find a listing of the hard-to-verify compare points, use the following two commands:

```
• report_unverified_points
• report_aborted_points
fm_shell (verify)> report_unverified_points
21 Unverified compare points:
  21 unverified because of interrupt or timeout
    0 unverified because failing point limit reached
    0 affected by matching changes
Ref DFF          r:/WORK/dp/angle_reg[10]
Impl DFF         i:/WORK/dp/angle_reg[10]
Ref DFF          r:/WORK/dp/angle_reg[11]
Impl DFF         i:/WORK/dp/angle_reg[11]
...
```

If there are hundreds of these unverified or aborted compare points, you will need to reduce the list down to a more reasonable number to debug. The techniques used to reduce the number of these compare points are the same techniques used to help solve hard verifications and will be covered in this paper.

Use this automated setup file debugging command on the unverified or aborted compare points:

```
report_svf_operation -status rejected -summary hard_to_verify_compare_point
```

Here is an example of using this command:

```
fm_shell (verify)> report_svf_operation -summary r:/WORK/dp/angle_reg[10]
Operation Line Command Status
-----
6 55 Replace accepted
7 72 Replace rejected
8 91 Merge rejected
10 164 Boundary accepted
11 440 Constraints accepted
13 453 Datapath accepted
15 462 Replace accepted
16 515 Boundary accepted
```

Use the `-status rejected` option to see only the rejected automated setup file guidance. You can also remove the `-summary` option to see complete details of the rejected guidance.

Automated Setup File Rejections

When analyzing the automated setup file rejections, look for the first rejected guidance command. One rejection may lead to other rejections. For example, a `guide_merge` command may rely on a `guide_replace` command which then depends on specific object names as a `pre_resource`. This `pre_resource` must exist in the Formality version of the RTL with exactly the same name as specified in the automated setup file. Formality will need these `pre_resource` objects in its view of the reference design and then will combine or modify them into `post_resource` objects.

You will need to determine the reason for the automated setup file guidance rejection. The easiest issue to resolve is a naming concordance problem. This means that Formality named an object slightly or completely different from Design Compiler or Fusion Compiler as shown in the automated setup file. For example, here is a naming concordance information message:

```
Info: guide_merge 12826 (Line: 90536) Could not find pre_resource 'channel12/
Hslice/add_205_3' in design 'port_chnA'.
```

Formality could not find the name `channel12/Hslice/add_205_3` in the reference design (its view of the RTL). Because of this problem, Formality would reject the automated setup file guidance command needing this `pre_resource` object.

There may be other reasons for Formality's rejections of automated setup file guidance commands. Here is an example:

```
Info: guide_merge 13169 (Line: 92269) Unmatched boundary nodes.
```

These other types of problems are not as easily resolved by the end-user. If you cannot provide a test case to the Formality team, you may need to relax your synthesis constraints and re-synthesize. However, there are a few more things to try which will be reviewed in this paper.

You will need to determine the type of the rejected `guide` command. This is easily done by just viewing it. Common datapath guidance commands include `guide_merge`, `guide_share`, `guide_tree`, `guide_multiplier`, and `guide_datapath`.

Resolving Hard Points

Datapath intensive designs or designs containing complex design components, such as parity generators, XOR trees, or very large cones of logic can cause points to be hard during Formality verification. A primary solution to these hard points is to adjust the optimizations DC or FC performs during compilation. This can be done most efficiently with the command `set_verification_priority`, utilizing the targets as suggested by the Formality `analyze_points` command. In addition, there are other techniques both in Formality and Design compiler or Fusion Compiler that are worthy of note.

DC/FC Command `set_verification_priority`

When you have hard verifications in Formality, you can use the `set_verification_priority` command in Design Compiler or Fusion Compiler to help reduce hard verifications. These constructs enable DC/FC to adjust optimizations such that the potential for hard verifications is reduced. Hard verifications occur when Formality cannot complete verification due to issues such as design complexity. To enable DC/FC to adjust optimizations for hard verifications, use the following methodology:

1. Run Design Compiler or Fusion Compiler
2. Run Formality
3. Analyze any Hard Points using the Formality command `analyze_points`
4. Re-run Design Compiler or Fusion Compiler using the `set_verification_priority` recommendation produced by `analyze_points`

This command sets the `verification_priority` attribute on the specified objects so that different levels of optimization can be applied during compile. DC/FC does not ungroup cells or designs with the `verification_priority` attribute. In addition, if the `-high` option is used, that maps to the low effort of the optimization effort level and it disables DesignWare ungrouping.

