SIEMENS EDA

# Catapult Formal User Manual

Software Version 2023.2

SIEMENS

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1 Introduction

Catapult Formal® addresses a key verification gap in ASIC/FPGA design flows that employ high-level synthesis (HLS) to build RTL designs. By using Catapult Formal in your verification flow, you will derive greater return on investment from the benefits of designing at a higher level of abstraction.

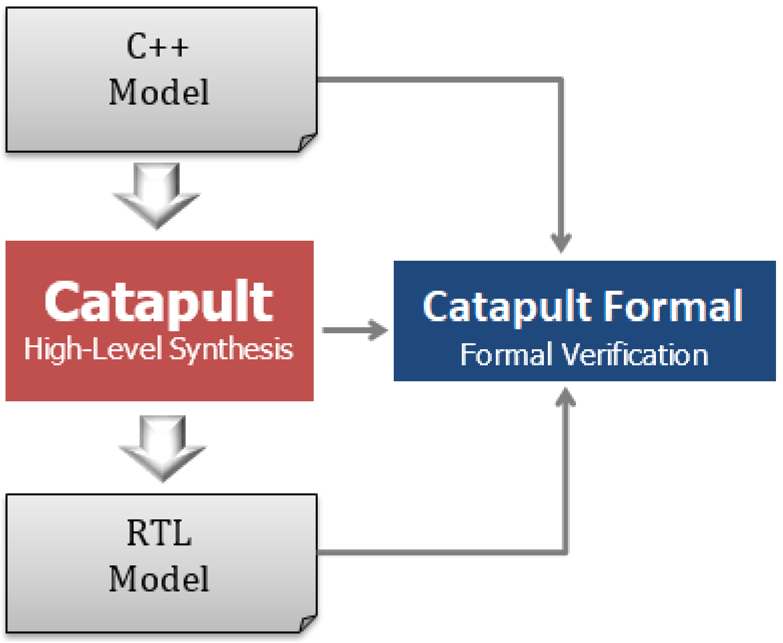
Typical users of the Catapult HLS platform start with a C++ model that describes the functional aspects of design intent. C simulation is fast and thus the model can be tested quite thoroughly. With the help of the Catapult HLS platform, you can synthesize from this well tested C model, an efficient micro-architectural representation that meets design constraints.

The HLS flow dramatically reduces the time and effort in realizing optimized hardware. However, you are still left with the validation overhead of running simulation vectors on the RTL implementation to establish coverage. This is often time and resource intensive. A formal equivalence solution will largely address this verification issue, but the wide abstraction (timing and design) gap between the C++ model and the synthesized RTL design has historically made this approach challenging and unscalable in most real world scenarios. Catapult Formal's CFormalSLEC function enables you to save time and computing resources during verification from designing at a higher level.

CFormalSLEC squarely addresses this validation challenge. CFormalSLEC is a dedicated application that provides a formal guarantee that RTL generated by Catapult is functionally the same as the C++ design entry point. Specifically, it proves that, beginning in a reset state, the two designs will generate the same output for any possible input sequence of any length. As part of this process, CFormalSLEC creates a wrapper around the top-level function in the C++ design; the wrapper then calls that top-level function repeatedly.

In addition to the C/RTL formal equivalence function, the Catapult Formal support includes specific verifications for functions added or inferred in the RTL which were not present in the untimed C++ source. These functions are stall verification for wait controllers, CFormalStall, verification of the the idle detection signal, CFormalIdle, and verification of user defined memory scheduling, CFormalIMP.

**Figure 1. Basic Catapult Formal Flow**



By leveraging the rich collection of formal verification strategies in the classic SLEC flow, and by developing a novel approach to gathering optimization hints during the synthesis process, CFormalSLEC presents a scalable and user-friendly approach.

In addition to establishing equivalence, you can employ Catapult Formal® to perform formal property checks both on C models and the corresponding RTL designs. You can enable Catapult Formal built-in checks for design defects such as checks for array bounds violations, use of uninitialized variables, and divide-by-zero. You may also use Catapult Formal to verify assertions on the C++ model and/or verify SVA properties on the RTL implementation, and establish verification coverage.

Catapult Formal presents a comprehensive formal verification platform aimed at addressing the "long pole" in design verification within the HLS design flow. Historically, this has been a tough task, given the timing and architectural gap between the specification reference C++ model and the synthesized RTL implementation. Catapult Formal has developed key IP to build a novel approach to bridging this abstraction gap and provide you a scalable and user-friendly mechanism to establishing formal equivalence between C++ models and their synthesized counterparts.

## Assumptions

This manual assumes that you have access to the Catapult HLS platform and have basic proficiency navigating its UI to build RTL designs. Basic interactions with the Catapult Formal flow can be conveniently performed via the Catapult GUI. CFormalSLEC is the equivalence verification function of Catapult Formal.

This manual is intended to give you the necessary tools to be quickly productive using CFormalSLEC in your design and verification flow. As a result, this manual does not go into details on how SLEC in general addresses the sequential equivalence problem.

If you are interested in understanding the foundational theory behind transaction-based sequential equivalence verification, refer to the *Sequential Logic Equivalence Checker (SLEC) User's Manual* and the accompanying training material.

## CFormalSLEC Flow Overview

The CFormalSLEC flow is designed with three metrics in mind: capacity, usability, and debug.

The following describes each of the three metrics:

- **Capacity** — Designs built using the Catapult HLS flow typically describe hardware for compute-intensive tasks. It is not uncommon for such designs to employ a large collection of arithmetic units. CFormalSLEC is quite capable of handling such "arithmetic-heavy" designs and not allowing them to undermine the verification capacity of the tool.

- **Usability** — In real-life design flows, establishing formal sequential equivalence between design models is often an involved task, attributed primarily to the sequential timing and architectural differences between them. Internal mapping relationships help rein in the verification capacity issues, but identifying these maps can be effort intensive, rendering the process unscalable. This problem is compounded in the context of HLS flows due to the wide abstraction gap between design entry and synthesized hardware.

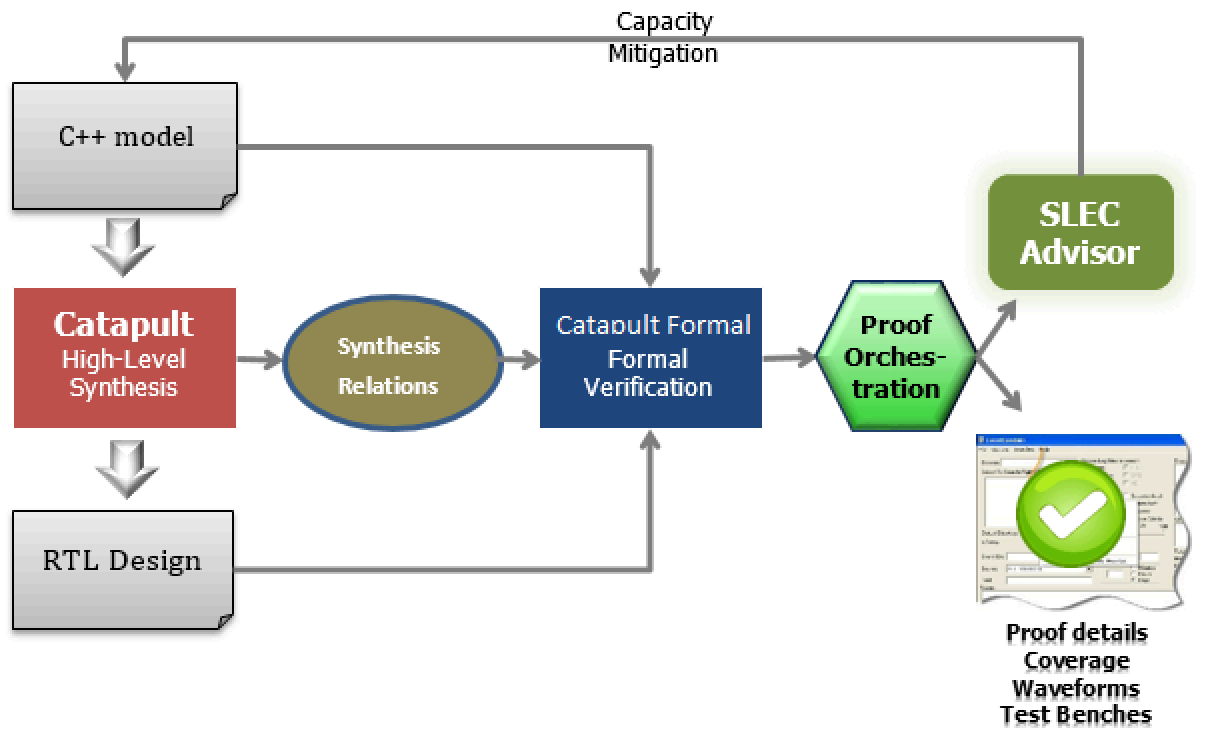
  The CFormalSLEC flow addresses this challenge by automatically deriving these mapping relationships and other useful information from hints generated by the Catapult HLS platform during the synthesis process. You can thus look forward to an out-of-the-box experience using CFormalSLEC as all verification collateral is handled automatically by the flow.

- **Debug** — The CFormalSLEC flow provides clear information on how the verification has proceeded and the list of verification points that were verified. You can access verification coverage information of a verification run to better understand the effect of constraints on the verification scope. CFormalSLEC has mature debugging capabilities via test benches and waveform traces. This allows you to effectively analyze verification failures and provide fixes. CFormalSLEC may suffer from capacity issues on certain verification problems. In such situations, the flow provides documentation on the source of the capacity issues and will suggest to you possible ways to alleviate the problem by proposing custom verification strategies and sometimes necessitating re-synthesis.

Figure 2: CFormalSLEC Flow Diagram provides a closer look at the CFormalSLEC flow.

For CFormalSLEC to be effective, you are expected to run Catapult in the "Formal Verification" mode. Under this mode, the underlying synthesis platform, in addition to building the RTL implementation, records a mutually agreed upon set of synthesis relationships. This set forms the basis of the automatic verification setup that CFormalSLEC generates. A large part of the synthesis relationships are the "probes" that are synthesized by Catapult as a side logic to relate the values of various variables and control structures to the abstract C++ variables and state. A preprocessing step in the CFormalSLEC flow reads in the C++ model, the synthesized RTL design and the set of synthesis relationships, and generates a comprehensive verification setup that includes the timing information (throughput and latency), and a collection of internal maps between the C++ model and the RTL design.

**Figure 2. CFormalSLEC Flow Diagram**



Because the verification setup is derived from hints provided by Catapult, CFormalSLEC cannot assume it to be correct. As part of its contract to establish a formal equivalence verification between the C++ model and the RTL design, CFormalSLEC is obligated to formally prove the correctness of the verification maps in order to preserve the sanctity of the assume-guarantee reasoning.

The CFormalSLEC flow employs the underlying SLEC proof engine orchestration to efficiently and effectively prove all the verification objectives. When encountering verification complexity that prevent its proof engines from converging, the SLEC Advisor accurately pinpoints the source(s) of the complexity. This takes the guess work out of the debug cycle, and you can now efficiently direct your efforts to mitigating verification complexity.

The following example outlines the basic concepts of the CFormalSLEC flow and how the verification task is partitioned into manageable chunks.

