Contents

1 Quick Start

2 Reference: Graph-Building API
   2.1 Class Graph .................................................. 7
       2.1.1 ADK-related ............................................ 7
       2.1.2 Adding Steps ........................................... 7
       2.1.3 Connecting Steps Together .......................... 8
       2.1.4 Parameter System ...................................... 8
       2.1.5 Advanced Graph-Building ........................... 8
   2.2 Class Step .................................................. 9
   2.3 Class Edge .................................................. 10

3 User Guide
   3.1 User Guide .................................................. 11
   3.2 Connecting Steps Together ............................... 11
       3.2.1 Automatic Connection by Name .................... 11
       3.2.2 Explicit Connections ................................. 13
   3.3 Instantiating a Step Multiple Times .................... 14
   3.4 Sweeping Large Design Spaces ........................... 15
       3.4.1 More Details .......................................... 17
   3.5 ADK Paths ................................................... 19
   3.6 Assertions ................................................... 20
       3.6.1 The File Class and Tool Class .................... 21
       3.6.2 Adding Assertions When Constructing Your Graph  ........................................ 21
       3.6.3 Escaping Special Characters ....................... 21
       3.6.4 Multiline Assertions ................................ 21
       3.6.5 Defining Python Helper Functions ................ 22
       3.6.6 Using Custom pytest Files ......................... 23
       3.6.7 Assertion Scripts in mflowgen ..................... 23
   3.7 Stashing Pre-Built Steps for Sharing ................... 24
   3.8 Mock Graphs for Modular Step Development ............ 28
   3.9 Static Checks ................................................ 31
       3.9.1 Intent-Implementation Split ........................ 31

4 Common Library Reference
   4.1 Greatest Common Divisor Pipe Cleaner .................. 35
   4.2 FreePDK45 and the Nangate Open Cell Library .......... 37
mflowgen is a modular flow specification and build-system generator for ASIC and FPGA design-space exploration built around sandboxed and modular steps.

mflowgen allows you to programmatically define and parameterize a graph of steps (i.e., sandboxes that run anything you like) with well-defined inputs and outputs. Build system files (e.g., make, ninja) are then generated which shuttle files between steps before running them.

Key features and design philosophies:

- **Process and technology independence** – Process technology libraries and variables can be abstracted and separated from physical design scripts. Specifically, a single node called the ASIC design kit (ADK) captures this material in one place for better maintainability and access control.

- **Sandboxed and modular steps** – Traditional ASIC flows are composed of many steps executing with fixed path dependencies. The resulting flows have low reusability across designs and technology nodes and can be confusing and monolithic. In contrast, _modularity_ encourages reuse of the same scripts across many projects, while _sandboxing_ makes each step self-contained and also makes the role of each step easy to understand (i.e., take these inputs and generate those outputs).

- **Programmatically defined build-system generator**: A Python-based scripting interface and a simple graph API allows flexible connection and disconnection of edges, insertion and removal of steps, and parameter space expansions. A simple graph can be specified for a quick synthesis and place-and-route spin, or a more complex graph can be built for a more aggressive chip tapeout (reusing many of the same steps from before).

- **Runtime assertions** – Assertions can be built into each modular node and checked at runtime. Preconditions and postconditions are simply Python snippets that run before and after a node to catch unexpected situations.
that arise at build time. Assertions are collected and run with pytest. The mflowgen graph-building DSL can also extend a node with *design-specific* assertions by extending Python lists.

- **A focus on hardware design-space exploration** – Parameter expansion can be applied to steps to quickly spin out parallel builds for design-space exploration at both smaller scales with a single parameter (e.g., sweeping clock targets) as well as at larger scales with multiple parameters (e.g., to characterize the area-energy tradeoff space of a new architectural widget with different knobs). Dependent files are shuttled to each sandbox as needed.

- **Complete freedom in defining what steps do** – Aside from exposing precisely what the inputs and outputs are, no other restrictions are placed on what steps do and a step can be as simple as hello world (one line). A step may conduct an analysis pass and report a gate count. A step can also apply a transform pass to a netlist before passing it to other tools. In addition, a step can even instantiate a subgraph to implement a hierarchical flow.

mflowgen ships with a limited set of ASIC flow scripts for both open-source and commercial tools including synthesis (e.g., Synopsys DC, yosys), place and route (e.g., Cadence Innovus Foundation Flow, RePLAce, graywolf, qrouter), and signoff (e.g., Synopsys PTPX, Mentor Calibre). In addition, we include an open-source 45nm ASIC design kit (ADK) assembled from FreePDK45 version 1.4 and the NanGate Open Cell Library.
This repo includes a small Verilog design that computes a greatest common divisor function. The datapath includes two state registers and the required muxing and arithmetic units to iteratively implement Euclid’s algorithm. You can use this design to demo the default ASIC flow that ships with mflowgen, which should work for most designs.

**Greatest Common Divisor Circuit (GcdUnit)**

This section steps through how to clone the repo and push this design through synthesis, place, and route with the included open-source 45nm ASIC design kit using either the open-source tools (e.g., Yosys) or the commercial tools (i.e., Synopsys, Cadence, Mentor).

You may want to work in a virtual environment:

```
% python3 -m venv venv
% source venv/bin/activate
```

First, clone the repo:

```
% git clone https://github.com/mflowgen/mflowgen
% cd mflowgen
% TOP=${PWD}
```

Install mflowgen with pip as an editable repo:

```
% pip install -e .
```

The greatest common divisor design has three demo graphs in $TOP/designs/GcdUnit:

1. `construct-open.py` - Open-source toolflow based on Yosys, graywolf, qrouter, and RePlAcE
2. `construct-commercial.py` - Commercial toolflow based on Synopsys, Cadence, and Mentor tools
3. `construct-commercial-full.py` - Commercial toolflow with more steps expanded for greater observability
All three flows use the 45nm ASIC design kit based on FreePDK45 and the NanGate Open Cell Library.

**Note:** To switch between the different graphs, open `$TOP/designs/GcdUnit/.mflowgen.yml` and specify one of the three choices. The remainder of this quickstart will assume you have modified this file and chosen the open-source toolflow.

Refer to *Greatest Common Divisor Pipe Cleaner* for a similar quick start that uses commercial tools (e.g., Synopsys, Cadence, Mentor) instead of open-source tools.

```bash
% cd $TOP
% mkdir build && cd build
% mflowgen run --design ../designs/GcdUnit
```

You can show information about the currently configured flow:

```bash
% make info  # -- shows which design is being targeted
% make list   # -- shows most things you can do
% make status # -- prints the build status of each step
% make graph  # -- dumps a graphviz PDF of the configured flow
```

Now run synthesis and check the outputs of the sandbox to inspect the area report. **Note:** For the commercial flow, check `make list` for the build target name.

```bash
% make open-yosys-synthesis
% cat *-open-yosys-synthesis/outputs/synth.stats.txt
```

You can also run steps using the number from `make list`:

```bash
% make list   # -- 3 : open-yosys-synthesis
% make 3
```

The yosys area report will look something like this:

```
=== GcdUnit ===
Number of wires: 406
Number of wire bits: 1011
Number of public wires: 406
Number of public wire bits: 1011
Number of memories: 0
Number of memory bits: 0
Number of processes: 0
Number of cells: 941
AOI211_X1 3
AOI21_X1 34
AOI22_X1 30
BUF_X1 626
CLKBUF_X1 5
DFF_X1 34
INV_X1 48
NAND2_X1 42
NAND3_X1 3
NOR2_X1 34
NOR3_X1 3
NOR4_X1 4
OAI211_X1 1
```

(continues on next page)
Chip area for this module: 932.330000

Report runtimes to check how long each step has taken:

```bash
% make runtimes
```

Then run place-and-route (requires graywolf and qrouter):

```bash
% make open-graywolf-place
% make open-qrouter-route
```
mflowgen provides a Python library for building graphs in the `construct.py` for your design. In general, you will only use methods from the `Graph` and `Step` classes. The `Edge` class is used internally when you call the various methods that make connections between steps in your graph.

## 2.1 Class Graph

A Graph is composed of nodes and edges (i.e., `Step` and `Edge` objects).

```python
class mflowgen.components.Graph
    Graph of nodes and edges (i.e., Step and Edge).
```

### 2.1.1 ADK-related

The following methods help interface with the ADK.

- **Graph.set_adk(adk)**
  - Sets the ASIC design kit.
  - Searches the paths in “Graph.sys_path” for ADKs (analagous to python sys.path).
  - **Parameters**
    - **adk** – A string representing the subdirectory name of the ADK

- **Graph.get_adk_step()**
  - Gets the Step object representing the ASIC design kit.
  - **Returns**
    - The Step object that was constructed from the currently set ADK.

### 2.1.2 Adding Steps

- **Graph.add_step(step)**
  - Adds a Step to the graph as a node.
The name of the new Step cannot conflict with any steps that already exist in the graph. This method fails an assertion if given a duplicate step name.

**Parameters**
- **step** – A Step object

Graph.get_step(step_name)

Gets the Step object with the given name.

**Parameters**
- **step_name** – A string representing the name of the step from Step.
- **get_name()**

Graph.all_steps()

### 2.1.3 Connecting Steps Together

The `Graph.connect_by_name()` method is preferred when possible to keep code clean. This requires setting up nodes such that the inputs and outputs are name-matched (e.g., stepA has output `foo` and stepB has input `foo`).

Graph.connect(l_handle, r_handle)

Graph.connect_by_name(src, dst)

### 2.1.4 Parameter System

Graph.update_params(params)

Updates parameters for all steps in the graph.

Calls `Step.update_params()` for each step in the graph with the given parameter dictionary.

