Writing Custom Flows

As configurable as the default ("Classic") flow may be, there are some designs that would simply be too complex to implement using the existing flow.

For example, hardening a macro + padframe for a top level design is too complex using the Classic flow, and may require you to write your own custom-based flow.

First of all, please review OpenLane's high-level architecture at this link.

This defines many of the terms used and enumerates strictures mentioned in this document.

Custom Sequential Flows

In Configuration Files

By Substituting Steps

If you're constructing a flow that is largely based on another flow, albeit with some substitutions or removals, you may declare your base flow and substitutions as follows:

```
{
  "meta": {
    "version": 2,
    "flow": "Classic",
    "substituting_steps": {
      "OpenROAD.STAMidPNR*": null,
      "Magic.DRC": "KLayout.DRC",
    }
  }
}
```

This replaces `Magic.DRC` with *another* `KLayout.DRC` step (which is useless but this is just for demonstration); and removes all steps starting with `OpenROAD.STAMidPNR` from the `Classic` flow.

Instead of replacing, you can also emplace steps before or after steps. Simply put a `-` before the target step ID to place before, and a `+` to place after.

Substitutions are more useful if you have custom steps registered in OpenLane Plugins.

By Listing Steps

If your custom sequential flow entirely relies on built-in steps, you can actually specify a flow entirely in the `config.json` file, with no API access needed:

```
{
  "meta": {
    "version": 2,
    "flow": [
      "Yosys.Synthesis",
      "OpenROAD.CheckSDCFiles",
      "OpenROAD.Floorplan",
      "OpenROAD.TapEndcapInsertion",
      "OpenROAD.GeneratePDN",
      "OpenROAD.IOPlacement",
      "OpenROAD.GlobalPlacement",
      "OpenROAD.DetailedPlacement",
      "OpenROAD.GlobalRouting",
      "OpenROAD.DetailedRouting",
      "OpenROAD.FillInsertion",
      "Magic.StreamOut",
      "Magic.DRC",
      "Magic.SpiceExtraction",
      "Netgen.LVS"
    ]
  }
}
```

Using the API

By Listing Steps

You'll need to import the `Flow` class as well as any steps you intend to use.

An equivalent flow to the one in By Listing Steps would look something like this:

```
from openlane.flows import SequentialFlow
from openlane.steps import Yosys, Misc, OpenROAD, Magic, Netgen

class MyFlow(SequentialFlow):
    Steps = [
        Yosys.Synthesis,
        OpenROAD.CheckSDCFiles,
        OpenROAD.Floorplan,
        OpenROAD.TapEndcapInsertion,
        OpenROAD.GeneratePDN,
        OpenROAD.IOPlacement,
        OpenROAD.GlobalPlacement,
        OpenROAD.DetailedPlacement,
        OpenROAD.GlobalRouting,
        OpenROAD.DetailedRouting,
        OpenROAD.FillInsertion,
        Magic.StreamOut,
        Magic.DRC,
        Magic.SpiceExtraction,
        Netgen.LVS
    ]
```

You may then instantiate and start the flow as shown:

```
flow = MyFlow(
    {
        "PDK": "sky130A",
        "DESIGN_NAME": "spm",
        "VERILOG_FILES": ["./src/spm.v"],
        "CLOCK_PORT": "clk",
        "CLOCK_PERIOD": 10,
    },
    design_dir=".",
)
flow.start()
```

The `openlane.flows.Flow.start()` method will return a tuple comprised of:

- The final output state ($State_n$).
- A list of all step objects created during the running of this flow object.

> **Important**
>
> Do NOT call the `run` method of any `Flow` from outside of `Flow` and its subclasses- consider it a protected method. `start` is class-independent and does some incredibly important processing.
>
> You should not be overriding `start` either.

Fully Customized Flows

Each `Flow` subclass must:

- Declare the steps used in the `Steps` attribute.
  - The steps are examined so their configuration variables can be validated ahead of time.
- Implement the `openlane.flows.Flow.run()` method.
  - This step is responsible for the core logic of the flow, i.e., instantiating steps and calling them.
  - This method must return the final state and a list of step objects created.

You may notice you are allowed to do pretty much anything inside the `run` method. While that may indeed enable you to perform arbitrary logic in a Flow, it is recommended that you write Steps, keeping the logic in the Flow to a minimum.

You may instantiate and use steps inside flows as follows:

```
synthesis = Yosys.Synthesis(
    config=self.config,
    state_in=...,
)
synthesis.start()

sdc_load = OpenROAD.CheckSDCFiles(
    config=self.config,
    state_in=synthesis.state_out,
)
```

While you may not modify the configuration object (in `self.config`), you can slightly modify the configuration used by each step using the config object's `openlane.config.Config.copy()` method, which allows you to supply overrides as follows:

```
config_altered = config.copy(FP_CORE_UTIL=9)
```

Which will create a new configuration object with one or more attributes modified. You can pass these to steps as you desire.

Another advantage of this over sequential flows is that you can handle Step failures more elegantly, i.e., by trying something else when a particular Step (or set of steps) fail. There are a lot of possibilities.