# Scope of Verification in Catapult Formal

It is important to know the scope of verification in CFormalSLEC.

## Block Level Verification

CFormalSLEC supports block-level verification of C++ models against generated RTL, which may be synthesized in either top-down or bottom-up manner. For designs containing multiple levels of block-type hierarchies, CFormalSLEC only verifies leaf-level blocks. For CCORE-type blocks, CFormalSLEC performs full hierarchical verification. See Flow Limitations.

### Design Entry

CFormalSLEC only supports the verification of C or C++ model and their counterpart RTL designs. Designs written in SystemC are not officially supported.

## A Design Example

A simple design example helps illustrate the CFormalSLEC flow.

Consider the following simple C++ function (*accum.cpp*) that describes the iterative accumulation of values in an array. The aggregated sum is then added to a counter that is incremented each time the function is invoked. The counter and the running sum are static types and retain their values across multiple invocations of the function.

*accum.cpp*:

```
int accum(int *A, size_t N) {
static int ctr = 0, sum = 0;
for (auto i = 0; i < N; ++i) {
 sum += A[i];
}
return sum + ctr++;
}
```

CFormalSLEC works natively at the SystemC level. Hence the first step to building the verification setup involves wrapping the C function with a light-weight SystemC wrapper to provide an interface. CFormalSLEC automatically generates this wrapper based on the interface information it has extracted from the RTL design that Catapult has synthesized. The snippet of code below captures the essence of the wrapper's role.

The SC_METHOD doit() effectively emulates the behavior of hardware by infinitely executing the accum() function.

*accum_wrapper.h*:

```
class accum_wrapper: public sc_module
{
 public: sc_in<bool> clk;
         sc_in<bool> rst;
         ...
         void reset()
         void doit();
         ...
         accum_wrapper() {
           SC_METHOD(doit);
           Sensitive_pos << clk;
         }...
}
```

*accum_wrapper.cpp*

```
 ...
void accum_wrapper::doit()
{
 if (rst) { reset() };
 else {
       ...
       accum(A, N)
      }...
}
 ...
```

The snippet of code in the above figure illustrates SLEC's API to the user C++ model. In the CFormalSLEC flow, this SystemC wrapper is generated automatically by a preprocessing step. If you are curious about the underlying process, read the companion SLEC Tutorial documentation for SystemC and Verilog for details and more examples.

The Catapult HLS platform gives you the freedom to specify several micro-architectural optimizations, pipelining of loops being one of them. In the function above, based on performance constraints, you have three options as far as loop pipelining is concerned. You can skip the pipelining step, or opt to pipeline the top loop in the SC_METHOD doit() or specify finer grained pipelining inside the accum() function. The CFormalSLEC verification setup for each of these pipelining scenario will vary significantly. Let us look closely at one of the scenarios where no pipelining is specified to understand how CFormalSLEC partitions the verification problem based on the concept of Catapult basic blocks.

During the Scheduling step, Catapult partitions the accum() function into three basic blocks, labeled A, B, and C as shown below. CFormalSLEC establishes a notion of transaction boundaries, which basically follows the program control flow at the basic block level. In the code snippet below, there are three possible transaction boundaries: A->B, B->B (signifying a loop) and `B->C`.

*accum.cpp*

```
int accum(int *A, size_t N) {
A: static int ctr = 0, sum = 0;
Loop: for (auto i = 0; i < N; ++i) {
 B: sum += A[i];
}
C: return sum + ctr++;
}
```

Catapult provides a list of all the variables that hold state across transaction boundaries. In this example, the variables ctr and sum hold state across all transaction boundaries, and variable i holds state across transaction boundaries A->B and B->B.

Catapult guarantees the preservation of the RTL logic in the synthesized implementation that correspond to the C variables that hold state across transaction boundaries. It achieves this by attaching data probes (side logic with no fanout) to the corresponding RTL registers. The preprocessing step in CFormalSLEC identifies these data probes and appropriately translates these Synthesis hints to intermediate maps in CFormalSLEC's parlance.

Recall that CFormalSLEC is attempting to establish functional equivalence between two models that differ widely in their latencies. Catapult provides synthesis hints by way of control probes that synchronize the execution of the basic blocks in the C++ model and the corresponding portions of the RTL. These control probes provide the control conditions under which each basic block should be executed. These boolean conditions are used as additional intermediate maps to align the C++ model and its RTL implementation.

In the next chapter, you will gain greater insights into the CFormalSLEC flow and the various steps that the flow automatically executes as part of the verification setup.

# Chapter 2
# Running the CFormalSLEC Flow

This chapter provides an overview of the entire CFormalSLEC flow, including preparing your design, generating the required setup files, running verification, and debugging any falsifications. It includes actions you must take when synthesizing your design in Catapult to ensure successful verification. This chapter also includes a description of the output logs generated during verification.

## Preparing Your Design for Verification

Before running CFormalSLEC, you must ensure your specification and implementation designs are ready for verification. This section describes the steps required to prepare your design.

Run Catapult in “Formal Verification” mode. In this mode, Catapult outputs synthesis relations required by CFormalSLEC.

In addition, you must ensure that your C++ model is correct. CFormalSLEC formally establishes functional equivalence between the C++ model and its synthesized RTL implementation. It assumes that the C++ model is correct and represents the intended behavior. CFormalSLEC therefore may not catch errors in the C++ model. Before running CFormalSLEC, run simulation vectors on the C++ model and validate the functionality.

In Catapult, run CDesignChecker to check for all properties related to synthesis-simulation mismatch. These properties check for C++ defects such as array bounds violations (ABR for read violations/ ABW for write violations), uninitialized memory reads (UMR), illegal shift amounts (ISE), incomplete switch-case statements (CAS), and divide-by-zero (DBZ). See *Catapult® Synthesis User and Reference Manual* for more information about CDesignChecker.

### Preparing Custom Memories

If your design contains memories, you must create memory libraries for your design before synthesizing in Catapult. Catapult provides a Memory Generator tool to read in an HDL memory model, set various parameters, and create the memory libraries. For CFormalSLEC to perform the proper mappings to memory, you must set the VHDL Array Path and Verilog Array Path parameters to the relative path of the memory within the model.

For example, suppose you have the following Verilog model provided by your memory vendor:

```
module ram_sync_dualRW_be ( data_in, addr, re, we, data_out, clk, a_rst,
 s_rst, en);

parameter words = 'd16;
parameter width = 'd16;
parameter addr_width = 4;

localparam byte_width = width / num_byte_enables;
parameter no_of_RAM_dualRW_readwrite_port = 2;
localparam byte_width = width / num_byte_enables;

reg [width-1:0] mem [words-1:0];

integer i;
  generate
    always @(posedge clk)
    begin
      if ( en == enable_active )
      begin
        for (i = 0; i< num_byte_enables; i = i + 1)
        begin
          if ( re[i] == re_active )
            data_out[i*byte_width+: byte_width] <= \
               mem[addr][i*byte_width+:byte_width];
          if (we[i] == we_active)
            mem[addr][i*byte_width+:byte_width] <= \
               data_in[i*byte_width+:byte_width];
        end
      end
    end

endmodule
```

In Catapult Memory Generator, you would set Verilog Array Path to “mem” because it is instantiated directly under the ram_sync_dualRW_be module. If the memory model contained a different hierarchy, you would specify the path relative to the top-level memory module.

If you use the Catapult library builder, you must set the VerilogArrayPath and VHDLArrayPath variables in the memory component as follows:

```
variable( "VerilogArrayPath"; type: string ) := "mem";
variable( "VHDLArrayPath"; type: string ) := "mem";
```

For more information about Memory Generator, see “Creating Memory Libraries” in *Generating Libraries for Catapult*.

## Catapult in CFormalSLEC Mode

The Catapult tool must be set up and run in CFormalSLEC mode.

To ensure compatibility with the CFormalSLEC flow, the Catapult setup file must be set to run in the Formal Verification mode.

Observe a sample setup file shown below. The lines that are boldfaced enable this mode in Catapult. These commands should be specified before any of the synthesis steps are performed. If you fail to ensure this, this mode will *not* be enabled and subsequently the CFormalSLEC flow will fail.

The first two lines (lines 4 and 5) are flow commands. By enabling the CFormalSLEC flow, these commands signal to Catapult that the synthesis relations and the associated wrappers need to be generated for CFormalSLEC at the end of the high level synthesis process. The two directives that follow (lines 6 and 7) enable a synthesis flow that is more formal verification friendly.

*setup.tcl*

```
 1   solution new -state initial
 2   solutions options default
 3   flow package require /SCVerify
 4   flow package require /CFormalSLEC
 5   directive set -FORMAL_VERIFICATION true
 6   directive set -TRANSACTION_DONE_SIGNAL true
 7   solution file add<c files>
 8      <other directives>
 9   go analyze
10   go compile
11   go assembly
12   go architect
13   go extract
```

Running Catapult in the FORMAL_VERIFICATION mode achieves two important goals. It enables Catapult to output the synthesis hints for CFormalSLEC. It also signals Catapult to disable a small collection of synthesis optimizations that have the potential to undermine the effectiveness of CFormalSLEC's verification strategies and thereby its convergence. Furthermore, Catapult generates side logic for probing the RTL design. The probe logic does not drive any logic and is expected to be optimized away by the downstream logic synthesis tools.

You can now focus on generating an RTL implementation that meets design constraints. Once the RTL design is fully synthesized by Catapult, you are now ready to run CFormalSLEC to verify the functional equivalence of the generated RTL and its reference model.

# Flow Limitations

The CFormalSLEC flow has certain limitations. These limitations may be removed in future releases.

## Language Constructs

CFormalSLEC does not support SystemC designs. For C++ designs, the following restrictions apply:

- Conditional return/continue statements in rolled loops are not supported.
- Usage of size API on ac_channels is only supported when checking if the channel is empty.
- CFormalSLEC models all non-blocking reads and writes with a return value of 'true.' It does not consider 'false' or 'then' branches of conditional statements with non-blocking reads and writes.

## Hierarchies

### Block Type Hierarchies

CFormalSLEC only verifies the leaf level blocks; any logic or interconnect between blocks is not verified. For block-type hierarchies synthesized with a bottom-up flow, you must run verification for each leaf-level block by running CFormalSLEC in the corresponding solution directory. For block-type hierarchies synthesized with a top-down flow, specify the -block option with the Setup Script: cat2slec.

For function-based block type hierarchies, the top-level functions must not be templatized. To ensure compatibility with CFormalSLEC, place any required instance of a templatized function inside a top-level wrapper function.

For C++ class-based hierarchies, leaf level classes must have only one process method with an interface; they should not contain any other hierarchy blocks.

### CCORE Hierarchies

CCORE hierarchies can be synthesized in a bottom-up or top-down manner. CFormalSLEC fully verifies CCOREs, including their instantiation in the block and any other nested CCOREs. If a CCORE contains state-holding elements such as static variables, you must perform bottom-up verification.

Perform top-down verification with a single run of CFormalSLEC. Perform bottom-up verification with separate CFormalSLEC runs for each of the CCOREs and corresponding hierarchies in which they are located.

Whether CCOREs are verified in a top-down or bottom-up manner has no relation to how they were synthesized.

CCOREs with variable latency are currently not supported.

## Interface Components

CFormalSLEC does not support inout ports.