The `set_verification_priority` command is not transitive, as illustrated in the following example.

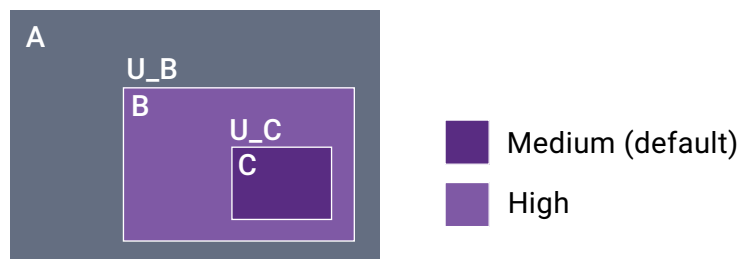


Figure 8: Using Formality Analyses Tool

`set_verification_priority [get_designs C] -- will set_verification_priority on only design C, designs A and B are unaffected.`

`set_verification_priority -high [get_designs B] -- will set_verification_priority of high on only design B, design C will still be set to medium and design A is unaffected.`

Addressing Concordance Issues

If there is a naming concordance problem, you can manually modify the automated setup file guidance file with a text editor to change the automated setup file name to match up with the Formality name of the same object. Note that Formality and DC/FC name objects using the line number on which they appear. For example, if the RTL code has $c = a + b$ on line 64, you will see `add_64` as an object name.

DC/FC will automatically generate an automated setup file named `default.svf`. This is a compressed binary file. You can use the `set_svf` command in Design Compiler or Fusion Compiler to change the default name or to generate additional automated setup files. Formality uses the same `set_svf` command to read in the automated setup file guidance information. Formality will automatically convert the automated setup file into ASCII format and will place the text file in its own subdirectory named `./formality_svf/svf.txt`.

You can modify this ASCII file to address naming concordance issues and then point Formality to the modified file when performing verification. Formality can use either a binary file or an ASCII file: it makes no difference. If the name change is difficult to achieve with text editing, you can use a guidance command to change the Formality name to match the current automated setup file name. The guidance commands to utilize are `guide_change_names` (for changing instance names) and `guide_rename_design` (for changing design names). The key to resolving the naming concordance problem is to change the names to match exactly between Formality and the automated setup file guidance file.

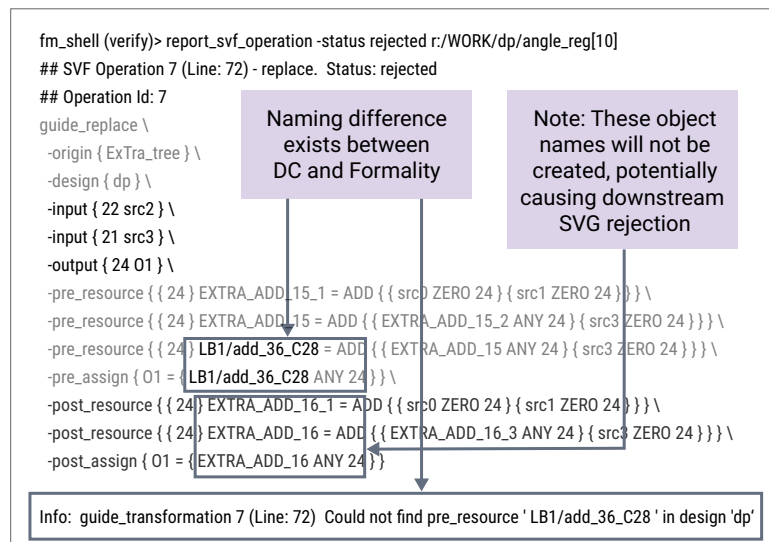


Figure 9: Example of Naming Concordance Issue (SVF)

Reducing Hard-to-Verify Points

There are additional techniques to reduce the hard-to-verify compare points in Formality. These techniques may possibly lead to a conclusive and successful verification. They include repartitioning the design, black-boxing hard blocks, hierarchical scripting and verification, factoring, and inserting cutpoints.