**Parameters**
- **params** – A dict of parameter names (strings) and values

### 2.1.5 Advanced Graph-Building

Graph.param_space(step, param_name, param_space)

Spins out new copies of the step across the parameter space.

For example, for a graph like this:

```
+-----+ +-----------+ +-----------+
| foo | -> | bar | -> | baz |
|     | ( p = 1 ) |     |     |
+-----+ +-----------+ +-----------+
```

this call:

```
g = Graph()
(....)
g.param_space( 'bar', 'p', [ 1, 2, 3 ] )
```

will be transformed into a graph like this:

```
+-----+ +-----------+ +-----------+ +-----+ +-----------+ +-----------+
| foo | -> | bar-p-1 | -> | baz-p-1 |
|     | ( p = 1 ) |     |     |     |     |
+-----+ +-----------+ +-----------+ +-----+ +-----------+ +-----------+
```

(continues on next page)
Parameters

- **step** – A string for the step name targeted for expansion
- **param_name** – A string for the parameter name
- **param_space** – A list of parameter values to expand to

**Returns** A list of (parameterized) steps (i.e., ‘bar-p-1’, ‘bar-p-2’, and ‘bar-p-3’).

## 2.2 Class Step

A Graph is composed of nodes and edges (i.e., *Step* and *Edge* objects).

```python
class Step(step_path, default=False)
classmethod Step.clone()
classmethod Step.get_input_handle(f)
classmethod Step.get_output_handle(f)
classmethod Step.i(name)
classmethod Step.o(name)
classmethod Step.all_input_handles()
classmethod Step.all_output_handles()
classmethod Step.extend_inputs(new_list)
classmethod Step.extend_outputs(new_list)
classmethod Step.pre_extend_commands(new_list)
classmethod Step.extend_preconditions(new_list)
classmethod Step.extend_postconditions(new_list)
classmethod Step.set_name(name)
classmethod Step.get_name()
classmethod Step.set_param(param, value)
classmethod Step.get_param(param)
classmethod Step.update_params(params, allow_new=False)
classmethod Step.params()
classmethod Step.expand_params()
classmethod Step.escape_dollars()
```
2.3 Class Edge

A Graph is composed of nodes and edges (i.e., Step and Edge objects).

class Edge (src, dst)
classmethod Edge.get_src()
classmethod Edge.get_dst()
3.1 User Guide

3.2 Connecting Steps Together

There are two ways to connect two steps in a graph (i.e., to draw edges between nodes):

1. With `Graph.connect_by_name()`
2. With `Graph.connect().Step.o()`, and `Step.i()`

`Graph.connect_by_name()` tries to automatically connect outputs of one step to inputs of the other step if the files have the same name. Multiple edges can be drawn with a single call.

`Graph.connect()` explicitly connects a single output of a step (specified by `Step.o()`) to a single input of another step (specified by `Step.i()`). Only a single edge is drawn.

Graph building is generally cleaner when connecting by name. The more explicit connect API is useful when names do not match and it is inconvenient to adjust step configurations to make them match.

3.2.1 Automatic Connection by Name

Here is a simple graph with two nodes, one for the design RTL and the other which runs synthesis. We would like to connect the design RTL to the synthesis node.
The RTL node has an output "design.v" and the synthesis node takes an input "design.v". Since these names match, we can use `Graph.connect_by_name()` to simply connect these nodes with an edge like this:

```python
g.connect_by_name( rtl, dc )
```

We get this graph with automatic connection:
3.2.2 Explicit Connections

Here is another simple graph with the same two nodes for design RTL and for synthesis. However, the names no longer match. We would still like to make this connection.

We want to connect the RTL node’s output “GcdUnit.v” to the synthesis node’s input “design.v”. Since these names do not match, connecting by name will not automatically draw any edges.

We can connect explicitly using `Graph.connect()`, `Step.o()`, and `Step.i()`:
We can read this clearly: Connect **rtl output** “GcdUnit.v” to **dc input** “design.v”.

*Step.o()* and *Step.i()* are helper methods necessary for differentiating outputs from inputs within a step. For example, the synthesis node has an input file “design.v” (i.e., the RTL design) and also an output file “design.v” (i.e., the synthesized netlist) with the same name.

We get this graph with our explicit connection:

![Diagram showing the connection between “rtl” and “design.v”]

### 3.3 Instantiating a Step Multiple Times

You can clone a Step in your `construct.py` and instantiate it multiple times in your graph.

For example, say you wanted a graph that looks like this, with the same step (i.e., GDS merge) instantiated twice and two nodes feeding different inputs to each (i.e., foo-gds and bar-gds):
You can use `Step.clone` to build this graph:

```python
# This is the default step
gdsmerge = Step( 'mentor-calibre-gdsmerge', default=True )

# Clone the step however many times you need

gdsmerge_for_foo = gdsmerge.clone()
gdsmerge_for_bar = gdsmerge.clone()

# Give them both new names

gdsmerge_for_foo.set_name( 'gdsmerge-for-foo' )
gdsmerge_for_bar.set_name( 'gdsmerge-for-bar' )

# Add both steps to the graph

g.add_step( gdsmerge_for_foo )
g.add_step( gdsmerge_for_bar )

# Connect up both steps

g.connect_by_name( foo_gds, gdsmerge_for_foo )
g.connect_by_name( bar_gds, gdsmerge_for_bar )
```

### 3.4 Sweeping Large Design Spaces

Being embedded in a high-level language like Python allows mflowgen to generate highly parameterized graphs to explore large design spaces in a single graph. For example, we could sweep the design clock period to quickly see where timing fails, or we could sweep the cross product of a design space including combinational operators (e.g., add, mul, floating ops), different bitwidths, different precision formats, and different input datasets both for testing...
and for energy and timing estimation.

As a simple example, suppose we would like to sweep the `clock_period` parameter in the `open-yosys-synthesis` step in this graph:

![Diagram showing the graph structure with nodes and edges labeled appropriately.]

The mflowgen Python API `Graph.param_space` expands the node for each parameter value in the list:

```python
g = Graph()
(... add steps and connect them together ...)
g.param_space( 'open-yosys-synthesis', 'clock_period', [ 0.5, 1.0, 1.5 ] )
```

The expansion propagates to all downstream nodes, resulting in three slightly different builds:
This essentially “unrolls the loop” within a single graph, allowing you to more easily define and explore a parameter space, while avoiding manually creating these spaces on your own. The three builds can be run in parallel as usual (e.g., with “make -j” on different nodes), and all file management is handled cleanly by the build system according to your expanded graph.

**Note:** If your graph updates parameters, please make sure to only run `Graph.param_space` after those calls.

Specifically, all parameters (other than the target parameter) should already be defined. This is because `Graph.param_space` unrolls the loop across the parameter space for the *target* parameter (e.g., “clock_period”) but uses the existing values for non-target parameters. It is difficult to adjust parameters after the loop has already been unrolled.

Note that parameters are passed as environment variables. This means that parameter sweeping can be flexibly applied anywhere in the physical design flow in a very simple manner:

1. Replace some code with a variable anywhere in your scripts
2. Identify this variable as a parameter (i.e., in the step’s configure.yml)
3. Use the `param_space()` mflowgen API to perform a sweep of that variable

This support is useful for automating design-space exploration sweeps involving one parameter or multiple parameters.

### 3.4.1 More Details

Here is a code example that sweeps the GcdUnit demo circuit across different clock targets (0.5ns, 0.75ns, 1.0ns, 1.5ns) with the open-source toolchain (yosys) and the 45nm standard cell library. Synthesis will fail timing for the 0.5ns clock target.

Set up the mflowgen demo circuit:
% mflowgen run --demo
% cd mflowgen-demo
% mkdir build && cd build

We switch to the open-source toolflow and add the clock period parameter sweep:

```python
# Parameterize

g.update_params( parameters )
g.param_space( 'open-yosys-synthesis', 'clock_period', [0.5, 0.75, 1.0, 1.5] )
return g
```

Then we run mflowgen and see the four synthesis targets with different clock periods:

% mflowgen run --design ../GcdUnit
% make status
(...)
- build -> 3 : open-yosys-synthesis-clock_period-0.5
- build -> 4 : open-yosys-synthesis-clock_period-0.75
- build -> 5 : open-yosys-synthesis-clock_period-1.0
- build -> 6 : open-yosys-synthesis-clock_period-1.5
(...)

We build all of the yosys targets:

% make 3 4 5 6 -j4

Here are the delays:

% grep "Current delay" *yosys*/mflowgen-run.log

4-open-yosys-synthesis-clock_period-0.75/mflowgen-run.log:ABC: Current delay (670.68 ps) does not exceed the target delay (750.00 ps). Upsizing is not performed.
5-open-yosys-synthesis-clock_period-1.0/mflowgen-run.log:ABC: Current delay (670.68 ps) does not exceed the target delay (1000.00 ps). Upsizing is not performed.
6-open-yosys-synthesis-clock_period-1.5/mflowgen-run.log:ABC: Current delay (670.68 ps) does not exceed the target delay (1500.00 ps). Upsizing is not performed.

The reports here show that the critical path delay through GcdUnit is 0.67ns, which meets timing for the 0.75ns, 1.0ns, and 1.5ns clock targets. No upsizing is needed for any gates in these cases.

The 0.5ns clock period is not met. In this case, yosys-abc tries to upsize gates and does better (0.569ns) but still does not meet timing.

% less 3-open-yosys-synthesis-clock_period-0.5/mflowgen-run.log
This example sweep is not particularly complex and could easily be done manually (e.g., if only a few points are needed). However, sweeping the cross product of multiple parameters at once (e.g., architectural RTL design parameters, bitwidths) quickly becomes cumbersome. In this case, mflowgen parameter sweeps become more useful.