CFormalSLEC supports the interface components for ANSI C++, Algorithmic C (AC), and array datatypes, and ac_channel handshaking interfaces, with the following restrictions:

### ac_channel

Only the following ac_channel interface components are supported:

- mgc_ioport.mgc_in_wire_en
- mgc_ioport.mgc_in_wire_wait
- mgc_ioport.mgc_out_stdreg_en
- mgc_ioport.mgc_out_stdreg_wait
- mgc_ioport.mgc_out_buf_wait
- mgc_ioport.mgc_out_fifo_wait
- mgc_ioport.mgc_chan_in
- mgc_ioport.mgc_out_prereg_en
- ccs_ioport.ccs_in
- ccs_ioport.ccs_out
- ccs_ioport.ccs_in_wait

- ccs_ioport.ccs_in_wait_coupled
- ccs_ioport.ccs_out_wait
- ccs_ioport.ccs_in_rdy
- ccs_ioport.ccs_out_vld
- ccs_ioport.ccs_out_pipe
- ccs_ioport.ccs_chan_sync
- ccs_ioport.ccs_sync_in_wait
- ccs_ioport.ccs_sync_out_wait

**Note:**
Arrays of ac_channel datatypes are not supported.

Any other ac_channel interface components in the design result in a CTS-NAMAS error as described in Catapult Flow Compatibility Errors.

### ANSI C++, AC Normal, and Array Datatypes

For ANSI C++, AC (normal), and array datatypes, CFormalSLEC supports all Catapult interface components except for the following:

- All mgc_ioport.mgc_inout_* components, such as mgc_inout_buf_wait, mgc_inout_fifo_wait, mgc_inout_prereg_en, mgc_inout_stdreg_en, and mgc_inout_stdreg_wait.
- mgc_ioport.mgc_out_reg_wait component.

If you are running CFormalSLEC in stall verification mode, then the following three components are also prohibited:

- mgc_ioport.mgc_out_buf_wait
- mgc_ioport.mgc_out_fifo_wait
- ccs_ioport.ccs_out_pipe

## Loops

Loops are transformed significantly during Catapult synthesis. The following transformations on loops are not supported by CFormalSLEC:

- Loop merging
- Partially unrolled loops

- I/O during reset loop (all I/O operations must occur in the main loop)
- Loops that appear both rolled and unrolled in different code paths

## Memories

CFormalSLEC efficiently models memories using an abstract representation during formal verification. However, large arrays of structs or arrays inside structs may lead to runtime problems during verification.

Catapult may optimize memory layout features for efficient modeling purposes. The following memory layout features are not supported by CFormalSLEC:

- Unused bits left while packing
- Memories with unused columns that Catapult optimized out of the design during synthesis
- Register files

**Note:**
You can use the less efficient memory representation if you run into these limitations; however, note that this may have an adverse impact on verification capacity.

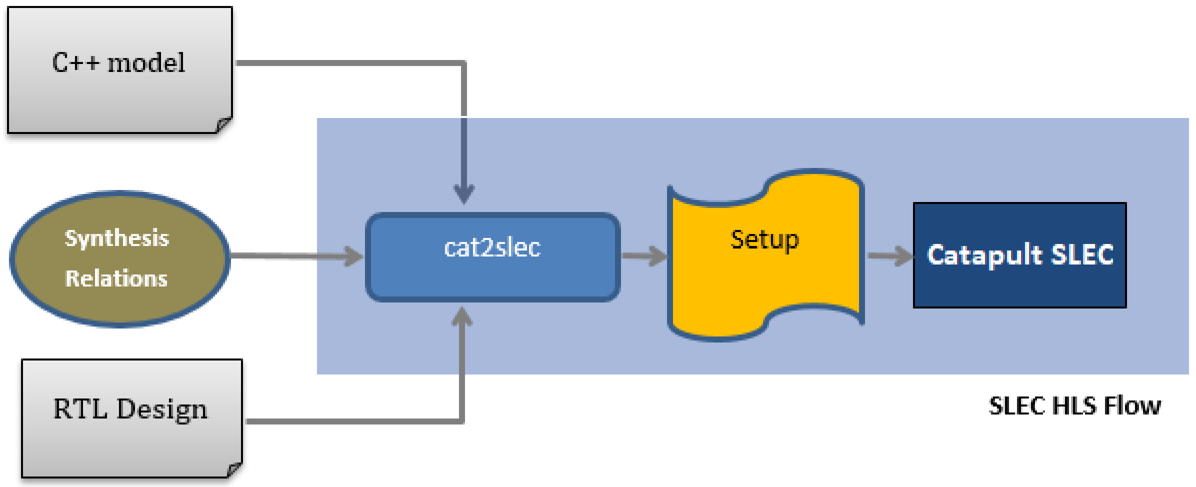
## CFormalSLEC Setup

Now that you have successfully synthesized an RTL implementation that meets design constraints, you are ready to generate the setup files. A large factor in the verification complexity is the completeness of the verification setup. The setup can be quite involved when the models considered for verification are far apart from a design abstraction point of view. The preprocessing step solves this problem by automating the generation of an efficient verification setup.

A preprocessing step called cat2slec takes the specification (the C++ reference model), the implementation (the synthesized RTL design), and the Synthesis relations that Catapult has generated, and builds the verification collateral.

This setup captures the key internal mapping points, the timing differences, and the inductive invariants that relate the implementation to the specification. Critically, the inductive invariants allow for the equivalence problem to be formulated as a combinational exercise to the SLEC verification engines. This enables the CFormalSLEC flow to provide a highly scalable verification solution.

**Figure 3. CFormalSLEC Preprocessing Flow**



The entire CFormalSLEC flow is conveniently encapsulated such that you can execute this flow completely within the Catapult GUI environment. You can also choose to invoke CFormalSLEC on the command line, outside of the Catapult environment. The latter approach is better when you are looking for added flexibility by way of additional debug switches to combat verification complexity.

## Generating Verification Setup Files

Generate the required setup files for verification.

### Prerequisites

- You have synthesized a design in Catapult as described in Catapult in CFormalSLEC Mode.

## Procedure

1. Navigate into the CFormalSLEC project directory:

   ```
   % cd <project_name>/<solution_name>.v<n>/SLEC/
   ```

2. Make any desired customizations to the CFormalSLEC setup files described in Verification Setup Files and Control Variables.

3. Run the *cat2slec.sh* script to generate the verification setup. See Setup Script: cat2slec.

## Results

You have successfully generated the setup files required to run verification.

# Setup Script: cat2slec

The cat2slec script builds the verification setup after reading in the specification and implementation designs, along with the synthesis relations generated by Catapult.

## Usage

cat2slec **{vlog|vhdl}** [autoverify] [-bup] [-block *block_name*] [-proof_depth *depth*]
[-disable_symbolic_memory] [-no_isolate_ccore] [-no_wrap] [-pbp] [-noccore] [-verify_stall]

## Arguments

- **{vlog|vhdl}**

  This required argument specifies whether the design target is written in Verilog (vlog) or VHDL (vhdl).
- -help

  Shows all the help options present in cat2slec.
- -help *command*

  Shows specified command help if available. For example:

  ```
  cat2slec -help -proof_depth
  ```
- autoverify

  Runs CFormalSLEC verification automatically after generating the setup.
- -bup

  Enable ccore bottom-up verification flow.
- -block *block_name*

  Specifies the hierarchy block to prepare for verification. The cat2slec script generates the C-to-RTL verification script only for leaf-level blocks. The script generates RTL idle verification scripts for all types of hierarchical blocks.

  **Note:**
  You can see a full list of block names by running cat2slec without the -block argument.
- -proof_depth *depth*

  Specifies the proof depth for CFormalSLEC in BEC mode
- -weaken_maps_for_umr

  Weaken the maps generated by cat2slec for handling UMR-related falsifications. Choosing this option over initializing the variables (as recommended) may result in degraded performance.
- -disable_symbolic_memory

  Do not use symbolic memory abstraction for verifying memories. Note that this can lead to capacity and runtime issues in CFormalSLEC.
- -noccore

Do not break the verification problem at CCORE port boundaries. Using this option can result in capacity problems while running CFormalSLEC.

- -no_isolate_ccore

  CCORE will not be verified in isolation.

- -no_wrap

  Skips generation of SystemC and RTL wrappers.

- -pbp

  Enables path-by-path verification.

- -verify_stall

  Generate scripts for verifying the stall functionality of Catapult-generated RTL if it contains streaming interfaces with handshakes, such as ac_channels.

## Examples

### Example 1

The following example shows the most common usage to generate verification setup.

```
cat2slec.sh <vlog|vhdl>
```

### Example 2

The following example generates verification setup for Verilog and invokes CFormalSLEC automatically.

```
cat2slec.sh vlog autoverify
```

### Example 3

The following example generates verification setup for VHDL and configures the setup for bottom-up verification of CCOREs.

```
cat2slec.sh vhdl -bup
```

# Verification Setup Files and Control Variables

The CFormalSLEC setup consists of files generated by CFormalSLEC and user-specified files.

## Generated Setup

CFormalSLEC automatically generates setup files that are required as input for verification. These files provide information about the mapping between the two models, timing differences, and other relevant design information.

The table below lists the files generated by CFormalSLEC and a brief description of each. All generated files are in the CFormalSLEC work directory.

**Table 1. Generated Setup Files**

| Filename | Description |
|---|---|
| *run_slec_hls.sh* | Shell script to invoke cat2slec and then SLEC. |
| *cat2slec.sh* | Shell script to create SLEC wrappers and TCL files. |
| *slec_vlog.tcl*<br>*slec_vhdl.tcl* | The setup Tcl file for SLEC when verifying Verilog or VHDL RTL respectively. |
| *ccs_slec_v.db*<br>*ccs_slec_vhdl.db* | Catapult DB used by cat2slec to create SLEC setup for verifying Verilog or VHDL RTL respectively. |
| *spec_wrapper.cpp* | The wrapper used for the C++ source. This defines the reset, read, write, and main function being verified. |
| *spec_wrapper.h* | The wrapper header file used for the C++ source. This file defines the IO interface, internal wrapper connection signals and the SystemC constructor. |
| *impl_wrapper.v*<br>*impl_wrapper.vhdl* | The design wrapper used for the RTL output respectively in Verilog and VHDL. |
| *design_c2rtl_vlog.db*<br>*design_c2rtl_vhdl.db* | C++ and Verilog/VHDL design DB generated by SLEC during cat2slec run. This is used during subsequent verification. |
| *calypto_xtors* | This folder is generated by cat2slec to store various transactors required to map C++ and RTL, and for certain design invariants. |
| *path_slec_vlog.tcl*<br>*path_slec_vhdl.tcl* | The setup Tcl file for individual paths in SLEC in path-by-path verification for Verilog and VHDL RTL respectively. |
| *run_paths.sh* | Shell script to run all the paths in path-by-path verification in sequence. |
| *run_paths.tcl* | Setup Tcl file for running all the verification paths using distributed processing systems, such as LSF. See Distributed Processing in SLEC in the *SLEC User's Manual*. |

The top-level scripts you need to run CFormalSLEC are *run_slec_hls.sh*, *cat2slec.sh*, and *slec_<vlog|vhdl>.tcl*

## User-Specified Setup

The setup also includes user-specified files read in by the top-level script. You can adjust the verification flow by modifying the contents of these files. This could be, for instance, to generate an optimized build for a specific verification problem, or to provide constraints and additional maps to address verification capacity issues. These files should reside at the same level as the Catapult <*project*> directory.