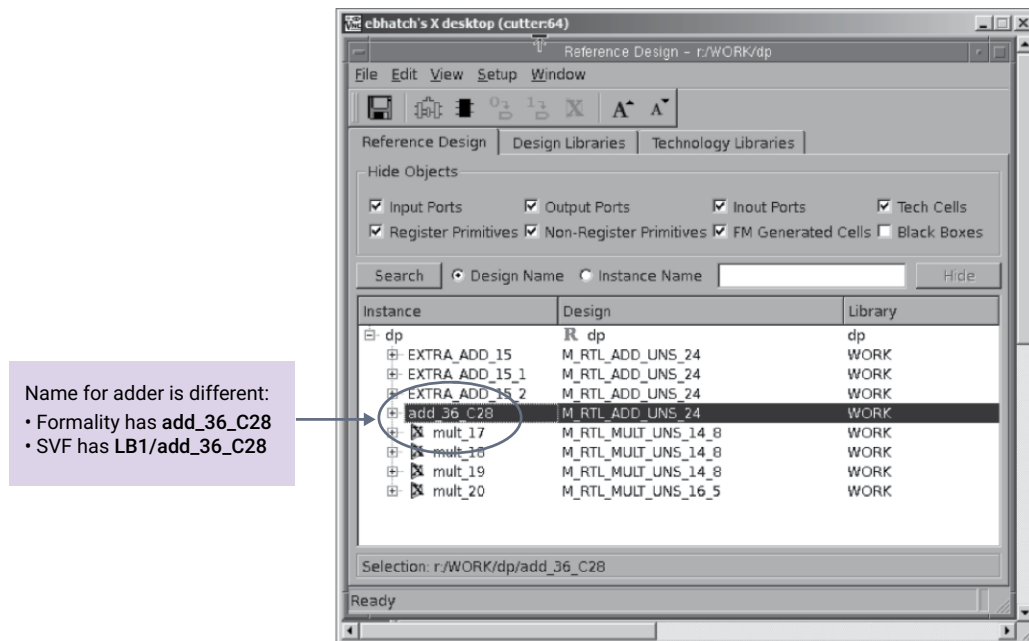


Figure 10: Example of Naming Concordance Issue (Formality GUI)

Partitioning

A partition is a grouping of compare points that have some overlap of logic cones. A partition can include one or more compare points. Formality will partition all remaining unverified compare points in a design every time it starts or restarts verification. This means that if Formality stops verification due to an interrupt or a timeout, it will repartition all remaining unverified compare points if the user specifies Formality to resume verification. Frequently this can reduce the number of unverified hard points, and sometimes can allow Formality to complete verification of all compare points.

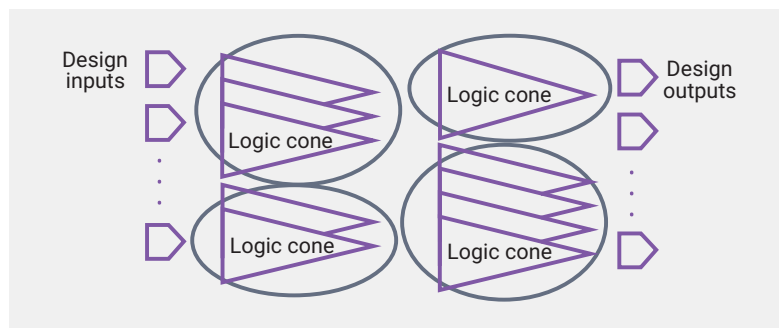


Figure 11: Example of Design Partitioning of Compare Points

There are some commands that can be used to direct Formality to stop verification. There is a partition timeout command which stops Formality processing a difficult partition after a time period:

```
set verification_partition_timeout_limit 4:0:0
```

There is also a total verification timeout command that directs Formality to stop all verification processing after a certain period of wall clock time:

```
set verification_timeout_limit 20:0:0
```

Here is an example of using the repartitioning workaround:

```
set verification_effort_level low
set verification_timeout_limit 2:0:0
verify;verify;verify    # solving easy cps
set verification_effort_level high
set verification_timeout_limit 6:0:0
verify                  # solving hard cps
report_aborted > report.aborted.cps.txt
report_unverified > report.unverified.cps.txt
```

There are things to be aware of with this technique. The first is that Formality may utilize auto-factoring during a verification effort level of medium or higher. Auto-factoring limits the benefits of small partition timeouts, which is why the effort level of low is used in the above example.