### 3.5 ADK Paths

ADKs can be located anywhere in the file system. Mflowgen uses a list of search paths to locate an ADK. For those familiar with how Python handles imports, this is nearly identical to how Python uses “sys.path” to locate Python packages. The default search path only contains the top-level “adks” directory in the mflowgen repo:

```python
from mflowgen.components import Graph
g = Graph()
p = g.sys_path
print(p)
['../adks']
```

mflowgen iterates through this list to locate the ADK you specify. For example, say this list had three search paths:

```python
['../adks', '/path/to/foo', '/path/to/bar']
```

And say we had set the adk to be “freepdk-45nm”:

```python
g.set_adk('freepdk-45nm')
```

Trying to locate the ‘freepdk-45nm’ ADK will result in mflowgen checking ..//adks/freepdk-45nm, /path/to/foo/freepdk-45nm, and /path/to/bar/freepdk-45nm in that order.

If mflowgen cannot find an adk at configure time, it will throw an OSError:

```bash
mflowgen run --design ../designs/GcdUnit
OSError: Could not find adk "foobar-45nm" in system paths: ['../adks']
```

There are two simple ways to add to the search path:

1. Using Python (i.e., in your construct.py)
2. Using the $MFLOWGEN_PATH environment variable (this is analogous to how Python uses the $PYTHONPATH environment variable):

```bash
% export MFLOWGEN_PATH=/path1:/path2:/path/to/adk/search/path
```

## 3.6 Assertions

Similar to how assertions can catch runtime exceptions in software, mflowgen allows you to define Python snippets that assert preconditions and postconditions before and after steps to catch unexpected situations at build time. Assertions are in Python to keep them concise and yet powerful. The assertions are collected and run with `pytest` to allow customization and user extensibility.

These assertions can be statically defined in a step configuration file or defined at graph construction time. For example, say we have a simple synthesis node with a configuration like this:

```yaml
name: synopsys-dc-synthesis
inputs:
  - adk # Technology files
  - design.v # RTL
outputs:
  - design.v # gate-level netlist
commands:
  - bash run.sh
```

We can assert that synthesis should not start unless it sees the technology files and the RTL design present. If either is missing, the build will not continue.

```text
preconditions:
  - assert File('inputs/adk') # Technology files must exist
  - assert File('inputs/design.v') # RTL must exist
```

Similarly, after synthesis has completed we can assert that the gate-level netlist exists and that there were no issues resolving references (a common synthesis error that breaks the build). Again, the build would stop if the postcondition were to fail.

```text
postconditions:
  - assert File('outputs/design.v') # Gate-level netlist must exist
  - assert 'Unable to resolve' not in File('logs/dc.log')
```

Each of these items is valid Python code and you can use Python any way you like to build your own assertions. For convenience, mflowgen natively provides a `File` class that overrides both boolean evaluation and containment, enabling the concise syntax you see here for checking whether or not a file exists as well as for using “in” and “not in” to search within a file. There is also a `Tool` class natively available that overrides boolean evaluation to concisely assert whether a tool exists or not.

```text
preconditions:
  - assert Tool('dc_shell-xg-t') # check for Design Compiler
```
3.6.1 The File Class and Tool Class

The File class internally handles boolean evaluation simply by calling `os.path.exists()`, so it can be used to check for existence of both files and directories.

Additional knobs are available to enable case sensitivity (default is case-insensitive) and regular expression search (default is disabled):

```python
>>> assert 'warning' in File('logs/dc.log', enable_case_sensitive=True)
>>> assert 'warn.*' in File('logs/dc.log', enable_regex=True)
```

The Tool class handles boolean evaluation using the `shutil.which()` function from the shell utilities Python library. This is equivalent to running `% which foo` on the command line.

```python
>>> assert Tool('dc_shell-xg-t') # This statement is roughly 
    % which dc_shell-xg-t # shell statement
```

3.6.2 Adding Assertions When Constructing Your Graph

The assertions defined in a step configuration file can be extended at graph construction time, meaning you can add your own design-specific assertions in each step. You can use the `Step.extend_preconditions` and `Step.extend_postconditions` methods to extend either list.

For example, say we wanted to add a check for clock-gating cells as a postcondition in our synthesis step. We can assert that this cell appears in the gate-level netlist like this:

```python
dc = Step('synopsys-dc-synthesis', default=True)
dc.extend_postconditions(["assert 'CKGATE' in File('outputs/design.v')"])
```

3.6.3 Escaping Special Characters

Certain characters are special in YAML syntax and must be escaped if you want to use them. For example, the following postcondition in the Mentor Calibre GDS merge step (i.e., “mentor-calibre-gdsmerge”) asserts that the report does not warn about duplicate module definitions (a dangerous warning that can corrupt your layout):

```yaml
postconditions:
  - assert 'WARNING: Ignoring duplicate structure' not in File('merge.log')
```

Unfortunately, the `:` character is a reserved character in YAML syntax since it is used for key-value stores (i.e., dictionaries in Python). The easiest way to escape this is not to explicitly escape the character, but to wrap the entire string in double quotes instead as shown below:

```yaml
postconditions:
  - "assert 'WARNING: Ignoring duplicate structure' not in File('merge.log')"
```

You can search for YAML syntax online to find more information on escaping characters in YAML files.

3.6.4 Multiline Assertions

Writing Python assertions in a single line of Python code can be very limiting. You can write assertions with multiple lines, but it requires using the YAML syntax for a block literal (i.e., a multiline string that preserves newline
characters):

```yaml
preconditions:
  - |
    import math
    assert math.pi > 3.0
```

Indentation matters in Python. Fortunately, YAML syntax uses the indentation of the first line after the `|` character to derive the indentation of all the following lines. So this entry correctly represents the following Python code:

```python
>>> import math
>>> assert math.pi > 3.0
```

The pytest function that mflowgen generates looks like this:

```python
def test_0_():
    import math
    assert math.pi > 3.0
```

Note that if you write a multiline entry without the `|` marker, YAML will simply wrap the lines as if there were no newlines:

```yaml
preconditions:
  - import math
    assert math.pi > 3.0
```

This is read as a single string, which is not valid Python:

```python
>>> import math assert math.pi > 3.0
```

### 3.6.5 Defining Python Helper Functions

You can provide your own Python helper functions to extract information about your build which you can use in assertions.

For example, suppose we want to assert that synthesis has successfully clock-gated the majority of registers in the design. The clock-gating report looks like this:

```
Clock Gating Summary
------------------------------------------------------------
| Number of Clock gating elements | 2 |
| Number of Gated registers      | 32 (94.12%) |
| Number of Ungated registers    | 2 (5.88%) |
| Total number of registers      | 34 |
------------------------------------------------------------
```

You can write a Python helper function that extracts the 94.12% figure:

```python
# assertion_helpers.py

# percent_clock_gated
#
# Reads the clock-gating report and returns a float representing the
```

(continues on next page)
```python
# percentage of registers that are clock gated.
#
def percent_clock_gated():
    # Read the clock-gating report
    with open(glob('reports/*clock_gating.rpt')[0]) as fd:
        lines = fd.readlines()
    # Get the line with the clock-gating percentage, which looks like this:
    gate_line = [l for l in lines if 'Number of Gated registers' in l][0]
    # Extract the percentage between parentheses
    percentage = float(re.search(r'(.*?%)', gate_line).group(1))/100
    return percentage

Then you can assert a postcondition in the step configuration for a clock-gating percentage of at least 80%:

```postconditions:
    # Check that at least 80% of registers were clock-gated
    - from assertion_helpers import percent_clock_gated
      assert percent_clock_gated() > 0.80
```

### 3.6.6 Using Custom pytest Files

You can write your own pytest functions and include them in your Step (or attach them as inputs). Then you can drop them in your step configuration file using the `pytest:` key as special syntax:

```preconditions:
    - pytest: test_foo.py
    - pytest: inputs/test_bar.py
```

These tests will then be collected and automatically run with all the other assertions.

### 3.6.7 Assertion Scripts in mflowgen

When executing a step, mflowgen generates two scripts, `mflowgen-check-preconditions.py` and `mflowgen-check-postconditions.py`, puts them in the build directory, and then runs these scripts before and after executing the step. At runtime if the postcondition check fails, re-running the step (e.g., `make 4`) will only re-run the postcondition check. It will not re-execute the step. This gives you the chance to enter the sandbox and fix things until the postconditions pass. The build status will not be marked “done” until all postcondition checks pass.

**Note:** To completely re-run a step, you should clean that step. For example if synthesis is step 4, `make clean-4` and `make 4` will do a clean rebuild of synthesis.
The two assertion scripts can also be run independently with pytest. The example below shows a precondition assertion firing and saying that Synopsys Design Compiler (i.e., dc_shell-xg-t) is missing. You can re-run the check yourself with default pytest options:

```bash
% cd 4-synopsys-dc-synthesis
% ./mflowgen-check-preconditions.py

> Checking preconditions for step "synopsys-dc-synthesis"

pytest -q -rA --disable-warnings --tb=no --color=no ./mflowgen-check-preconditions.py
F... → [100%]
==================================== short test summary info
====================================
PASSED mflowgen-check-preconditions.py::test_1_
PASSED mflowgen-check-preconditions.py::test_2_
PASSED mflowgen-check-preconditions.py::test_3_
FAILED mflowgen-check-preconditions.py::test_0_ - AssertionError: assert Tool( 'dc_shell-xg-t' )
1 failed, 3 passed in 0.05s
```

Or you can call pytest explicitly with your own arguments for a longer traceback (although this traceback does not say very much):

```bash
% cd 4-synopsys-dc-synthesis
% pytest -q --tb=short mflowgen-check-preconditions.py