**Table 2. User-Specified Constraints Files**

| Filename | Description |
|---|---|
| *slec_pre_build.tcl* | Specify pre-build_design constraints in this file. This is sourced in the cat2slec-generated SLEC setup. |
| *slec_constraints.tcl* | Specify I/O constraints or any post-build design constraints in this file. This is sourced in the cat2slec-generated SLEC setup. |
| *slec_constants.tcl* | Specify constant constraints in this file. If present, cat2slec automatically reads it and constrains the ports with constant values in the wrapper itself. |
| *flop_maps.tcl* | Specify user-specific intermediate maps in this file. This is sourced in the cat2slec-generated SLEC setup. |

### Verification Control Variables

The CFormalSLEC setup allows for user-specified customizations to the verification flow. To make customizations, set the desired control variables identified in the table below. The preferred method is to set these variables in the *slec_pre_build.tcl* control file. CFormalSLEC reads in this file during the setup process and includes the specified settings in the generated setup.

**Table 3. Control Variable Definitions**

| TCL Variable | Default Value | Definition |
|---|---|---|
| Enable_DB_Flow | 1 | Use existing database previously generated by *cat2slec.sh*. Set to 0 to accommodate minor changes in the design without having to re-run *cat2slec.sh*. A value of 0 specifies that SLEC repeat the design build during verification. |
| Enable_Memory_Maps | 1 | Perform verification of memories. Set to 0 when debugging other parts of the design. |
| Enable_CCore_Maps | 1 | Use divide and conquer strategy around CCOREs to speed up verification. Set to 0 only if there is a problem setting up hierarchical CCORE verification (though performance will likely worsen). See CCORE Hierarchies. |
| Enable_Coverage_Maps | 0 | Enable formal coverage analysis of both C++ and RTL designs. Coverage analysis runs in conjunction with equivalence and property checking. See *SLEC Coverage* Application Note. |
| Enable_Source_Code_Analysis | 0 | Perform formal coverage analysis only of the C++ code. Do not perform equivalence or property checking. This is useful for finding unused code in the C++ design. |
| Proof_Mode | FP | Sets the proof mode. Default value is full proof (FP). Set this variable to BEC to perform |

**Table 3. Control Variable Definitions (continued)**

| TCL Variable | Default Value | Definition |
|---|---|---|
| | | bounded verification up to Proof_Depth number of verification transactions. |
| Proof_Depth | 10 | Specifies the maximum number of transactions to perform in bounded verifcation mode. |

### Examples

The following code shows an example *slec_pre_build.tcl* file:

```
## USE FSDB format for waveform files
config_trace_files -format fsdb

## SLEC shouldn't stop after finding a falsification
set_global stop_at_first_falsification 0

## Enable formal coverage flow
set Enable_Coverage_Maps 1

## Enable UMR checking
config_property_checks -enable -umr

## Distribute computing setup to speed up verification engines
set_global max_local_async_workers 2
set_global max_remote_async_workers 10
set_global host_setup_configuration "qsub -q GPU_project -o stdout_%RJC.log
 -l OpSys=redhat[67].*"
set_global enable_solver_async_subsessions 1
set_global enable_seqsat_async_subsessions 1
```

The following code shows an example *slec_constraints.tcl* file:

```
## range of values on rows direct input
set_symbolic_constant -inputs {spec.slecIN_rows_rsc_idx_0
 impl.slecIN_rows_rsc_idx_0} -values {1 - 80}

## assume all ABR violations not to occur (index expressions are within
 bound)
## Use with due care, otherwise can lead to false proofs
assume_properties -prop -abr
```

The following code shows an example *slec_constants.tcl* file:

```
# set the X dimension of incoming images to be a constant
set_constant -spec -module dct_quantize_one_block -value 412
 image_in.size_x
set_constant -impl -module dct_quantize_one_block -value 412
 image_in.size_x
```

The following code shows an example *flop_maps.tcl* file:

```
## Added a probe manually in C++, mapped it to Catapult synthesized version
create_map -intermediate spec.probe_image_type impl.DUT.image_type_acprb

## Weaken the output checking by a valid condition
create_map -output spec.out impl.out -valid impl.DUT.image_valid
```

## CFormalSLEC Verification

This section describes how to run verification, and also provides details about the verification process, including the back-end solver stack and the high-level verification strategy.

After performing the setup procedure described in Generating Verification Setup Files, execute the following command:

```
% $SLEC_HOME/bin/slec ./slec_<vlog|vhdl>.tcl
```

With the two models, the specification and the implementation, formally represented as a one-transaction unit, CFormalSLEC concurrently builds a list of proof obligations—primary output pairs from the two models and all intermediate mapping points identified as part of generating the verification setup from Catapult synthesis relations.

SLEC employs a sound assume-guarantee methodology to satisfy its proof obligations. Additionally, CFormalSLEC strives to build a fully inductive setup so that it can conduct most of the formal analyses and verification of the design at the word level. This translates to improved tool capacity and greatly extends the tool's capability to handle large compute-intensive designs. CFormalSLEC has a carefully-crafted, third generation word-level verification engine with a light-weight deductive apparatus and a powerful collection of decision procedures. This engine is called L1.

Some maps (control maps, for instance) you can verify more efficiently at the bit-vector level. For this family of verification problems, we have the L3 verification engine. This engine employs a combination of design transformations, SAT-based techniques, and other bit-level approaches to solve the verification problems.

Both verification engines L1 and L3 are "full verification" engines. They are exhaustive in nature, and when they converge, they provide a full proof. There is a third verification engine, L2, that provides a bounded proof of correctness. This engine is formal and exhaustive in nature but only up to the bound you specify. The notable exception is when the verification engine L2 can determine the diameter of the verification problem and thus formally interpret and log the bounded verification as full verification.

There are low- and high-effort versions of these verification engines. The low-effort versions, as their name implies, skip some of the solver strategies and impose a time limit on each proof obligation. If they are unable to verify a specific problem (map) within a prescribed time limit, that verification session is aborted and the low-effort engine moves on to the next verification problem. This gives you a quick summary of the verification problems (maps) that can be verified with low effort. This enables the CFormalSLEC verification flow to focus on the harder problems and only use the high-effort engines where necessary. The CFormalSLEC flow has a default recipe where it runs the low effort engines with a pre-defined time limit. This recipe works quite well for a wide variety of workloads. However, you are free to customize this order and can skip, re-arrange, or emphasize any or all of these engines. For a more detailed description of the verification engines, see Verification Engines in the *SLEC User's Manual.*

The verification manager orchestrates these engines and the order in which you want to execute them, keeping track of all verification obligations, and ensuring that the assume-guarantee is sound and that the flow addresses all outstanding proof obligations.

CFormalSLEC supports distributed processing for verifying maps in parallel. You can adjust the level of parallelism with specific global variables. See “Distributed Processing in SLEC” in the *SLEC User’s Manual*.

Each of these verification engines, L1, L2, and L3, have access to a wide range of solvers. A light-weight automated theorem proving framework, SMT solvers, intelligent simulation engines, and a family of SAT solvers all work in parallel to discharge a verification problem. The solver that converges the fastest, returns the result to its Engine manager, which then records the result, resets the state of all the solvers, and advances to the next verification problem (map).

**Note:**
Appropriate licensing is required to make use of distributed processing in CFormalSLEC.

The default settings are recommended for the verification engines when you run CFormalSLEC for the first time on a design. You can choose to customize the flow based upon a careful observation of the log files and identifying bottleneck areas. You can re-invoke CFormalSLEC with a fresh set of global values. Table 4: Customization of Low Effort Verification Engines lists the common SLEC global variables you can modify to customize the verification flow.

**Table 4. Customization of Low Effort Verification Engines**

| SLEC Global | Description |
|---|---|
| seq_enable_ll1_engine | Enable Low effort L1 in the verification flow |
| seq_enable_ll3_engine | Enable Low effort L3 in the verification flow |
| ll1_problem_time_limit | Time limit for each LL1 verification run |
| ll3_problem_time_limit | Time limit for each LL3 verification run |

The first two globals enable the low effort L1 and low effort L3 engines. By default, both globals are enabled. If you want to disable one or both of them, that can be achieved by specifying these globals. The last two globals specify the time spent on each verification problem, and by default are set to 30 seconds each. You may choose to change this time limit at the time of invoking CFormalSLEC. A sample global assignment follows:

```
% Disable LL1 Engine
set_global enable_ll1_engine 0

% Assign new time limit of 180 seconds to LL3 engine
set_global ll3_problem_time_limit 180
```

Specify these globals in the *slec_pre_build.tcl* file. CFormalSLEC automatically picks up user customization from this file before commencing the verification task.

In addition to flow customization, you can enable the verification flow to discharge the proof obligations in parallel with distributed processing. See “Distributed Processing in SLEC” in the *SLEC User’s Manual*.

## CFormalSLEC Reports

The CFormalSLEC flow logs useful information about the verification run that gives you insight into the verification setup and proof status.

All files generated during a CFormalSLEC run are consolidated under the *calypto* directory. The verification run dumps a lot of pertinent information intended for users and experts enabling them to analyze the setup, proof progress and possibly complexity issues.

The following four files are useful in providing a high level summary of the setup and verification run and will usually highlight any unexpected issues in the verification setup:

- *slec.log*
- *reset.log*
- *mapping.log*
- *results.log*

## slec.log

The top level *slec.log* summarizes the CFormalSLEC verification run, keeping a running tab on the progress and alerting you to any anomalies in the flow that could potentially affect the verification outcome or overall tool capacity. It also notifies you about the verification status on the various proof obligations.