Furthermore, verification using UPF default mode can utilize auto factoring during all verification phases, so the partition technique is not recommended for UPF verifications. The one exception is if `verification_force_upf_supplies_on` is set to true, then the partition technique as illustrated is valid.

Black-boxing

Black-boxing sub designs may help only if both the reference and implementation designs have a hierarchy. You can isolate the hard-to-verify blocks by black-boxing them, and then verifying the remaining design. Afterward, verify the previously black-boxed design in isolation. This will reduce the size of the logic cones for the compare points in the isolated block.

Here are the black-box commands:

```
set_black_box ref_design_name
set_black_box impl_design_name
```

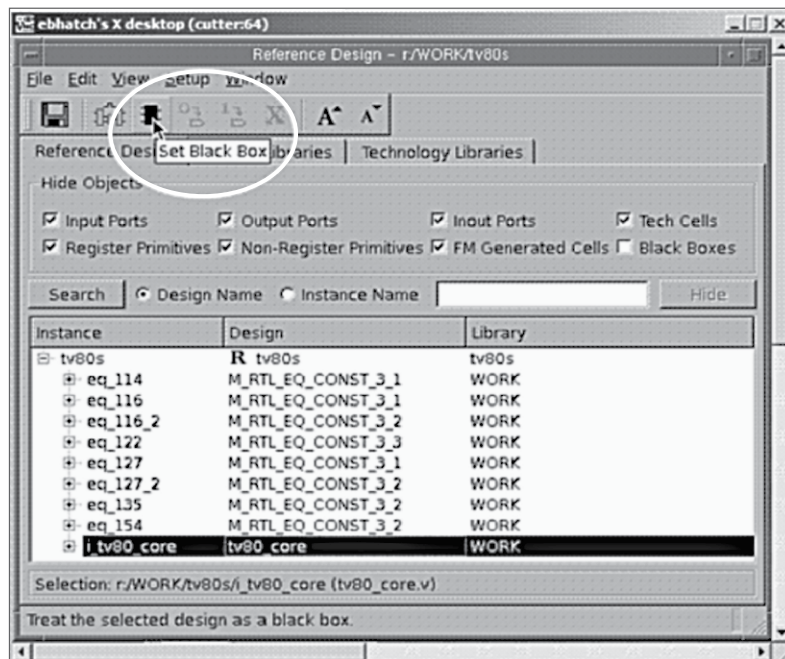


Figure 12: Setting a Design as a Black Box

Hierarchical Verification

Another technique to try with a hard verification is the use of the hierarchical verification script. Again, this technique would be useful only if both the reference and implementation design have a hierarchy.

There are two particular advantages of using hierarchical verification:

- The cones of logic will be smaller and therefore could make the design more verifiable
- Using hierarchical verification to identify which blocks are hard to verify

The disadvantage is there may be false differences due to synthesis optimization across hierarchical boundaries. This disadvantage may be overcome by using the `set_parameters -flatten` syntax on the parent blocks of those with false differences.

When using the `write_hierarchical_verification_script` command, consider using the `-level` switch to get to the block-level of interest. One suggestion is to start with level 3. Another option is to use the `-path` switch to verify only those blocks along the path of the hard-to-verify compare points. It is important to set a timeout limit when using the hierarchical verification script on a hard-to-verify design.

Factoring

Formality deploys automatic factoring during verification. Formality will search for selectors throughout the design. It will hold input of a selector at constant 0, perform verification, then will hold the input of the selector at constant 1, and perform verification again. Using constants on selectors greatly reduces the effort needed to complete verification. This technique helps complete verifications containing MUXed datapath operations.

You can use manual factoring in the unlikely event that automatic factoring does not help. You may want to try factoring on the reset signals or other signals of interest.