F... → [100%]
================================== FAILURES
==================================
mflowgen-check-preconditions.py:44: in test_0_
    assert Tool( 'dc_shell-xg-t' )
E   AssertionError: assert Tool( 'dc_shell-xg-t' )
E   + where Tool( 'dc_shell-xg-t' ) = Tool('dc_shell-xg-t')
================================== short test summary info
==================================
FAILED mflowgen-check-preconditions.py::test_0_ - AssertionError: assert Tool( 'dc_shell-xg-t' )
1 failed, 3 passed in 0.17s
```

3.7 Stashing Pre-Built Steps for Sharing

mflowgen supports stashing to help you share pre-built steps between people working on the same graph.

For example, here is a “make status” printout for the GcdUnit design with most steps already built:
At the command line, you can run “mflowgen stash help” to see help messages on how to use stash commands.

We will walk through a simple example here. First you will choose a place to put your stash:

```
% mflowgen stash init --path /tmp
Linked to stash: /tmp/2020-0316-mflowgen-stash-972c83
```

The stash contents are empty at the start:

```
% mflowgen stash list
Stash List
- ( the stash is empty )
Stash: /tmp/2020-0317-mflowgen-stash-588e56
```

We can stash our synthesis step with a message like this:

```
% mflowgen stash push --step 4 -m "Pushing synthesis as a test"
Stashed step 4 "synopsys-dc-synthesis" as author "ctorng"
```

**Note:** Only the outputs of a step are stashed by default. Saving the outputs is often far smaller compared to saving the entire build of a step. You can use the optional --all flag to save an entire step, including all of its inputs, logs, and intermediate files. Note that this can be very slow if there are many small files to copy.

Now the stash contents show the pre-built synthesis step tagged with a “4d1c23” hash:

```
% mflowgen stash list
Stash List
- 4d1c23 [ 2020-0316 ] ctorng synopsys-dc-synthesis -- Pushing synthesis as a test
```

(continues on next page)
You can stash other steps, and you can stash the same step multiple times (they all get a different hash in the stash for uniqueness).

Now say we cleaned our copy of synthesis for whatever reason:

```
% make clean-4 # Delete our copy of synthesis
% make status
```

```
Status:
- done -> 0 : constraints
- done -> 1 : freepdk-45nm
- build -> 2 : info
- done -> 3 : rtl
- build -> 4 : synopsys-dc-synthesis
- build -> 5 : cadence-innovus-flowsetup
- build -> 6 : cadence-innovus-init
- build -> 7 : cadence-innovus-power
- build -> 8 : cadence-innovus-place
- build -> 9 : cadence-innovus-cts
- build -> 10 : cadence-innovus-postcts_hold
- build -> 11 : cadence-innovus-route
- build -> 12 : cadence-innovus-postroute
- build -> 13 : cadence-innovus-postroute_hold
- build -> 14 : cadence-innovus-signoff
- build -> 15 : mentor-calibre-gdsmerge
- build -> 16 : synopsys-ptpx-genlibdb
- build -> 17 : mentor-calibre-drc
- build -> 18 : mentor-calibre-lvs
- build -> 19 : cadence-innovus-debug-calibre
```

So everything from synthesis and onwards is marked not done. We can pull the pre-built stashed copy like this:

```
% mflowgen stash pull --hash 4d1c23
Pulled step "synopsys-dc-synthesis" from stash into "4-synopsys-dc-synthesis"
```

```
% make status
```
Synthesis is now back and it is marked “pre-built”, meaning it is forced always up-to-date until you remove it or run its clean target. Pulling does not remove the step from the stash. You can pull the same stashed step as many times as you want. Pulling will overwrite the existing step of the same name in your current directory. You can also “stash pop” to pull a pre-built step and then drop it from the stash.

Note: The mflowgen stash commands mimic those from git stash. However, mflowgen stashes are meant for sharing stashed copies with yourself and also with other users, while git stashes seem more intended for temporarily “shelving” your own versions for yourself to use.

It can be particularly useful for teams to share pre-built steps to enable others to start from an intermediate point in a fresh build. For example, we could stash pull on synthesis and continue onwards from there to do PnR without re-executing any of the earlier steps in the flow.

First we link a build to a stash directory:

```plaintext
# Some other person
% cd build-x
% mflowgen stash link --path /tmp/2020-0316-mflowgen-stash-972c83
```

Then we pull:

```plaintext
% mflowgen stash pull --hash 4d1c23
% make status
```
The upstream nodes are not built yet, but the pre-built synthesis step is forced up-to-date (until cleaned). This means we can continue the flow from this point and never worry about managing the pre-built copy or its predecessors.

Also note that you can drop something from the stash like this:

```
% mflowgen stash drop --hash 4d1c23
Dropped step "synopsys-dc-synthesis" with hash "4d1c23"
```

As a final note, be aware that some steps cannot be shared if they contain hardcoded paths, which may break when executed from another location. Ideally, steps should be designed to be as portable as possible, but this is not always feasible.

### 3.8 Mock Graphs for Modular Step Development

Developing a step can be difficult when its inputs are generated as part of a larger flow and you are not yet sure what the commands should be. In this situation, it may not make sense to hook a partially developed step into a full flow to debug the issues. Instead, you can tell mflowgen to mock up a graph where you can provide inputs and test the step in isolation.

The mock-up contains the “design-under-test” node and a “mock-push” node that automatically provides placeholder outputs matching your step’s inputs. For example, the “synopsys-dc-synthesis” node in its mock-up graph would look like this:
You could make sure your step works by spinning in a normal mflowgen environment with full access to normal build targets (e.g., make status). You would provide inputs and iterate on developing your step. You could test the commands, the inputs and outputs, the preconditions and postconditions, and the parameters before connecting it in a larger flow.

As an example, say we are developing and testing a new step “test” that simply reads a file (i.e, “input-foo”) and generates a file (i.e, “output-bar”). Its step configuration file looks like this:

```yaml
# test/configure.yml

name: test

inputs:
  - input-foo

commands:
  - mkdir -p outputs
    - cat inputs/input-foo > outputs/output-bar

outputs:
  - output-bar
```

You can tell mflowgen to mock up a graph for this step:

```bash
% ls
  test

% mkdir build && cd build
% mflowgen mock init --path ../test
```

The status and information reflects the tiny mock-up graph:
Running the “make graph” target would generate this graph:
The mock files in “mock-push/outputs” can be replaced with real files. You could then clean and re-run your step as you develop. Remember to pull your changes back into the source copy of your step.

3.9 Static Checks

UNDER CONSTRUCTION

Mflowgen nodes are typically designed to reused across different designs, different technologies, or both. When a designer attempts to reuse an mflowgen node, we want to provide early and rapid feedback on whether or not that node is being used properly in the new flow. However, long CAD tool runtimes make runtime assertions insufficient for this purpose, as designers would have to wait for the flow to reach the code fragment of interest, which could take hours or even days.

Mflowgen static checks are designed to provide early and rapid feedback on whether or not a reusable mflowgen node is being reused properly by running static program analysis at graph elaboration time to flag any inconsistencies, instead of actually running any CAD tools.

3.9.1 Intent-Implementation Split

The intent-implementation split is one of mflowgen’s built-in static checks. Using the schema below, designers can separately express the intent of a particular code fragment (along with formal properties which any implementation of that intent must follow), from a potentially design- or technology-specific implementation of that intent.
Here’s an example where the intent-implementation split is used to add power domains to 2 different blocks while reusing the same power domain intent.

Specifically, this power domains strategy involves the creation of an AON region within the block’s floorplan. It’s the same concept regardless of which block it’s placed in, but the AON region must be placed carefully to avoid a variety of DRC and/or LVS errors.

Here’s how a designer would write the intent block for drawing this AON region. The intent runs the necessary tool command, accepts parameters from an implementation as inputs, and contains a set of properties that any implementation must obey.

```
proc mflowgen.intent.createAON { $aon_llx $aon_lly $aon_urx $aon_ury $x_gap $y_gap } { 
    modifyPowerDomainAttr AON \
       -box $aon_llx $aon_lly $aon_urx $aon_ury \ 
       -minGaps $aon_y_gap $aon_y_gap $aon_x_gap \ 
       $aon_x_gap
    array set mflowgen.property.even_y { 
        property "((aon_lly//y_pitch) % 2 == 0) & ((aon_ury//y_pitch) % 2 == 0)" 
        describe "The AON region must begin and end on an even numbered standard cell row to prevent obstructed power switches and LUP DRCs"
    }
    array set mflowgen.property.max_width { 
        property "(aon_urx - aon_llx) < horiz_switch_pitch)" 
        describe "The always on region must be narrower than the pitch between power switch pitches so that no more than 1 column of switches is obstructed"
    }
    array set mflowgen.property.away_from_left_edge { 
        property "aon_llx > (M3_str_offset + 2*M3_str_pitch)" 
        describe "AON region must be far enough from left edge to allow space for at least one set of M3 VDD/VSS stripes"
    }
    array set mflowgen.property.away_from_right_edge { 
        property "aon_urx < (fp_width - M3_str_offset - 2*M3_str_pitch)" 
        describe "AON region must be far enough from right edge to allow space for at least one set of M3 VDD/VSS stripes"
    }
    array set mflowgen.property.M3_stripe_obstruction_left { 
        property "math.fmod(aon_llx - M3_str_offset, M3_str_pitch) > (2 * M3_str_width)"
    }
} 
```
describe "Left edge of AON region cannot overlap with M3 stripe"
)
array set mflowgen.property.M3_stripe_obstruction_right {
    property "math.fmod(aon_urx - M3_str_offset, M3_str_pitch) > (2 * M3_str_width)"
    describe "Right edge of AON region cannot overlap with M3 stripe"
}

An implementation must provide all of the inputs that the intent requires. Here’s an example implementation for creating the AON region in a given block:

```tcl
proc mflowgen.implement.createAON {} {
    # Params
    set x_offset 3
    set aon_width 14
    set aon_height 10
    set lly 50
    set minGap 1

    # Place AON region in middle of tile (+ x_offset) on X-axis
    set aon_llx [expr $core_width/2 - $aon_width/2 + $x_offset]
    set aon_urx [expr $core_width/2 + $aon_width/2 + $x_offset]
    set aon_lly $lly
    set aon_ury [expr $aon_lly_coord + $aon_height]

    # Snap coords to placement grid
    set aon_llx_snap ...
    set aon_lly_snap ...
    set aon_ury_snap ...
    set aon_urx_snap ...
    set aon_ury_snap ...

    return [list $aon_llx_snap $aon_lly_snap $aon_urx_snap $aon_ury_snap $minGap]
}
```

As you can see, this implementation provides values for all of the parameters required by the createAON intent. Now that we have both an implementation and an intent for this feature, we can run the code fragment like this:

```tcl
mflowgen.intent.createAON {*}[mflowgen.implement.createAON]
```

All of the parameter values are passed calculated by the implementation are passed to the intent, and they are all checked for consistency at graph elaboration time, not tool runtime. This allows for immediate feedback if a given implementation is incorrect and greatly shortens the debug loop for any designer trying to reuse this code.

Here are 2 separate blocks that reused the same createAON intent, but with different implementation blocks.
This section walks through the common library included in mflowgen, using standard commercial tools to synthesize an RTL design, place and route the gates, generate a layout, and verify it with a set of signoff steps.

This section provides a high-level overview of running an entire pipe cleaner before doing deeper dives into the open-source technology files, submodular node organization, the DC synthesis node, the Innovus Foundation Flow, the Innovus nodes, how to run each step, and which scripts and reports are the most important to inspect.

Pipe cleaners are small designs that run through the flow quickly and help to identify errors early. It is good practice to frequently run pipe cleaners while developing the flow before running your full design. We will be using the GcdUnit design as a pipecleaner.

### 4.1 Greatest Common Divisor Pipe Cleaner

The GcdUnit module computes the greatest common divisor function and is a great design that has only 100-200 gates to quickly spin through the flow. Importantly, it contains only standard cells and no memory macros, greatly decreasing the complexity of the design. The datapath includes two state registers and the required muxing and arithmetic units to iteratively implement Euclid’s algorithm. The diagram below shows a hybrid Moore/Mealy FSM for controlling the datapath. Mealy transitions in the calc state determine whether to swap or subtract.

To start the GcdUnit pipe cleaner, first start the built-in demo:

```bash
% mflowgen run --demo
% cd mflowgen-demo
% top=$(pwd)
```

Then create a build directory and configure for this design:

```bash
% cd $top
% mkdir build && cd build
% mflowgen run --design ../GcdUnit
% make list
```
Then open the mflowgen graph to see what this pipe cleaner flow looks like:

```bash
% make graph
(open graph.pdf in a PDF viewer like evince)
% evince graph.pdf
```

Here is the list of steps:

```bash
% make list
- 0 : freepdk-45nm
- 1 : rtl
- 2 : info
- 3 : constraints
- 4 : testbench
- 5 : rtl-sim
- 6 : gen-saif-rtl
- 7 : synopsys-dc-synthesis
- 8 : cadence-innovus-flowsetup
- 9 : cadence-innovus-init
- 10 : cadence-innovus-power
- 11 : cadence-innovus-place
- 12 : cadence-innovus-cts
- 13 : cadence-innovus-postcts_hold
- 14 : cadence-innovus-route
- 15 : cadence-innovus-postroute
- 16 : cadence-innovus-signoff
- 17 : mentor-calibre-gdsmerge
- 18 : synopsys-pt-timing-signoff
- 19 : synopsys-ptpx-rtl
- 20 : synopsys-ptpx-genlibdb
- 21 : mentor-calibre-drc
- 22 : gl-sim
- 23 : mentor-calibre-lvs
- 24 : gen-saif-gl
- 25 : synopsys-ptpx-gl
```

Feel free to cross-check the construct.py, the graph visualization, and the step configuration files to see which files are passing between which steps.

You will see that the GcdUnit RTL, a constraints file, and the technology interface from freepdk-45nm passes into the “synopsys-dc-synthesis” node. Compare this graph to the Python description of the graph in GcdUnit/construct.py.

Here is a high-level overview of the Innovus steps. The `cadence-innovus-flowsetup` node first uses a script generator to generate the base scripts used in each of the downstream Innovus nodes (also see the section on the Cadence Innovus Foundation Flow later in this handout). The `cadence-innovus-init` step starts place and route by reading in the design. Each of the subsequent nodes reads in a checkpoint (i.e., an Innovus database), does something (e.g., runs placement), and outputs another checkpoint. At the end of the flow, the `cadence-innovus-signoff` step dumps all outputs of place and route. These outputs include the final gate-level netlist, GDS, and other files, which are used to run further signoff steps (e.g., Calibre DRC and LVS). Notice that a GDS merge step is needed to merge the GDS from place and route (which includes only the wires) with the GDS of the standard cell library.

You can run the GcdUnit pipe cleaner through to signoff like this:

```bash
% make list # check which step signoff is
% make 16 # assuming signoff is step 16
% make status # signoff should be marked done
```

Each of the Innovus steps can be brought up on a GUI with a debug target like this:
\% make debug-16 \# brings up Innovus GUI after signoff

You should also get a sense of the runtime of each step:
\% make runtimes

### 4.2 FreePDK45 and the Nangate Open Cell Library

The FreePDK45 kit is an open-source generic process design kit (PDK) (i.e., does not correspond to any real process and cannot be fabricated) that allows researchers and students to experiment with designing in a modern technology node without signing restrictive non-disclosure agreements or paying for licenses. The PDK allows you to use commercial full-custom layout tools (e.g., Cadence Virtuoso) to design both analog and digital circuits. Later in our flow, we will be leveraging the Calibre design-rule check (DRC) and layout-vs-schematic (LVS) rule decks that come with the FreePDK45 kit to verify our design. You can find more details about FreePDK45 here.

Digital synthesis tools work with libraries of pre-drawn and pre-characterized standard cells (often provided by IP vendors like ARM). The Nangate Open Cell Library is a generic open-source digital standard-cell library designed using the FreePDK45 kit. For example, here is the layout for a NAND_X3 standard cell:

![Layout of NAND_X3 standard cell](image)

You can see the VDD rail (blue strip at the top), the VSS rail (blue strip at the bottom), the poly (red vertical strips), contacts (dark blue squares), and input pins on M1 (three dark blue squares that connect to the poly gates). The physical database containing these layout shapes is known as a graphic database system (GDS) and can be found in “stdcells.gds” in the base kit here.

Each cell in the library is characterized with a large number of SPICE simulations and the timing information is captured in the “stdcells.lib” and “stdcells.db” files (the same information is captured in both, just different file formats). You can find more details about this library here. The liberty (.lib) file is human-readable and you can directly read it in the base kit here.

The Nangate library also comes with the routing technology kit (RTK) technology LEF, often called the “tech LEF”. This describes all available layers that the synthesis and place-and-route tools can use for routing. This file is also human-readable and you can read it in the base kit here. For example, the “metal1” parameters look like this:

```
LAYER metal1
  TYPE ROUTING ;
```

(continues on next page)
4.3 Sub-Modular Node Design

The mflowgen steps we provide for DC synthesis and Innovus place-and-route steps are designed to be submodular. This means that each step (which is already modular on its own) is further split into individual scripts for various purposes (e.g., reporting, setting up variables and margins, tweaking constraints). An mflowgen parameter called `order` then allows the user to parameterize how the step is run from a Python interface. For example, the user can remove a script that is not needed for their design, add an additional custom script for their design, or simply define a reordering of the existing scripts.

Take a look at the design initialization step (i.e., `cadence-innovus-init`), which is responsible for reading in the post-synthesis design files from Synopsys DC and executing floorplanning. This step configuration file lists the following inputs and outputs:

<table>
<thead>
<tr>
<th>input</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>innovus-foundation-flow</td>
<td>Scripts generated by the Innovus Foundation Flow.</td>
</tr>
<tr>
<td>adk</td>
<td>ASIC design kit interface to the process technology and library files.</td>
</tr>
<tr>
<td>design.v</td>
<td>The post-synthesis gate-level netlist.</td>
</tr>
<tr>
<td>design.sdc</td>
<td>Constraints dumped from synthesis.</td>
</tr>
<tr>
<td>design.checkpoint</td>
<td>The working Innovus database after the step finishes.</td>
</tr>
</tbody>
</table>

The configuration file also defines the `order` parameter, which lists scripts in the following order (note that the contents of these scripts can be found in the `scripts` subdirectory):

<table>
<thead>
<tr>
<th>script</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.tcl</td>
<td>Main script from the Innovus Foundation Flow for design initialization.</td>
</tr>
<tr>
<td>quality-of-life.tcl</td>
<td>Useful variables and settings when working with Innovus.</td>
</tr>
<tr>
<td>floorplan.tcl</td>
<td>Creates the floorplan with the core area and all margins.</td>
</tr>
<tr>
<td>pin-assignments.tcl</td>
<td>Places IO pins along the perimeter of the design.</td>
</tr>
<tr>
<td>make-path-groups.tcl</td>
<td>Creates timing path groups to help the timing engine prioritize certain paths.</td>
</tr>
<tr>
<td>reporting.tcl</td>
<td>Dumps a variety of reports.</td>
</tr>
</tbody>
</table>

The order parameter determines the order in which the scripts are run. For example, floorplanning (`floorplan.tcl`) will be executed after we read in the design (`main.tcl`) and set up quality-of-life variables (`quality-of-life.tcl`), with reporting done at the end of the step (`reporting.tcl`). The order parameter can be accessed as a normal python list in your mflowgen graph defined in construct.py. For example, the parameter can be printed like this:

```python
>>> init = Step( 'cadence-innovus-init', default=True )
>>> order = init.get_param( 'order' )
>>> print( order )
```
The commands run Innovus with a special script that simply loops through the order parameter and runs each item one by one. It searches for each script in both the step’s local scripts directory (for the default versions of the scripts) as well as the inputs directory (where the user can supply custom versions of the scripts) with priority given to the inputs. This allows users to extend the step with new scripts.

**Note:** If the same script is found in both the scripts and inputs directories, priority is given to the copy in the inputs directory.

For example, suppose that “scripts/foo.tcl” exists and the flow designer then supplies “inputs/foo.tcl”. In this case, “inputs/foo.tcl” is run and “scripts/foo.tcl” is ignored.

For example, if we had a node called custom-init that provided new-last-step.tcl as an output, we could parameterize the init step to append the new script at the end:

- main.tcl
- quality-of-life.tcl
- floorplan.tcl
- pin-assignments.tcl
- make-path-groups.tcl
- reporting.tcl
- new-last-step.tcl (a new custom script)

The mflowgen construct script would use the mflowgen API for extending the list of inputs in a step (i.e., extend_inputs()) and extend the init step inputs with the list of all outputs of our custom node (i.e., all_outputs()). Running make graph would then produce a graph with a new input edge into the init step like this:

```
init = Step( 'cadence-innovus-init', default=True )
custom_init = Step( ... ) # Comes from somewhere and
# has an output 'new-last-step.tcl'
```

4.3. Sub-Modular Node Design
Finally, we would parameterize the order of the init step to include the new input script:

```python
order = init.get_param( 'order' )
order.append( 'new-last-step.tcl' )  # Append the script to run last
init.update_params( {'order': order} )

print( order )  # -> [ 'main.tcl', ..., 'new-last-step.tcl']
```

Note that we could simply have replaced the entire init node with our own custom version. However, it can be very useful to reuse the majority of an existing step while tweaking just a small part of it.

### 4.4 Synthesis

We use Synopsys DC to synthesize a single RTL netlist file into gates. You can run the design up to this step like this:

```
% cd $top/build
% make synopsys-dc-synthesis
```

Here are the inputs, outputs, and scripts in the synthesis step and what they do.

<table>
<thead>
<tr>
<th>input</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adk</td>
<td>ASIC design kit interface to the process technology and library files.</td>
</tr>
<tr>
<td>design.v</td>
<td>The RTL design in Verilog/SystemVerilog.</td>
</tr>
<tr>
<td>constraints.tcl</td>
<td>Constraints on the design (e.g., input/output delays, min/max delays).</td>
</tr>
<tr>
<td>run.saif</td>
<td>An optional SAIF activity file for RTL-driven power estimation.</td>
</tr>
<tr>
<td>output</td>
<td>description</td>
</tr>
<tr>
<td>design.v</td>
<td>The post-synthesis gate-level netlist.</td>
</tr>
<tr>
<td>design.sdc</td>
<td>Constraints dumped from synthesis.</td>
</tr>
<tr>
<td>design.namemap</td>
<td>An optional name mapping for RTL nets and matching post-synth nets.</td>
</tr>
</tbody>
</table>

After running synthesis, feel free to enter the synthesis sandbox (e.g., `4-synopsys-dc-synthesis`) and find the following files in the reports directory:

- GcdUnit.mapped.area.rpt
- GcdUnit.mapped.power.rpt
- GcdUnit.mapped.qor.rpt
- GcdUnit.mapped.timing.setup.rpt
- GcdUnit.premapped.checkdesign.rpt

The quality-of-results (QoR) report is the most useful and summarizes both timing and area. The area report is a hierarchical breakdown of the module based on the area numbers reported for each gate (from the stdcells.lib). The power report is a *high-level estimate* of the power of your design assuming about 10% switching activity on all nets (not very accurate, but a good first-order estimate). The setup timing report contains the full trace for the most critical timing paths, with the critical path listed first.

There are also many parameters that can be used to slightly tweak the behavior of this node. Here are the key parameters:
• **design_name** – *(string)* Target a particular module within the input RTL

• **clock_period** – *(float)* Clock target in library time units

Here are other useful parameters:

• **flatten_effort** – *(int)* Flatten effort “0” is strict hierarchy, and effort “3” is full flattening. Default = 0.

• **topographical** – *(bool)* Enable DC topographical mode, which does mini placements at synthesis time to estimate wire delays more accurately. Default = True.

• **nthreads** – *(int)* The maximum number of threads given to DC. Note that DC is not always able to make use of all threads, even if they are available. Default = 16.

• **high_effort_area_opt** – *(bool)* Tell DC to do an additional post-compile area optimization pass (has longer spin time). Default = False.

• **gate_clock** – *(bool)* Automatic fine-grain clock gating. Default = True.

• **uniquify_with_design_name** – *(bool)* Uniquify by prefixing every module in the design with the design name. This is useful for hierarchical LVS when multiple blocks use modules with the same name but different definitions. Default = True.

### 4.5 The Innovus Foundation Flow

**Relevant mflowgen step:** `cadence-innovus-flowsetup`

Cadence Innovus comes with its own flow generator called the Innovus Foundation Flow. The mflowgen steps that we provide for Innovus use the foundation flow to generate a base set of scripts that execute each of the major steps in place and route (e.g., init, place, cts, route, postroute, signoff). Each script is connected to a downstream mflowgen step that wraps the script and enhances it for convenience and debuggability.

There is a major benefit to relying on the Cadence Innovus Foundation Flow for the canonical commands. As Cadence updates its version of Innovus year after year, their recommended options to achieve more optimal quality of results changes. Some options are deprecated, other options are newly suggested, and new commands are introduced. Using the foundation flow generator allows us to easily leverage the most up-to-date, Cadence-recommended commands with no maintenance costs of our own.

You can dump the foundation flow yourself by running the `writeFlowTemplate` command in an Innovus shell. This is essentially what the `cadence-innovus-flowsetup` node is responsible for (see the step configuration file). Feel free to open Innovus and run it yourself:

```
$ innovus
>>> writeFlowTemplate
```

The generator will dump files into the current directory. The master script is `SCRIPTS/gen_flow.tcl`. The `cadence-innovus-flowsetup` mflowgen node runs this master script and generates the base scripts for the following steps:

<table>
<thead>
<tr>
<th>Step</th>
<th>Script Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design initialization</td>
<td><code>innovus-foundation-flow/INNOVUS/run_init.tcl</code></td>
</tr>
<tr>
<td>Placement</td>
<td><code>innovus-foundation-flow/INNOVUS/run_place.tcl</code></td>
</tr>
<tr>
<td>Clock tree synthesis (CTS)</td>
<td><code>innovus-foundation-flow/INNOVUS/run_cts.tcl</code></td>
</tr>
<tr>
<td>Post-CTS hold-fixing</td>
<td><code>innovus-foundation-flow/INNOVUS/run_postcts_hold.tcl</code></td>
</tr>
<tr>
<td>Route</td>
<td><code>innovus-foundation-flow/INNOVUS/run_route.tcl</code></td>
</tr>
<tr>
<td>Postroute</td>
<td><code>innovus-foundation-flow/INNOVUS/run_postroute.tcl</code></td>
</tr>
<tr>
<td>Signoff</td>
<td><code>innovus-foundation-flow/INNOVUS/run_signoff.tcl</code></td>
</tr>
</tbody>
</table>
The output of this node is the entire `innovus-foundation-flow` directory, which feeds into all downstream Innovus nodes. Feel free to open each of these scripts to see how the canonical place and route commands are run. Pay particular attention to the super commands (e.g., `placeDesign`, `optDesign`).

## 4.6 Design Initialization and Floorplanning

The first step during place and route is called `init` (i.e., the `cadence-innovus-init` step) and reads in the design from synthesis before executing floorplanning. You can run the design up to the init step like this:

```
% cd $top/build
% make cadence-innovus-init
```

Refer back to *Sub-Modular Node Design* to see what the inputs, outputs, and scripts are and what they do. Here is a list of scripts you might tweak at this step:

- **floorplan.tcl** – This is where we size the core area (targeting a certain density) and aspect ratio. This is also where we place macros. A reasonable density target is about 75%, and a very aggressive density is over 90%. Aspect ratios that are square are generally preferred over long rectangular ones. Rectangular floorplans place more stress on either horizontal or vertical routing resources and also make it difficult to place macros. Note that `planDesign` is the command that automatically places your macros.

- **pin-assignments.tcl** – This is where we use the “editPin” command to spread pins along the sides on specific metal layers. We could also read a pre-saved io file in this script.

- **make-path-groups.tcl** – You can create path groups to tell the timing engine to prioritize certain paths that you may find more important than others (e.g., paths to macros). During timing optimization, the tool loops through each of the path groups and tries to fix the worst paths in each group. If there were only a single path group, the tool might never work on paths further down the list that you as a designer know are important. Enough useful path groups have been set for you that you will likely not need to add any more.

Here is a list of checks you will want to run through before moving on to the next step:

- **reports/preplace.summary.gz** – The timing summary report at the end of init assumes no wire delays. This means that the timing reports should look nearly identical to those from synthesis. Therefore, the first thing you should check is that the timing is not be significantly better or worse than in synthesis. When wire delays are factored in, these numbers will only look worse, so the tool should not be trying too hard to meet timing at this point. In particular, there should **never** be negative slack in this report.