## reset.log

The *reset.log* file captures the reset values on the sequential logic in the specification and the implementation design. Unexpected X values on sequential elements may point to subsequent verification failures that needs to be analyzed. The following is a sample reset.log file:

```
# Flops with reset-value of 'X' are symbolically initialized by the formal
#  solvers.
# Mapped-flops which have reset-value of 'X' are initialized to the same
#  symbolic value as the flops they are mapped to (refer to 'mapping.log'
#  for the list of mapped-flops)
set_reset_value -list \
 impl.DUT.mmap.LOOP_COMPONENT_PIXEL_1_err_q_slc-value 'h0
set_reset_value -list \
 impl.DUT.mmap.LOOP_COMPONENT_PIXEL_1_p_conc_1_itm_1_7_0 -value 'h00
set_reset_value -list \
 impl.DUT.mmap.LOOP_COMPONENT_1_LOOP_PIXEL_1_p_conc_1_itm_1_8 \
 -value 'h0
set_reset_value -list impl.DUT.mmap.res_size_0_0_lpi_1_dfm_1 -value 'h0
set_reset_value -list spec.prop_abr_QuantDivisor_ln216 -value 'h0
set_reset_value -list spec.prop_abr_QuantDivisor_ln216_1 -value 'h0
set_reset_value -list spec.prop_abr_QuantOffset_CCS_ln314 -value 'h0
set_reset_value -list spec.prop_abr_QuantOffset_CCS_ln314_1 -value 'h0
set_reset_value -list spec.prop_abr_QuantOffset_CCS_ln314_2 -value 'h0
```

## mapping.log

The *mapping.log* file, as the name suggests, lists all the (spec, impl) mapping relationships CFormalSLEC has inferred from the synthesis relations that Catapult provided. Specifically, it provides mapping information for input, output, and intermediate candidates via SLEC's create_map command. The following is a sample *mapping.log* file:

```
## List of unmapped spec flops
# spec._Calypto_present_state_0 # (2 bits unsigned)
# spec.prop_abr_QuantDivisor_ln216 # (1 bit unsigned)

### List of unmapped impl flops)
# impl.CALYPTO_DUT.mmap.LOOP_COMPONENT_1_LOOP_PIXEL_1_err_q
# impl.CALYPTO_DUT.mmap.LOOP_COMPONENT_1_LOOP_PIXEL_1_p_conc_1_itm_1_7_0

## List of input-maps
create_map -input [create_waveform -hold_start 0 -hold_interval 3 \
 spec.slecIN_ctrl2enc_data_pmode_rsc_idx_0  ] \
 impl.slecIN_ctrl2enc_data_pmode_rsc_idx_0 ;

### List of output-maps
create_map -output -induct 1 [create_waveform -sample_start 1 \
 spec.slecOUT_enc2ctrl_data_prefixcg_rsc_idx_0] \
 [create_waveform -sample_start 7 \
 impl.slecOUT_enc2ctrl_data_prefixcg_rsc_idx_0] -valid \
 [create_waveform -sample_start 7 \
 impl.CALYPTO_DUT.enc2ctrl_data_prefixcg_rsc_triosy_lz];

### List of active intermediate-maps
create_map -intermediate -state_encoder impl \
 {spec.mmap@a_cpnt_prev[0][7:0]}
impl. DUT.mmap_core_inst.main_lpnxt_a_cpnt_prev_8_0_7_0_dtprb  -valid \
 spec.main_loop_valid

### List of properties
assume_properties -prop -induct 1 -list spec.prop_asc_ln214 ; # ( asc )
check_properties -prop -induct 1 -list \
 spec.prop_abr_QuantDivisor_ln216 ; # ( abr )
check_properties -prop -induct 1 -list \
 spec.prop_abr_QuantDivisor_ln216_1 ; # ( abr )
check_properties -prop -induct 1 -list \
 spec.prop_abr_QuantOffset_CCS_ln314 ; # ( abr )
check_properties -prop -induct 1 -list \
 spec.prop_ise_ln65_9 ; # ( ise )
check_properties -prop -induct 1 \
 -list spec.prop_opt_hints_mark_select_177 ; # ( opt_hints )

### List of coverage-targets
create_coverage_target [create_waveform -sample_start 0 \
 spec.block_valid_ln_11]
create_coverage_target [create_waveform -sample_start 0 \
 spec.block_valid_ln_68_2]
create_coverage_target [create_waveform -sample_start 0 \
 spec.block_valid_ln_68_3]
```

This file also lists the sequential elements in the spec and in the impl that do not figure in any mapping relation. You can choose to include additional mapping relationships if needed.

This file also lists the environmental and design properties assumed and part of the verification obligations of CFormalSLEC. When CFormalSLEC is invoked in coverage mode, the auto coverage targets, in addition to the user provided ones, will be listed here.

## results.log

The *results.log* file lists all the proof obligations listed in the *mapping.log* file that have been verified to be true or false. It also lists the status of the coverage targets. This file ends with a summary statistics that gives you the big picture of the verification run. A sample *result.log* is shown below. The sample output file shows properties, optimization hints from Catapult, output pairs, and coverage targets. Observe that the proofs are conditional in nature. This is because of the presence of an assume_only property specified by you in the specification. This property specifies an assumption on the environment and hence cannot be verified in this context. The summary of results gives you a dashboard view of the verification targets.

```
Format for non-falsified pairs:
[<status>] <problem-pair>: <spec> (<throughput>, <latency>) <impl> \
 (<throughput>, <latency>) <valid> (<latency>)
Format for falsified pairs:
[<status> <transaction-number>] <problem-pair>: <spec> (<throughput>, \
 <latency>) <value> <time> <impl> (<throughput>, <latency>) <value> \
 <time> <valid> (<latency>) <time>

[PROVEN CONDITIONALLY] Property: spec.prop_abr_l_ln99 ( PROP, abr )
[PROVEN CONDITIONALLY] Property: spec.prop_opt_hints_mark_select_323 \
 ( PROP, opt_hints )

[PROVEN CONDITIONALLY] Output-pair: \
 spec.slecOUT_enc2ctrl_data_prefixco_rsc_idx_0 (1,2) \
 impl.slecOUT_enc2ctrl_data_prefixco_rsc_idx_0 (3,10) \
 valid=impl. DUT.enc2ctrl_data (latency=10)
[PROVEN CONDITIONALLY] Output-pair: \
 spec.slecOUT_enc2ctrl_data_prefixy_rsc_idx_0 (1,2) \
 impl.slecOUT_enc2ctrl_data_prefixy_rsc_idx_0 (3,10) \
 valid=impl.DUT.enc2ctrl_data (latency=10)
[PROVEN CONDITIONALLY] Output-pair: \
 spec.slecOUT_enc2ctrl_data_suffix_sizecg_rsc_idx_0 (1,2) \
 impl.slecOUT_enc2ctrl_data_sizecg_rsc_idx_0 (3,10) \
 valid=impl.DUT.enc2ctrl_data (latency=10)

[COVERED AT 2] Coverage-target: \
 spec.INST_xtor_CPS_0.Calypto_transaction_done (1,0)
[COVERED AT 1] Coverage-target: spec.main_loop_valid (1,0)
[COVERED AT 1] Coverage-target: spec.block_valid_ln_444 (1,0)

Summary of key results:
========================================================================
************* Design Rule Violations (calypto/ccheck.log) **************
========================================================================
                   Errors   Warnings
Spec                    0         55
Impl                    0         83
========================================================================
******* Design Characteristics (calypto/characteristics.log) ***********
========================================================================
    Inputs/BBox-outputs  Outputs/BBox-Inputs  Inouts  WL-Flops   BL-Flops
Spec                 62                   23       0        165
 384
Impl                 61                   23       0        196
 1182
=========================================================================
****************** Mapping (calypto/mapping.log) ********************
=========================================================================
     Input-Maps     Output-Maps        Active Intermediate-Maps
             60             106                               5
     Inactive Intermediate-Maps     Unreachable Input-Maps     Memory-maps
                              0                          0               0
     Unmapped Inputs      Unmapped Outputs      Unmapped Flops
Spec               2                     1                 165
Impl               1                     0                 195
========================================================================
************************ Reset (calypto/reset.log) ******************
========================================================================
           Reset WL-Flops      Unreset WL-Flops
Spec                  165                     0
Impl                  196                     0
=========================================================================
************** Coverage Results (calypto/results.log) ****************
=========================================================================
       Proven-uncoverable  Covered  Not-coverable-upto-bound  Unresolved
                        0      498                         0          34
=========================================================================
*************** Equivalence Results (calypto/results.log) *************
=========================================================================
                      Proven Cond-Proven Bounded-Proven Falsified
 Unresolved
Output-Maps               0         106              0         0          0
Active Intermediate-Maps 0           4              0         0          0
Properties                0         139              0         0          0

The problem setup has 3 assume_only maps (see 'mapping.log' for full list).
```

In the case where CFormalSLEC reports a verification mismatch (falsification), you are provided with some debugging aids to enable you to dig further and find the root cause of the falsification. The next section introduces some of these debugging capabilities.

## Debugging Falsifications

In the ideal scenario, Catapult synthesizes your C++ reference model into an RTL design that matches the model, and CFormalSLEC verifies that they are logically equivalent. Before this is accomplished, however, you may encounter various falsifications. This section describes the various causes of falsifications in CFormalSLEC and how to resolve them.

## Falsifications Due to C++ Reference Model Defects

Certain defects in the C++ reference model, such as uninitialized reads and reading/writing outside array bounds, can lead to falsifications.

Although reference models with these defects may simulate correctly, their synthesized designs may not be functionally equivalent. These falsifications typically appear as violations of properties synthesized to model undefined C++ behavior in the presence of such defects. See Property Checking for C++ Designs in the *SLEC User's Manual*.

To identify defects in your C++ code, Run CdesignChecker before synthesis, as described in Preparing Your Design for Verification. There may also be some defects which are not identified until equivalence checking in CFormalSLEC.

### Uninitialized Variables and Arrays

A specific class of C++ coding defects, related to uninitialized memory, warrants further discussion, as it can result in falsifications that are difficult to debug.

A falsification due to an uninitialized variable typically appears as a non-deterministic X-value in the C++ model versus a concrete or X-value in the RTL, as shown below.

```
[FALSIFIED 3]
Intermediate-pair: spec.imp#0@ls (     1,     0)    'hX @ 253
impl.CALYPTO_DUT.imp_core_inst.for_lpnxt_ls_dtprb (     1,     0)
 'h00000000 @ 353
```

If a falsification occurs on an output map, then the uninitialized variable corrupts the design output. If the falsification occurs on an intermediate probe map, however, it may not necessarily affect the design output. Regardless of whether the falsification appears on an output or intermediate map, you must resolve it to proceed with verification. To resolve the falsification, simply initialize the appropriate variables and/or arrays in the C++ model. Initialize scalars by assigning them a value. Initialize class instances by adding a constructor to the class definition. Initialize arrays either with an unrolled loop, or with the ac::init_array functionality as shown below:

```
ac_int<10, false> A[100];
ac::init_array<AC_VAL_0>(A, 100)
ac_fixed<18,5,true> F[20];

#pragma hls_unroll yes
for(int i=0; i<20; ++i)
 F[i] = 0.0;
```

Alternatively, you can apply the -weaken argument with the *cat2slec* setup script, which instructs CFormalSLEC not to access the variables until they are initialized. While this resolves the falsification, it can negatively impact runtime.

**Note:**
Arrays that map to memories do not require initialization.

To diagnose falsifications related to uninitialized variables, you can configure CFormalSLEC to formally check that each variable is initialized in the C++ reference model. Enable this check by adding the following line to *slec_pre_build.tcl*:

```
config_property_checks -enable -umr
```

Then run verification once again. You can confirm that the Uninitialized Memory Read (UMR) property checks appear in the *results.log* output. See Property Checking for C++ Designs in the *SLEC User's Manual*.

## Falsifications Due to Invalid Input Space

Because CFormalSLEC performs block-level verification, it does not apply contextual information from higher-level scopes. In particular, CFormalSLEC does not apply any constraints from the context in which the block resides. As a result, some falsifications may arise due to an invalid input to the block.

For example, consider two blocks, A and B, where block A produces an index between 0 and 9, and block B consumes the index to access an array of size 10. The index must be a 4-bit value, which allows values 0 to 15. CFormalSLEC may report a falsification on block B related to reading out of the array bounds when the index value is greater than 9. Because this is outside the legal space, this falsification is not valid.

To restrict inputs to legal space, you must specify constraints on the inputs. In this example, you would specify a constraint on the index value to be in the range 0-9. See Manual Constraints.

## Other Falsifications

There are several other types of falsifications that you may encounter.