For example:

```
fm_shell> set_factor_point -type port $ref/data/sel2_*
Set factoring variable at 'r:/WORK/top/data/sel2_1'
Set factoring variable at 'r:/WORK/top/data/sel2_2'
Set factoring variable at 'r:/WORK/top/data/sel2_3'
```

Cutpoints

Using cutpoints is not commonly performed, but also could help with hard verifications. Cutpoints divide logic cones into smaller pieces thereby reducing complexity. The difficulty in using cutpoints is finding the same insertion points in both the reference and implementation designs. False differences are an inherent problem with this technique. Also, you might need to match up the nets or pins associated with the cutpoints using the `set_user_match` command if the names of the nets or pins are different from each other.

A potential strategy is to insert cutpoints on the outputs of DP_OP blocks or of DesignWare blocks within the logic cone of the hard-to-verify compare point. You can use the `report_matched_points -point_type block_pin` command to search for candidates.

RTL modifications

A final technique to avoid hard verifications is to modify your RTL to decrease datapath-related problems. Here are a few suggestions to try:

- Break up complex equations using temporary variables. Use separate lines in your RTL for datapath operations, instead of placing several operations in one single line of code
- Manually perform share or tree operations in RTL instead of Design Compiler optimization
- In general, you should avoid using explicit Don't Care assignments
- Use fully specified CASE statements
 - Assign known values to all branches of CASE to reduce implicit Don't Care conditions
- Avoid mixing sign and unsigned operations, especially when using bit-slices of buses
- For non-datapath designs like CRC, parity generators, or XOR trees
 - Maintain separate levels of hierarchy by changing Verilog functions into Verilog modules. Do not ungroup

Formality Methodology Training

There are additional training presentations and labs on SolvNet to practice using Formality. See the Formality Jumpstart training: <https://solvnet.synopsys.com/retrieve/024602.html> See also the hard verification debugging training: <https://solvnet.synopsys.com/retrieve/025252.html>

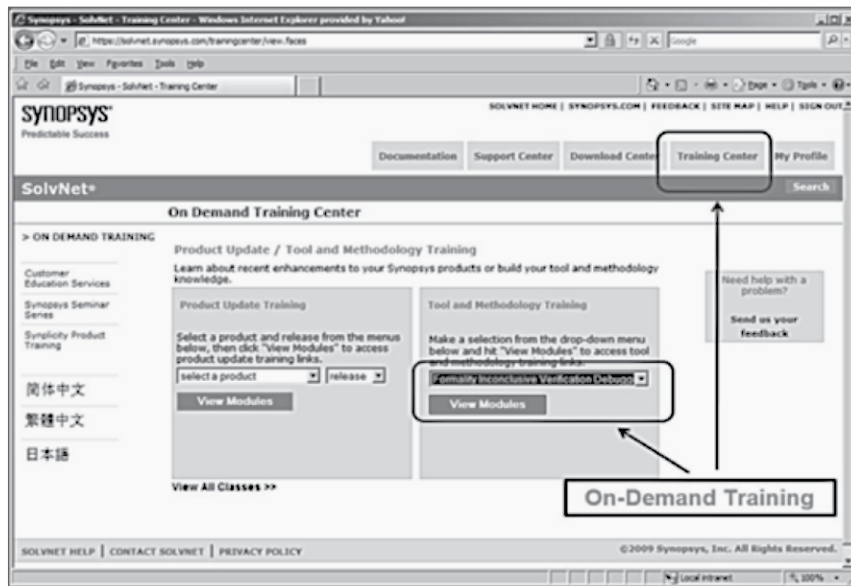


Figure 13: On-Demand Training

Summary

Formality delivers superior completion on designs compiled with Design Compiler or Fusion Compiler which provide advanced optimizations such as retiming, phase inversion and ungrouping. Equivalence checking remains an NP-complete problem. Any equivalence checking tool will encounter some designs that lead to a hard verification. When using Formality, the steps below should be used to assist in resolving the hard verification:

- Study the log files for obvious errors
- Create a minimized list of hard to verify compare points
 - report_unverified_points
 - report_aborted_points
- Find the cause of hard points
 - analyze_points
- Resolve the hard points
 - DC/FC set_verification_priority command utilizing the recommendation from analyze_points
- Additional Formality techniques include automated setup file modification, partitioning, black-boxing, hierarchical verification, factoring and inserting cutpoints
- Additional DC/FC techniques include using the latest release, disabling auto-ungrouping, creating a separate hierarchy, and RTL modifications.