```
+--------------------+---------+---------+---------+---------+
| Setup mode | all | req2reg | reg2cgate| default |
+--------------------+---------+---------+---------+---------+
| WNS (ns):| 0.636 | 0.636 | 1.244 | 0.759 |
| TNS (ns):| 0.000 | 0.000 | 0.000 | 0.000 |
| Violating Paths:| 0 | 0 | 0 | 0 |
| All Paths:| 56 | 34 | 2 | 54 |
+--------------------+---------+---------+---------+---------+
|analysis_default | 0.636 | 0.636 | 1.244 | 0.759 |
| | 0.000 | 0.000 | 0.000 | 0.000 |
| | 0 | 0 | 0 | 0 |
| | 56 | 34 | 2 | 54 |
```
(continues on next page)
Density: 0.000%

- `logs/run.log` – There should be no errors in the logs. Most possible errors at this stage will be related to reading the technology files and libraries.

- Check the floorplan in the GUI – Run the debug target on this step and bring up the GUI. GcdUnit does not have any macros, but this is generally the time to make sure that the macros are placed at reasonable locations and orientations (e.g., pins facing the core area, no narrow channels). It can be useful to do a fast prototype of cell placement to see what the design will look like with stdcells. A floorplan mode placement does this by placing cells while ignoring DRCs for speed:

```
> setPlaceMode -place_design_floorplan_mode true
> place_design
```

- You can also open the Design Hierarchy viewer and click on a module to see where the stdcells in a particular module were placed. For example, the control module for GcdUnit was randomly placed in a clump near the top in this floorplan-mode prototype placement.

4.7 Power Strategy

The next step is the power strategy (i.e., the `cadence-innovus-power` step), which sets up the power rings, stripes, and grid. You can run the design up to this step like this:

```
% cd $top/build
% make cadence-innovus-power
```

Here is a list of what the defaults inputs, outputs, and scripts are and what they do:
The working Innovus database from the previous step.

The working Innovus database after the step finishes.

Logically connects power nets (e.g., VDD and VSS) to pins on cells in the design, even if their pins are not explicitly called VDD or VSS.

Main script that calls a power strategy.

Builds a fine mesh on lower metals for di/dt and a coarse mesh on higher metals for IR drop (you will not use this script in 45nm).

Builds a single power mesh on higher metals to minimize IR drop.

To iterate on your power strategy, we highly recommend opening the GUI (running the debug target for init gives you a clean slate) and copy-pasting the power commands one by one while tweaking them. You should press U to undo if you do not like the result of an addStripe or addRing command. You can also just type the undo command. If you want to start over, you can run the following command to delete power-related shapes:

\[ > \text{deleteAllPowerPreroutes} \]

At any point in your work, you can run an Innovus DRC which only checks the shapes that Innovus can see (i.e., wires). After opening Verify DRC just accept the default settings. You can also run an Innovus LVS with Verify Connectivity, although this will always fail at this stage because the cells and wires do not exist yet which certainly does not match the source netlist.

The violation browser is the red triangle button and lists all violations found by these checks. This view should be clear of DRC violations.
4.8 Placement

The next step is placement (i.e., the `cadence-innovus-place` step), which iterates on different placements with the global router. You can run the design up to this step like this:

```
% cd $top/build
% make cadence-innovus-place
```

Here are the inputs, outputs, and scripts in the place step and what they do.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>design.checkpoint</td>
<td>The working Innovus database from the previous step.</td>
</tr>
<tr>
<td>Output</td>
<td>Description</td>
</tr>
<tr>
<td>design.checkpoint</td>
<td>The working Innovus database after the step finishes.</td>
</tr>
<tr>
<td>Script</td>
<td>Description</td>
</tr>
<tr>
<td>setup-cellpad.tcl</td>
<td>Sets up padding next to each DFF to reserve space for buffers during timing optimization. Without padding in very dense designs, there may not be enough space left for timing-fixing buffers.</td>
</tr>
<tr>
<td>main.tcl</td>
<td>Calls the Innovus Foundation Flow script.</td>
</tr>
<tr>
<td>clean-cellpad.tcl</td>
<td>Removes the padding constraint after placement is over.</td>
</tr>
</tbody>
</table>

Here is a list of checks you will want to run through before moving on to the next step:

- `reports/place.summary` – With wires in place, the timing will be worse than it was in init and synthesis. The timing must look good in this report. The goal of all future steps is only to `preserve` the timing in this report.
- `logs/run.log` – Look for the final congestion analysis table in the log. Make sure that the overflows are at most a few percent. If the routing tracks are too oversubscribed (across all GCells in the design), the resulting
congestion will make timing very hard to meet. The table below is nearly clean because there is no congestion in GcdUnit.

<table>
<thead>
<tr>
<th>Layer</th>
<th>%Gcell</th>
<th>OverCon</th>
</tr>
</thead>
<tbody>
<tr>
<td>metal1</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>metal2</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>metal3</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>metal4</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>metal5</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>metal6</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>metal7</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0.02%</strong></td>
<td><strong>0.02%</strong></td>
</tr>
</tbody>
</table>

Overflow after earlyGlobalRoute (GR compatible) 0.00% H + 0.00% V
Overflow after earlyGlobalRoute 0.00% H + 0.00% V

- Check density and congestion overlays in the GUI – Open the debug target for place. Then enable overlays as shown in the following figure. This is a visual version of what you can already find in the logs.

4.9 Clock Tree Synthesis

The next step is clock tree synthesis (i.e., the `cadence-innovus-cts` step), which skew-balances the clock tree. You can run the design up to this step like this:

```bash
% cd $top/build
% make cadence-innovus-cts
```

Here are the inputs, outputs, and scripts and what they do. Again, you should not need to change any of these.
<table>
<thead>
<tr>
<th>input</th>
<th>design.checkpoint</th>
<th>The working Innovus database from the previous step.</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>design.checkpoint</td>
<td>The working Innovus database after the step finishes.</td>
</tr>
<tr>
<td>script</td>
<td>setup-ccopt.tcl</td>
<td>These are options for clock tree optimization that you will probably not need to touch.</td>
</tr>
<tr>
<td>script</td>
<td>main.tcl</td>
<td>Calls the Innovus Foundation Flow script.</td>
</tr>
</tbody>
</table>

Here is a list of checks you will want to run through before moving on to the next step:

- Look at the clock tree in the GUI – You can check boxes in the right panel of the GUI to show just the clock tree. This can give you a good visual sense of how the clock is being distributed. Notice how the clock enters from the middle pin of the left side in the GcdUnit layout. If your clock pin was placed in an odd corner during floorplanning, it could affect your clock tree.

- Look at the clock tree debugger in the GUI – Open the Clock Tree Debugger to see a tree of all clock gates, buffers, and sinks in your design. The y-scale is the clock insertion delay for that cell (i.e., how long it takes for the clock to propagate from the pin to that cell).
4.10 Route, Postroute, and Signoff

The final steps are less involved from a designer perspective. These steps run detailed route (i.e., the `cadence-innovus-route` step) and loop until timing is met or until the tool quits (i.e., the `cadence-innovus-postroute` step). Then the final timing analysis is run and output files are written out (i.e., the `cadence-innovus-signoff` step). You can run the design up to this step like this:

```
% cd $top/build
% make cadence-innovus-route
% make cadence-innovus-postroute
% make cadence-innovus-signoff
```

Here are the inputs, outputs, and scripts and what they do. You should not expect to change these.

<table>
<thead>
<tr>
<th>input</th>
<th>design.checkpoint</th>
<th>The working Innovus database from the previous step.</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>design.checkpoint</td>
<td>The working Innovus database after the step finishes.</td>
</tr>
<tr>
<td>output</td>
<td>design.gds.gz</td>
<td>The GDS layout with all wires and empty holes for the stdcells. The GDS merge step that comes next will combine this one with the stdcell.gds in the ADK (and any SRAM gds) to create the final GDS.</td>
</tr>
<tr>
<td>output</td>
<td>design.lvs.v, design.vcs.v, and design.virtuoso.v</td>
<td>These are slightly different versions of the post-place-and-route gate-level netlist for different purposes. The LVS version removes physical-only cells (e.g., fillers) that would cause spurious LVS failures. The Virtuoso version removes decaps that significantly slow down SPICE simulation.</td>
</tr>
<tr>
<td>output</td>
<td>design.lef</td>
<td>The LEF view of your final design. This can be used to insert your design into a larger design in another flow.</td>
</tr>
<tr>
<td>output</td>
<td>design.pt.sdc</td>
<td>The constraints that Innovus used, meant to tell the timing signoff engine in Synopsys PT what constraints to use.</td>
</tr>
<tr>
<td>output</td>
<td>design.spef.gz</td>
<td>This file has parasitic RC values for every net in your design. This is used in timing signoff and power estimation.</td>
</tr>
<tr>
<td>script</td>
<td>setup-optmode.tcl</td>
<td>This is where you might set up slack targets for setup/hold. If you are having trouble with small amounts of negative slack, you can set numbers here.</td>
</tr>
<tr>
<td>script</td>
<td>generate-results.tcl</td>
<td>This script writes out all the design files after place and route is over.</td>
</tr>
</tbody>
</table>

Here is a list of checks you will want to run through:

- `reports/signoff.area.rpt` – Compare this area report to the report from synthesis. This area will be higher because it includes not just the gates but also margins, empty space (filler), halos, buffers, and any overheads from floorplan restrictions.

- `logs/run.log` – Search for violations and make sure the violation count is zero. This is an automated Innovus DRC that is equivalent to clicking Verify DRC in the GUI. Also check the connectivity report in the log right next to the DRC. This is Innovus running LVS on the wires and making sure connections in the layout match those in the netlist.

- `reports/signoff.summary` – The numbers in this report should be similar to those in the place summary report. The timing engine that generated these numbers is signoff quality, so the exact numbers will be slightly different.

- `reports/signoff_hold.summary` – This is the hold timing report. This report must only have positive numbers!

- Pull up the GUI – This is a good chance to just look at your final layout and check if anything odd catches your eye. Feel free to do any of the checks from previous steps again.
4.11 Design Rule Check (DRC)

Here we run DRC with Mentor Calibre (i.e., the mentor-calibre-drc step). You can run the design up to this step like this:

```
% cd $top/build
% make mentor-calibre-drc
```

Here are the two inputs to this step:

<table>
<thead>
<tr>
<th>input</th>
<th>design_merged.gds</th>
<th>A merged GDS containing the Innovus GDS, the stdcell GDS, and any macro GDS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>adk</td>
<td>This step uses the Calibre DRC rule deck from the ADK.</td>
</tr>
</tbody>
</table>

You should see zero total results generated at the end of the log from this step:

```
--- CALIBRE::DRC-H EXECUTIVE MODULE COMPLETED. CPU TIME = 2 REAL TIME = 1
--- TOTAL RULECHECKS EXECUTED = 167
--- TOTAL RESULTS GENERATED = 0 (0)
--- DRC RESULTS DATABASE FILE = drc.results (ASCII)
```

If you see any results, open the debug target for this step to bring up the GDS viewer GUI (Calibre DESIGNrev) and inspect each violation. After the layout loads, press > many times to increase the depth and see inside all the cells. To show layout of macros press Shift + f.

The RVE window will also pop up showing all violations. In this case, DRC is clean for the GcdUnit. If there are errors, you can right click on the error and select Highlight, this will highlight the error in the DESIGNrev GUI. This is usually very zoomed in. You can change zoom from the View menu or using shortcuts (also shown in the View menu).

4.12 Layout-vs-Schematic (LVS)

Next we run LVS also with Mentor Calibre (i.e., the mentor-calibre-lvs step). You can run the design up to this step like this:

```
% cd $top/build
% make mentor-calibre-lvs
```
Here are the inputs, outputs, and scripts and what they do.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rules.sv</td>
<td>An optional file with additional LVS commands to run.</td>
</tr>
<tr>
<td>design_merged.gds</td>
<td>This must be the same merged GDS that you ran DRC on. The combination of DRC and LVS tells you that the layout is manufacturable (DRC) and that that same layout matches the schematic (LVS) that you run gate-level tests on. If you run DRC and LVS on different layouts, then this semantic link is broken.</td>
</tr>
<tr>
<td>design.lvs.v</td>
<td>The post-place-and-route gate-level netlist with physical-only cells removed.</td>
</tr>
<tr>
<td>adk</td>
<td>This step uses the Calibre LVS rule deck from the ADK.</td>
</tr>
</tbody>
</table>

Open the LVS report to check the comparison result:

- `lvs.report` – This will have a smiley face at the top if you pass LVS. The most important table to look at in the lvs.report is the “Information and Warnings” table part of the way down. This will show the number of ports, nets, and devices found in the layout and in the source and how the numbers match up.

![INFORMATION AND WARNINGS](image)

If you see a mismatch, open the debug target for this step to bring up the GDS viewer GUI (Calibre DESIGNrev). After the layout loads, press $>$ many times to increase the depth and see inside all the cells.

Unfortunately, LVS mismatches are very difficult to debug because of how LVS transforms and squashes transistors together, causing a mismatch in one place to manifest in another. The debug GUI will still be useful to list which nets and ports Calibre thinks are missing.
### 4.13 List of mflowgen Common Library Nodes

<table>
<thead>
<tr>
<th>Base Tool</th>
<th>#</th>
<th>Description of Modular Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence Genus</td>
<td>1</td>
<td>Synthesis</td>
</tr>
<tr>
<td>Cadence Innovus</td>
<td>14</td>
<td>Init, place, cts, route, postroute, signoff, post-pnr ecos, foundation flow setup, hold-fixing nodes, power grid setup</td>
</tr>
<tr>
<td>Cadence Pegasus</td>
<td>4</td>
<td>DRC, LVS, GDS merging, metal fill</td>
</tr>
<tr>
<td>Synopsys DC</td>
<td>1</td>
<td>Synthesis</td>
</tr>
<tr>
<td>Synopsys Formal-</td>
<td>1</td>
<td>Logical equivalence check</td>
</tr>
<tr>
<td>ity Synopsys PT(PX)</td>
<td>6</td>
<td>Timing/power signoff, ECOs, gen lib/db, RTL- and gate-level power estimation</td>
</tr>
<tr>
<td>Synopsys VCS</td>
<td>2</td>
<td>RTL- and gate-level simulation, vcd2saif</td>
</tr>
<tr>
<td>Mentore Calibre</td>
<td>7</td>
<td>DRC, LVS, GDS merging, metal fill, convert verilog2spice, drawing chip art</td>
</tr>
<tr>
<td>Open-Source</td>
<td>8</td>
<td>Synthesis (yosys), place (graywolf), place (RePlAcE), route (qrouter) LVS (netgen), DRC (magic) gds2spice and def2spice (magic)</td>
</tr>
<tr>
<td>Total # of Nodes</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Open-Source Technologies</td>
<td>2</td>
<td>SkyWater 130nm, FreePDK45 with NanGate Open Cell Library</td>
</tr>
</tbody>
</table>
5.1 My builds are continuing on to the next step despite errors in this step?

Builds will stop for errors if the shell exit status (i.e., $?) is detected to be non-zero. This is standard shell behavior and a Makefile will automatically stop if any command fails.

You can check the exit status like this:

```
% bash -c "exit 0"
% echo $?
0

% bash -c "exit 1"
% echo $?
1

% bash -c "non_existent_command"
bash: non_existent_command: command not found
% echo $?
127

% bash -c "echo abc"
abc
% echo $?
0
```

If you add the command `exit 1` to a step (i.e., exiting with an error) then the entire build will error out and will certainly not continue to the next steps. For example, say we added this to the synthesis step:

```bash
name: synopsys-dc-synthesis
(...)
```

(continues on next page)
There would be an error and the build would not continue:

```
% make 4
make: *** [4-synopsys-dc-synthesis/.execstamp] Error 1
```

Remember you can check `make status` to see whether a step will build or not. By adding `exit 1`, the synthesis status will never be "done" (in green). It will always show "build" (in red):

```
% make status
(...)  
- build -> 4 : synopsys-dc-synthesis
```

Most issues arise when you are calling a script which has errors but does not propagate the exit code to the caller. For example, say the synthesis step runs a script called `run.sh`:

```
name: synopsys-dc-synthesis

(...)  
commands:
- bash run.sh  # <-- errors inside but does not propagate exit code
```

The script has the following contents:

```
% non_existent_command
% echo "hi"  # <-- valid with no errors... masks exit status of script
```

Upon running this step, you will see the error but the script will continue on to print “hi” and execute future steps:

```
% make 4
run.sh: line 1: non_existent_command: command not found
hi
```

You can instead have the script exit with an error exit status like this:

```
% non_existent_command || exit 1
% echo "hi"  # we never get here because we exit after the error
```

You will see the error, but the following commands will not run, and the build will also stop:

```
% make 4
run.sh: line 1: non_existent_command: command not found
make: *** [4-synopsys-dc-synthesis/.execstamp] Error 1
```

There are many ways to propagate exit status properly in a shell script. We recommend explicitly controlling exits for errors on the commands you know are sensitive as shown above. Because mflowgen run scripts have error checking flags enabled, we recommend sourcing scripts instead of calling them in a subshell:

```
name: synopsys-dc-synthesis

(...)  
```

(continues on next page)
commands:
- bash run.sh # <-- not recommended.. you must propagate error
          # exit status flags on your own
- source run.sh # <-- we recommend to source in the existing shell

Related options for bash are available to exit on non-zero exit status (set -e:) and for enabling pipefail (set -o pipefail).
Python Module Index

- **e**
  Edge, 10

- **m**
  mflowgen.components, 7

- **s**
  Step, 9
Index

A
add_step() (mflowgen.components.Graph method), 7
all_input_handles() (in module Step), 9
all_inputs() (in module Step), 9
all_output_handles() (in module Step), 9
all_outputs() (in module Step), 9
all_outputs_execute() (in module Step), 10
all_outputs_tagged() (in module Step), 10
all_outputs_untagged() (in module Step), 10
all_steps() (mflowgen.components.Graph method), 8

C
clone() (in module Step), 9

do connect() (mflowgen.components.Graph method), 8
connect_by_name() (mflowgen.components.Graph method), 8

D
dump_yaml() (in module Step), 10

E
Edge (module), 10
escape_dollars() (in module Step), 9
expand_params() (in module Step), 9
extend_inputs() (in module Step), 9
extend_outputs() (in module Step), 9
extend_postconditions() (in module Step), 9
extend_preconditions() (in module Step), 9

G
get_adk_step() (mflowgen.components.Graph method), 7
get_commands() (in module Step), 10
get_debug_commands() (in module Step), 10
get_dir() (in module Step), 10
get_dst() (in module Edge), 10
get_input_handle() (in module Step), 9
get_name() (in module Step), 9
get_output_handle() (in module Step), 9
get_param() (in module Step), 9
get_sandbox() (in module Step), 10
get_src() (in module Edge), 10
get_step() (mflowgen.components.Graph method), 8
Graph (class in mflowgen.components), 7

I
i() (in module Step), 9

M
mflowgen.components (module), 7

O
o() (in module Step), 9

P
param_space() (mflowgen.components.Graph method), 8
params() (in module Step), 9
preExtend_commands() (in module Step), 9

S
set_adk() (mflowgen.components.Graph method), 7
set_name() (in module Step), 9
set_param() (in module Step), 9
set_sandbox() (in module Step), 10
Step (module), 9

U
update_params() (in module Step), 9
update_params() (mflowgen.components.Graph method), 8