Falsifications can arise due to incorrect synthesis assumptions when running Catapult. For example, one possible assumption might be that no memory access dependencies exist across iterations of a pipelined loop. If you specify this assumption in Catapult (as described in section "Using ignore_memory_precedence" of the *Catapult® Synthesis User and Reference Manual*), the synthesized design will represent this assumption. During verification, however, CFormalSLEC may discover that indeed there are memory access dependencies across iterations of the pipelined loop. If CFormalSLEC does find these dependencies, it generates the corresponding falsification and waveform traces.

If you encounter falsifications related to incorrect synthesis options, either change these assumptions in Catapult, or provide CFormalSLEC appropriate constraints to restrict the verification space. See Setting Input Constraints in the *SLEC User's Manual*.

If the falsification does not match any of those previously described, it may be due to incorrectly synthesized hardware design. If this is the case, contact your support representative.

# Chapter 3
# Managing Verification Complexity

During the verification process, you may encounter situations where design complexity becomes excessive, leading to very long verification times. This chapter describes several methods of reducing verification complexity.

## Design for Verification

In addition to avoiding design styles described in the limitations earlier, certain design styles are amenable for more efficient formal verification, and improve the likelihood of successful verification. Most of these styles are also considered to be generally good software development practices.

Here are some coding guidelines to that effect:

- Limit the scope of variables.
- Avoid interdependent variables, particularly across loop boundaries.
- Use simple loop bounds, particularly for nested loops. For example, use a loop index that is simply incremented by one on each iteration.
- Refactor large chunks of repeated code.
- Use Algorithmic C Data Types and functions for performing bit manipulations. See https://hlslibs.org/ for more information. Avoid custom implementations of functions.
- Declare arrays intended to be ROMs with const keyword.
- Reduce the “strength” of operators, if possible, such as using right shifts/bit-range selection instead of division/modulus by a power of 2, etc.
- Avoid pointer arithmetic, instead, use array index lookup.
- Do not initialize memories, particularly in unrolled loops.

## CCORE Hierarchies

A Catapult C Optimized Reusable Entity (CCORE) is a user-defined function that Catapult has synthesized and optimized for re-use throughout the design. CCOREs allow CFormalSLEC to partition the formal verification of the design for better divide and conquer. CCOREs can exist within design blocks, and can also contain other CCOREs. Put often-repeated code or complicated arithmetic functionality in separate CCOREs to help SLEC decompose the problem better.

By default, CFormalSLEC verifies the design in a top-down manner in a single verification run. CFormalSLEC also supports bottom-up verification of CCOREs, whereby CCOREs are verified separately from the design block in which they reside.

Bottom-up CCORE verification is typically more scalable than top-down verification, but requires additional verification runs. Top-down and bottom-up CCORE verification are independent from the synthesis of the CCORE; either approach is valid whether the CCORE was synthesized top-down or bottom-up.

**Note:**
If a CCORE contains state-holding elements such as static variables, you must perform bottom-up verification.

## Performing Bottom-Up CCORE Verification

The following procedure describes how to perform bottom-up CCORE verification. Bottom-up verification can mitigate capacity issues encountered during top-down verification.

### Prerequisites

- You have synthesized your design in Catapult with formal verification enabled.
- The design contains CCOREs.
- You wish to perform bottom-up CCORE verification instead of top-down CCORE verification.

### Procedure

1. In the solution for the design block containing CCOREs, run the following command:

```
cat2slec.sh {vlog|vhdl} -bup
```

2. Run the normal SLEC verification flow. This step proves equivalence for the design block with the assumption that the CCORE logic is correct.
3. For each CCORE solution recursively contained in the design block, change to the respective solution directory and repeat steps 1 and 2 above.

### Results

The entire design block is formally proven when you have completed the procedure above for all CCOREs in the design block.

## Pass-Through CCOREs

Another technique to rein in complexity involves targeted synthesis changes. In designs that are “arithmetic heavy,” and where Catapult has optimized the hardware implementation to the extent that it is hard to establish correspondence between the reference model and the implementation, CFormalSLEC provides clues on verification hotspots when it is not able to converge on a proof.

The *hard_problem.log* file lists the arithmetic operators and the line numbers in the specification and implementation that contribute to the verification complexity. The operators are sorted in the order of their relative complexity to proof convergence with normalized hardness factors. A sample *hard_problem.log* file is shown below.

During the course of the verification of a subproblem, the solver encounters complexity. The statistics of the netlist under verification are listed, and given the high number of arithmetic operators, it is fairly evident that the subproblem has significant hardware complexity.

```
Time    13229 s.: Hard problem {
(PEN_PAIR: spec_INST_adapter_std_ave_rsc_dout_0_ln102_1_$123_651245_$0,
impl_INST_adapter_std_ave_rsc_dout_arr_0_26648_$323_651246_$0)
Abstraction: Last_Cut
Problem file: pen-----0.v

CSMT Problem divided into 2 sub-problems
Hardness estimate for the first sub-problem which timed out:

Netlist statistics:
  Operator     Count (Spec Impl Common)    Details
  Add            900 ( 166  416    318)   15 of which are 16-bit
  Sub            116 (  70    0     46)   2 of which are 15-bit
  Rshift        200 (  36  104     60)   200 of which are 10-bit
  Rashift        25 (  12    6      7)   16 of which are 11-bit
  Mux          22347 (1990 13610   6747) 26 of which are 71-bit
  Input         1630 (  78  798    754)   2 of which are 71-bit
  TOTAL        43768 (3920 26616  13230)

Hardness  Operator   Bitwidth   Inputs   Source   Trace
   1.000   Rashift        11       2   spec    btc_utils.h:244
   0.055   Add             13       2   spec  btc_enc_vid_front.cpp:650
   0.051   Add           12       2   spec  btc_enc_vid_front.cpp:172
   0.051   Add             12       2   spec  btc_enc_vid_front.cpp:159
   0.051   Add             12       2   spec  btc_enc_vid_front.cpp:594
   0.051   Add             12       2   spec  btc_enc_vid_front.cpp:625
   0.051   Add             12       2   spec  btc_enc_vid_front.cpp:757
   0.047   Sub             11       2   spec  btc_enc_vid_front.cpp:745
   0.047   Sub             11       2   spec  btc_enc_vid_front.cpp:748
   0.047   Sub             11       2   spec  btc_enc_vid_front.cpp:751
}
```

The CFormalSLEC flow employs several state-of-the art techniques to bridge the abstraction gap between the two models under verification. However, Catapult's optimization steps can sometimes render the task of establishing equivalence challenging. In this case, the Rashift operation on line #244 in the specification module contributes the most to the verification complexity. It is quite possible that this operation may have been optimized away or folded in with another operation in the implementation RTL design. As a result, the solvers are unable to establish intermediate equivalence points thus resulting in verification complexity.

An effective approach to this is to rein in Catapult optimization around this operation, so that a well-defined intermediate mapping point exists between the two models at this point. This can be achieved by reducing the scope of Catapult's optimization region and placing a fence around this operator so that its interface is preserved. This is achieved by defining a pass-through CCORE around the operator of concern. A pass-through CCORE is shown on the left side of Table 5: Example Pass-Through CCORE. It basically defines an interface around the operator(s) directing Catapult to modify the scope of its optimization appropriately. Notice that this usage of the pass-through CCORE does not change the functionality of the design.

**Table 5. Example Pass-Through CCORE**

| passthrough | C++ Specification |
|---|---|
| `#pragma map_to_operator [CCORE]`<br>`#pragma hls_ccore_type`<br>`combinational`<br>`template<typename T>`<br>`T ptcc(const T &a) {`<br>`  return a;`<br>`}` | `template <int N>`<br>`static ac_int<N, true> round_normal(`<br>`  ac_int<N, true> in,`<br>`  ac_int<3, false> cnBitsRnd`<br>` )`<br>` {`<br>`   if (cnBitsRnd == 0)`<br>`      return in;`<br>`   in = passthrough(in >>(cnBitsRnd -`<br>` 1);`<br>`   in = in + 1;`<br>`   ac_int<N,true> result = in >> 1;`<br>`   return result;`<br>`} //end r` |

The passthrough definition in Table 5: Example Pass-Through CCORE is provided as a convenience to you if you include *ac_probe.h* prior to synthesis. You can then use the passthrough construct directly in your C model.

You are now required to resynthesize the design and then run CFormalSLEC on the models. This technique of managing complexity using pass-through CCOREs might require a number of iterations until you find the right combination of pass-through CCOREs that helps alleviate the verification complexity and enables CFormalSLEC to converge.

**Note:**
Pass-through CCOREs tend to impose an overhead on design optimization, but the targeted nature of this technique keeps the overhead to a minimum. For most designs, you should see little to no impact on overall design quality of results.

# Solver Tuning

CFormalSLEC by default comes with appropriate configuration to solve the large majority of verification problems. However, there are occasional instances where tuning various parts of the verification flow can improve CFormalSLEC's performance. Here are some of the high level symptoms, and the outlines of steps the user can take to aid CFormalSLEC. Details of the steps are provided in the *SLEC User's Manual* and *SLEC Reference Manual*.

## Slow State Check

State check optimization is the first formal engine CFormalSLEC invokes after simulation based validation. If this stage is taking more time than desired, you can disable it with:

```
set_global seq_perform_state_checks only_bec
```

## Low effort engine (LL1/LL3) performance

If the engines are proving the maps provided, but are slow due to a large number of maps, you can enable distributed processing. See Distributed Processing in SLEC in the *SLEC User's Manual*.

In some cases, however, many problems may be aborted, as shown below:

```
[SEQ-SWTWP]  Current Solver Progress : 5 output problems proven, 0
 falsified, 21 aborted, 43 remaining : 1943 seconds elapsed.
```

In this case, you can disable low effort engines with the following command:

```
set_global enable_ll3_engine 0
```

Alternatively, you can increase the time limit (default: 30 sec) provided for each map:

```
set_global ll3_problem_time_limit 50
```

## High effort L1 (HL1) engine performance

The HL1 engine operates in two distinct phases. The first phase finds and proves intermediate equivalences. The second stage uses the intermediate equivalences to prove the maps.

In some cases, the proof of intermediate equivalences may take longer than desired:

```
[SEQ-PSF]      26517 seconds:     414 proven,     178 rejected,       0
 aborted,   17392 processed,     251 remaining in 2 equivalence classes.
```

In this case, you can disable intermediate proofs with the following command:

```
set_global enable_pens 0
```

In some cases, however, the engines are proving the maps provided, but are slow due to a large number of maps, as shown below:

```
[SEQ-SWTWP]  Current Solver Progress : 46 output problems proven, 0
 falsified, 0 aborted, 943 remaining : 1546 seconds elapsed.
```

In this case, you can enable distributed processing to reduce verification time.

### High effort L3 (HL3) engine performance

In most cases, the HL3 engine is not required; most verification problems are handled by the aforementioned engines. If the HL3 engine is running and running longer than desired, you can enable distributed processing.

## Case Splits

Using case splits to decompose a verification problem is an effective way to rein in verification complexity. This is particularly significant when the C++ reference model has its fair share of conditional statements.

CFormalSLEC provides a couple of different ways to employ case split strategies. A fully automated approach identifies effective case split candidates in the design based on heuristics and applies them to effectively decompose the verification problem. These case splits can be either structural or temporal in nature, or both. CFormalSLEC guarantees the soundness and the completeness of this approach. The feature is enabled by default in the CFormalSLEC flow. Sometimes, this may be insufficient to manage verification complexity and you may want to exercise greater freedom in specifying case split candidates.

CFormalSLEC provides a way for you to specify this information using the following Tcl command:

```
case_split list_of_signals
```

There is a form of case split called external case splits, which works well with manual constraints. See Manual Constraints.

The CFormalSLEC tool does all the bookkeeping and ensures that the verification is sound. See the reference page for this command in the *Sequential Logic Equivalence Checker (SLEC) Reference Manual* for more information.

### Related Topics

case_split [Sequential Logic Equivalence Checker (SLEC) Reference Manual]

## Manual Constraints

One technique for reducing verification complexity is to reduce the scope of the verification problem. The most common way to reduce the scope is to constrain the input space. While CFormalSLEC provides mechanisms to constrain the input space, it is your responsibility to run verification under different input constraints to achieve the desired coverage.

The following sections describe several ways of setting manual constraints on three common types of inputs: DirectInputs, streaming inputs, and non-streaming inputs.

**Note:**
CFormalSLEC does not track the overall coverage of the cumulative runs. However, if you utilize the external case split feature, CFormalSLEC does, in fact, runs all combinations and tracks the cumulative coverage.

## Constraining DirectInputs

In Catapult, DirectInputs are inputs that typically hold their values indefinitely (for example, configuration parameters). By default, CFormalSLEC holds these inputs to a symbolic constant value throughout verification. You can constrain these inputs to a specific value with the set_constant command. Alternatively, you can constrain these inputs to a specific subset of values with the set_symbolic_constant command.

For example, consider a design with a DirectInputs x_bound, y_bound, and mode. The verification wrappers create the following signals corresponding with the input signals:

**Table 6. Input Signal Mapping**

| Input Signal | Wrapper Signal |
|---|---|
| mode | slecIN_mode_rsc_idx_0 |
| x_bound | slecIN_x_bound_rsc_idx_0 |
| y_bound | slecIN_y_bound_rsc_idx_0 |

**Note:**
The above signal names are listed in the CFormalSLEC wrappers, which are generated by the cat2slec setup script.

You can constrain the 'mode' value to 1 in both the specification and implementation designs by adding the following lines to *slec_constraints.tcl*.

```
set_constant -spec -module spec_wrapper -value 1 slecIN_mode_rsc_idx_0
set_constant -impl -module impl_wrapper -value 1 slecIN_mode_rsc_idx_0
```

You can constrain the x_bound and y_bound inputs to the ranges of {1-79} and {1-1023}, respectively, by adding the following lines to *slec_constraints.tcl*.

```
set_symbolic_constant -inputs {spec.slecIN_x_bound_rsc_idx_0
impl.slecIN_x_bound_rsc_idx_0} -values {1 - 79}

set_symbolic_constant -inputs {spec.slecIN_y_bound_rsc_idx_0
impl.slecIN_y_bound_rsc_idx_0} -values {1 - 1023}
```

See “Handling DirectInputs and non-blocking IO” in the *Catapult® Synthesis User and Reference Manual*.

## Constraining Non-Streaming Inputs

Non-streaming inputs hold their values throughout the execution of the top-level function call.

You can constrain non-streaming inputs with the create_constraint command as follows:

```
# Create a waveform to describe constraint
create_waveform -name X_BOUND -bitwidth 10 "(1-79)+"
# Set spec side input port to this constraint
create_constraint -waveform X_BOUND  {spec.slecIN_x_bound_rsc_idx_0}
# Map the input ports, so the same constraint is applied to the RTL input port
create_map -input spec.slecIN_x_bound_rsc_idx_0 impl.slecIN_x_bound_rsc_idx_0

# Create a waveform to describe constraint
create_waveform -name Y_BOUND -bitwidth 10 "(1-1023)+"
# Set spec side input port to this constraint
create_constraint -waveform Y_BOUND {spec.slecIN_x_bound_rsc_idx_0}
# Map the input ports, so the same constraint is applied to the RTL input port
create_map -input spec.slecIN_y_bound_rsc_idx_0 impl.slecIN_y_bound_rsc_idx_0
```

**Note:**
You must specify the input names contained in the CFormalSLEC wrappers, which are generated by the cat2slec setup script.

## Constraining Streaming Inputs

Streaming inputs provide a new value on each read from the interface. The index value at the end of the signal name increments for each read.

For example, if the signal corresponding to the first read is slecIN_x_bound_rsc_idx_0, the following signals would represent subsequent reads:

```
slecIN_x_bound_rsc_idx_1
slecIN_x_bound_rsc_idx_2
slecIN_x_bound_rsc_idx_3
... (and so on)
```

To properly constrain the streaming input, you must repeat the procedure described in Constraining Non-Streaming Inputs for each of the indexed signals. Obtain the signal names listed from the input mapping lines in the *slec_<vlog|vhdl>.tcl* script, which is generated by *cat2slec*.

## Constraints Using Property Assumptions

In some cases, you may need to create complex constraints involving multiple inputs or changing values across time, or any combination thereof. There are two ways to set such constraints: inserting assumptions in the C++ source code, and inserting System Verilog Assertions (SVAs).

### Assumptions in the C++ Source Code

You can add an assertion describing the constraint in the C++ source code, with a pragma instructing CFormalSLEC to consider the assertion as a constraint. Here is an example to constrain a composite index made out of two inputs.

```
idx = row*10+col;
#ifdef CATAPULT_SLEC
#pragma assume yes
assert(0 <= idx && idx < 100);
#endif
sum += Array[idx];
```

### Assumptions Using System Verilog Assertions

You can specify assumptions using System Verilog Assertions (SVAs), a standard language to specify spatial and temporal assertions for formal and simulation-based verification tools. Place the SVAs within transactors, which can be written in either Verilog or SystemVerilog, and connect them to the appropriate design. An example SVA constraint is shown below.

```
assume property @(posedge clk ready -> ##2 valid);
```

The example above sets a constraint to check that the valid signal is asserted 2 cycles after ready is asserted.

Note that in addition to transactors, CFormalSLEC also supports connecting SVA auxiliary logic to Verilog RTL using the SystemVerilog bind statement. For more details on transactors and properties, see Using Transactors in *SLEC User's Manual*.

### Understanding the Effects of Property Assumptions

When implementing property assumptions, you must be aware of the impact on the scope of the verification problem. For example, a C++ assert statement restricts the input space of the context in which it resides and may have a cascading effect on other portions of the design. If conflicting assertions exist, the input space may contain no valid inputs, leading to a meaningless, vacuous proof.

The example below demonstrates a case where the user inadvertently placed the >= operator instead of the <=, leading to an empty input space.

```
for (int i=0; i<10; ++i)
  for (int j=0; j<10; ++j) {/
    #pragma assume yes
    assert(i+j>=100);
    ...
  }
```

Even if the constraints do not directly conflict, they can still be too limiting, resulting in overly constrained exploration. To ensure sufficient coverage, enable formal coverage analysis as described in Determining Formal Coverage.

## External Case Splits

CFormalSLEC also provides the case split functionality for dividing a single verification problem into smaller problems, each with a distinct set of input values. With external case splits, CFormalSLEC launches separate subsessions to solve each of the smaller problems, often leading to reduced total runtime.

For example, assume you have two inputs: mode and filter_size. The verification wrappers create the following signals corresponding with the input signals:

**Table 7. Input Signal Mapping**

| Input Signal | Wrapper Signal |
|---|---|
| mode | slecIN_enc_mode_rsc_idx_0 |
| filter_size | slecIN_filter_size_rsc_idx_0 |

**Note:**
The above signal names are listed in the CFormalSLEC wrappers, which are generated by the cat2slec setup script.

You can implement an external case split to test distinct scenarios on the mode and filter_size inputs by adding the following lines to *slec_constraints.tcl*.

```
case_split -external spec.slecIN_enc_mode_rsc_idx_0 0 1 2; # 4th case is
 all other values
create_waveform -name WF_SMALL -bitwidth 10 (0-100)
create_waveform -name WF_MEDIUM -bitwidth 10 (101-300)
case_split -external spec.slecIN_filter_size_rsc_idx_0 WF_SMALL WF_MEDIUM;
 # 3rd case is all other values
```

A snippet of the external case split result is shown below:

```
------------------------------------------------------------------------
    Number of workers  : 4
       6 Passed jobs    : ecs_0_0 ecs_0_1 ecs_0_2 ecs_0_3 ecs_1_0
                          ecs_1_1 ecs_1_2 ecs_1_3 ecs_2_0 ecs_2_1
                          ecs_2_2 ecs_2_3
       0 Failed jobs    :
```

```
       0 Falsified jobs :
       0 Running jobs   :
-------------------------------------------------------------------------
     [SLEC-GIM]   Status of external case split combinations: 12 jobs
 passed, 0 jobs falsified, 0 jobs failed.
     [ORC-VEC]    Verified all 12 external case split combinations on 2
 inputs.
```

The jobs are named in the form ecs_<*filter_case*>_<*mode_case*>, where <*filter_case*> and <*mode_case*> correspond with distinct scenarios on the filter_size and mode inputs, respectively. As you can see, the encoding mode contains four different classes: one for each of the specified values, and a fourth for all other values. Similarly, the filter_size contains three different classes: one for each of the specified values, and a third for all other values. This yields a total of 12 combinations, each of which runs in a separate CFormalSLEC subprocess.

For further details, see Reducing Runtime with Case Split in the *SLEC User's Manual* and case_split in the *SLEC Reference Manual*.

## Path-By-Path Mode

If the C++ specification model has a collection of unrolled loops coupled with arithmetic-heavy computations, it is usually a prime candidate for CFormalSLEC's powerful path-by-path technique. As the name suggests, this approach decomposes the designs under verification into a collection of paths.

Let us review the discussion on transactions from Chapter Introduction, "Introduction" on page 7, to get an intuitive understanding of this decomposition approach. CFormalSLEC establishes a notion of transaction boundaries, which basically follows the program control flow at the basic block level. In the code snippet below, there are three possible transaction boundaries (paths):

- A -> B
- B -> B (signifying a loop)
- B -> C

Code snippet from file *accum.cpp*:

```
int accum(int *A, size_t N) {
A: static int ctr = 0, sum = 0;
Loop: for (auto i = 0; i < N; ++i) {
   B: sum += A[i];
}
C: return sum + ctr++;
}
```

As the number of loops in the design increases, coupled with the possibility of nesting, the number of possible program paths increases dramatically. CFormalSLEC in path-by-path mode keeps track of all the paths, verifying them separately, and doing all the bookkeeping to ensure soundness and completeness of the verification exercise. You can enable path-by-path decomposition mode simply by invoking CFormalSLEC on a different top-level target Tcl file, as follows:

```
% $SLEC_HOME/bin/slec run_paths.tcl
```

The CFormalSLEC tool runs the selection of vlog or vhdl chosen when running the cat2slec command.

There is another side benefit to running CFormalSLEC in the "path-by-path" mode. Because this verification technique is formulated to render the verification of each path largely as a stand-alone and independent exercise, it is amenable to path-level parallelization. See "Distributed Processing in SLEC" in the *SLEC User's Manual*.

In the sample log snippet below, the path-by-path flow has decomposed the verification run into two paths in addition to the init_check and the property_check tasks. The size of each of the verification problems is smaller than the monolithic run, and this leads to improved CFormalSLEC performance.

**Note:**
CFormalSLEC in path-by-path mode, depends critically on the verification setup being inductive for the design under verification. If the setup is incomplete and misses specifying all the invariants needed to make the verification problem inductive, this decomposition strategy will not work, and you will observe inductive falsifications in the course of verifying the individual paths. Contact the SLEC technical support team to help resolve this issue.

```
Passed  jobs   :
Failed  jobs   :
Running jobs   : init_check path_1_2 path_2_2 property_check
-----------------------------------------------------------
Finished job property_check with status: 0 @2018-06-14 13:58:27
==================================================================
****** Equivalence Results (property_check/results.log) **********
==================================================================
                  Proven Cond-Proven Bounded-Proven Falsified Unresolved
Output-Maps         0           2         0            0          0
Active Maps          0           0         0            0          0
Properties          0         156         0            0          0
-----------------------------------------------------------
2018-06-14 13:59:50
Passed jobs   : property_check
Failed jobs   :
Running jobs  : init_check path_1_2 path_2_2
-----------------------------------------------------------
Finished job init_check with status: 0 @2018-06-14 14:00:07
-----------------------------------------------------------
2018-06-14 14:00:20
Passed jobs    : property_check init_check
Failed jobs    :
Running jobs   : path_1_2 path_2_2
----------------------------------------------------------
Passed jobs   : property_check init_check path_1_2 path_2_2
Failed jobs   :
Running jobs  :
-----------------------------------------------------------
Finished job path_1_2 with status: 256 @2018-06-14 14:05:09
-----------------------------------------------------------
-----------------------------------------------------------
Finished job path_2_2 with status: 256 @2018-06-14 14:05:08
-----------------------------------------------------------
```

## Bounded Proof Mode

To mitigate verification complexity, you can a perform bounded proof rather than a full proof. Bounded proofs limit the number of transactions verified. Bounded formal verification is an established verification practice, and can significantly reduce verification time while still providing satisfactory coverage.

Launch a bounded proof by specifying the proof_depth in the verify command. See Bug-Finding Mode in the *SLEC User's Manual*.

For the *accum.cpp* example shown in Path-By-Path Mode, setting a bound of 10 would limit the verification to the following:

- One execution of step A
- 9 iterations of the accumulator loop B

Since the function specifies a variable-sized loop, implementing a bounded proof can significantly reduce verification time.

## Determining Formal Coverage

When placing limits on full formal exploration in terms of constraints or bounded proof, you may want to know which parts of the design have been omitted. CFormalSLEC inserts coverage monitors for each block or branch condition, and reports whether formal analysis has explored that branch.

Enable formal coverage by setting enable_coverage_maps to 1 in *slec_pre_build.tcl*, as described in Verification Setup Files and Control Variables.

Below is a sample summary of formal coverage analysis:

```
=========================================================================
*************** Coverage Results (calypto/results.log) ***************
=========================================================================
       Proven-uncoverable  Covered  Not-coverable-upto-bound  Unresolved
                        5       43                         2           0
```

You can find a detailed report for each coverage target in the coverage_report.log file in the CFormalSLEC work directory.

For further details on the formal coverage flow in CFormalSLEC, consult the Application Note *Formal Coverage with SLEC*.

# Chapter 4
# Troubleshooting

CFormalSLEC may encounter occasional compatibility issues with the Catapult tool. When this happens, error codes are issued to help troubleshoot.

## Catapult Flow Compatibility Errors

You may encounter various error codes when running CFormalSLEC on Catapult Designs. The error codes primarily serve to highlight inconsistencies in the design flow that could prove fatal to the CFormalSLEC flow.

The following table lists the most common errors that you might come across while running CFormalSLEC.

**Table 8. Common Catapult Flow Compatibility Errors**

| Error Code | Cause of the Error |
|---|---|
| CPT-CMLC | There are conflicting mark_loop commands. CFormalSLEC does not allow a loop at a given location (given as a "file/line pair") to be unrolled through one path and flattened through another path. This limitation can also be exposed if Catapult marks one copy of that loop as a dead loop, because CFormalSLEC will have to unroll the dead loop and flatten the other. |
| CPT-HFAS | CFormalSLEC does not support top-down verification of CCOREs containing static variables. See Resolving CPT-HFAS Error. |
| CPT-LIPUL | Indicates error with partial loop unrolling. See Resolving CPT-LIPUL Error. |
| CPT-MICR | The CFormalSLEC build_design command needs to unroll a loop but cannot do so. If the unroll_limit has not been specified, and the number of unrolled iterations is greater than the default value of 1024, the build_design command will error out. The workaround is to increase the default number of unrolled iterations using the global variable maximum_iter_count.<br><br>Specifying the unroll_limit is usually for cases where the communication between Catapult and CFormalSLEC has broken down and the loops are expected to be unrolled. If there is indeed a discrepancy between loops, there will be mismatches. |
| CPT-MMLC | Indicates a setup issue. Catapult is expected to pass appropriate mark_loop commands to CFormalSLEC for each relevant loop. If a loop does not have a mark_loop command, this error is issued. If the loop is not dead code, you can instruct CFormalSLEC to continue by providing the command "disable_msg CPT-MMLC" before the build_design command. This should also work if the loop is dead code, as loops are typically dead due to observability, and by leaving the loop in (in unrolled form), CFormalSLEC's model will just have additional computation that will not have any bearing on the outputs. There may be some intermediate probes that could differ, resulting in falsifications. |

**Table 8. Common Catapult Flow Compatibility Errors (continued)**

| Error Code | Cause of the Error |
|---|---|
| | This error is also raised when a design is specified with a struct at its interface and the constructor of the struct has a loop. This can be handled as above, or else you can manually remove the loop from the C code before synthesis. |
| CPT-PRISL | A pointer is declared and defined outside the sequential loop and redefined in the body of the loop. Consider the following code:<br>`int temp1;`<br>`int* ptr = &amp;temp1;`<br>`int temp2;`<br>`while (1) {`<br>`   out.write(*ptr);`<br>`   ptr = &amp;temp2;`<br>`   wait();`<br>`}`<br>In above code, for the first iteration of while loop, ptr points to temp1 and then it points to temp2 for the rest of the iterations. This is not supported. |
| CTS-AOCNS | Indicates the presence of arrays through ac_channel. This is not supported. |
| CTS-MIA | There is a missing path to a memory instance in the Catapult DB. See Resolving CTS-MIA Error. |
| CTS-NAMAS | Indicates an unsupported ac_channel protocol. Supported protocols are listed in Interface Components. |
| CTS-NULP | Catapult has introduced an extra loop when initializing and mapping a local array. See Resolving CTS-NULP Error. |
| CTS-PULNS | Indicates error with partial loop unrolling. See Resolving CPT-LIPUL Error. |
| CTS-RTFR | The Catapult directive TRANSACTION_DONE_SIGNAL is not set to true. See Resolving CTS-RTFR Error. |
| CTS-SMNM | Indicates that the setup phase is not able to map arrays to symbolic memories. Automatic commands for memory maps may not be generated due to the presence of the following: scalars mapped to memory, local arrays, ROMs, unsupported memory layouts. |
| CTS-UBR | Indicates an error when flattening a loop where the first statement is a break statement. See Resolving CTS-UBR Error. |

## Error Resolution Procedures

This section describes how to resolve certain errors.

## Resolving CPT-HFAS Error

This procedure describes how to resolve the CPT-HFAS error.

### Symptoms

You have run CFormalSLEC and observed the CPT-HFAS error.

### Causes

CFormalSLEC does not support top-down verification of CCOREs containing static variables.

### Solution

Perform bottom-up verification of CCOREs as described in Performing Bottom-Up CCORE Verification.

## Resolving CPT-LIPUL Error

This procedure describes how to resolve the CPT-LIPUL error.

### Symptoms

You have run cat2slec and observed the CPT-LIPUL error.

### Causes

Indicates partial loop rolling, which is not supported.

### Solution

Manually perform partial unrolling or completely unroll the loop.

**Note:**
Completely unrolling the loop can affect synthesis quality of results, including area, timing, and latency.

# Resolving CTS-MIA Error

This procedure describes how to resolve the CTS-MIA error.

## Symptoms

You have run cat2slec and observed the CTS-MIA error.

## Causes

There is a missing path to a custom memory instance in the Catapult database.

## Solution

Follow the steps below to enter the missing memory information in Catapult Library Builder.

1. Go to the directory containing the *.lib under investigation.
2. Invoke the Catapult Library Builder with the following line:

   ```
   % $MGC_HOME/bin/catapult -p lb
   ```

3. Open the appropriate library file.
4. In the lefthand pane, click on **Mods > Vars**.
5. Click the **New** button at the bottom to create a new variable.
6. In the **Variable** field, enter *VerilogArrayPath* (for Verilog verification) or *VHDLArrayPath* (for VHDL verification).
7. For the **Value** field, enter the value as follows:
   a. Click on **Mods > Netlist**.
   b. Open the Verilog or VHDL file, depending on the RTL verification language.
   c. Identify the memory variable in the file.
   d. Set the **Value** field to the name of the memory variable identified in step (c).
8. Save the library.
9. Restart Catapult and re-run synthesis on the design.

**Note:**
For full documentation on Catapult Library Builder, see the Generating Libraries for Catapult at https://support.sw.siemens.com/en-US/.

## Resolving CTS-NULP Error

This procedure describes how to resolve the CTS-NULP error.

### Symptoms

You have run cat2slec and observed the CTS-NULP error.

### Causes

When Catapult initializes a local array in the design and maps it to a memory location, it also adds an extra loop. This extra loop is flagged as an error. The error message displays the name of the array along with its source information.

### Solution

Declare the array as static to ensure Catapult initializes the static arrays in the reset behavior.

## Resolving CTS-RTFR Error

This procedure describes how to resolve the CTS-RTFR error.

### Symptoms

You have run cat2slec and observed the CTS-RTFR error.

### Causes

The Catapult directive TRANSACTION_DONE_SIGNAL is not set to true. This setting is required for correct verification wrapper generation.

### Solution

Set the Catapult directive TRANSACTION_DONE_SIGNAL to true.

## Resolving CTS-UBR Error

This procedure describes how to resolve the CTS-UBR error.

### Symptoms

You have run cat2slec and observed the CTS-UBR error.

## Causes

Indicates a problem with loop flattening. If the first statement in a loop is a break statement, and if Catapult automatically moves this break to the end of the loop (due to loop rephrasing), this error is reported.

## Solution

Move the break to the last statement in the loop. An example is shown below.

Original:

```
void test( int a, int &out)
{
   bool last = 0;
   int tmp = 0;
   for(int ind=0; ind< 5; ind++)
   {
      if (last) break;   // Make this the last stmt of this loop
      tmp += a;
      last = tmp;
   }
   out = tmp;
}
```

Corrected:

```
void test( int a, int &out)
{
   bool last = 0;
   int tmp = 0;
   for(int ind=0; ind< 5; ind++)
   {
      tmp += a;
      last = tmp;
  if (last) break; // Correct
   }
   out = tmp;
}
```

# Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *slec/ legal* directory.