Reporting Progress

Correctly-written steps will by default output a log to the terminal, but, running in a flow, there will always be a progress bar at the bottom of the terminal:

```
Classic - Stage 17 - CTS ━━━━━━━━━━━━━━━━━━ ━━━━━━━━━━━━━━━━━━━━━━ 16/37 0:00:2
```

The Flow object has methods to manage this progress bar:

- `openlane.flows.Flow.progress_bar.set_max_stage_count()`
- `openlane.flows.Flow.progress_bar.start_stage()`
- `openlane.flows.Flow.progress_bar.end_stage()`.

They are to be called from inside the `run` method. In Sequential Flows, $|Steps| = |Stages| = n$, but in custom flows, stages can incorporate any number of steps. This is useful for example when running series of steps in parallel as shown in the next section, where incrementing by step is not exactly viable.

Multi-Threading

> **Important**
>
> The `Flow` object is NOT thread-safe. If you're going to run one or more steps in parallel, please follow this guide on how to do so.

The `Flow` object offers a method to run steps asynchronously, `openlane.flows.Flow.start_step_async()`. This method returns a `Future` encapsulating a State object, which can then be used as an input to future Steps.

This approach creates a dependency chain between steps, so if you attempt to inspect the last Future from a set of asynchronous steps, it will automatically run

the required steps, in parallel if need be.

Here is a demo flow built on exactly this principle. It works across two stages:

- The Synthesis Exploration - tries multiple synthesis strategies in *parallel*. The best-performing strategy in terms of minimizing the area makes it to the next stage.
- Floorplanning and Placement - tries FP and placement with a high utilization.
  - If the high utilization fails, a lower is fallen back to as a suggestion.

```
class Optimizing(Flow):
    Steps = [
        Yosys.Synthesis,
        OpenROAD.CheckSDCFiles,
        OpenROAD.STAPrePNR,
        OpenROAD.Floorplan,
        OpenROAD.IOPlacement,
        OpenROAD.GlobalPlacement,
    ]

    def run(
        self,
        initial_state: State,
        **kwargs,
    ) -> Tuple[State, List[Step]]:
        step_list: List[Step] = []

        self.set_max_stage_count(2)

        synthesis_futures: List[Tuple[Config, Future[State]]] = []
        self.start_stage("Synthesis Exploration")

        log_level_bk = get_log_level()
        set_log_level(LogLevels.ERROR)

        for strategy in ["AREA 0", "AREA 2", "DELAY 1"]:
            config = self.config.copy(SYNTH_STRATEGY=strategy)

            synth_step = Yosys.Synthesis(
                config,
                id=f"synthesis-{strategy}",
                state_in=initial_state,
                flow=self,
            )
            synth_future = self.start_step_async(synth_step)
            step_list.append(synth_step)

            sdc_step = OpenROAD.CheckSDCFiles(
                config,
                id=f"sdc-{strategy}",
                state_in=synth_future,
                flow=self,
            )
            sdc_future = self.start_step_async(sdc_step)
            step_list.append(sdc_step)

            sta_step = OpenROAD.STAPrePNR(
```

```
            config,
            state_in=sdc_future,
            id=f"sta-{strategy}",
            flow=self,
        )

        step_list.append(sta_step)
        sta_future = self.start_step_async(sta_step)

        synthesis_futures.append((config, sta_future))

    synthesis_states: List[Tuple[Config, State]] = [
        (config, future.result()) for config, future in synthesis_futures
    ]

    self.end_stage()
    set_log_level(log_level_bk)

    min_strat = synthesis_states[0][0]["SYNTH_STRATEGY"]
    min_config = synthesis_states[0][0]
    min_area_state = synthesis_states[0][1]
    for config, state in synthesis_states[1:]:
        strategy = config["SYNTH_STRATEGY"]
        if (
            state.metrics["design__instance__area"]
            < min_area_state.metrics["design__instance__area"]
        ):
            min_area_state = state
            min_strat = strategy
            min_config = config

    info(f"Using result from '{min_strat}…")

    self.start_stage("Floorplanning and Placement")

    fp_config = min_config.copy(FP_CORE_UTIL=99)
    fp = OpenROAD.Floorplan(
        fp_config,
        state_in=min_area_state,
        id="fp_highutl",
        long_name="Floorplanning (High Util)",
        flow=self,
    )
    self.start_step(fp)
    step_list.append(fp)
    try:
        io = OpenROAD.IOPlacement(
            fp_config,
```

```
                state_in=fp.state_out,
                id="io-highutl",
                long_name="I/O Placement (High Util)",
                flow=self,
            )
            self.start_step(io)
            step_list.append(io)
            gpl = OpenROAD.GlobalPlacement(
                fp_config,
                state_in=io.state_out,
                id="gpl-highutil",
                long_name="Global Placement (High Util)",
                flow=self,
            )
            self.start_step(gpl)
            step_list.append(gpl)
        except StepError:
            info("High utilization failed- attempting low utilization…")
            fp_config = min_config.copy(FP_CORE_UTIL=40)
            fp = OpenROAD.Floorplan(
                fp_config,
                state_in=min_area_state,
                id="fp-lowutl",
                long_name="Floorplanning (Low Util)",
                flow=self,
            )
            self.start_step(fp)
            step_list.append(fp)
            io = OpenROAD.IOPlacement(
                fp_config,
                state_in=fp.state_out,
                id="io-lowutl",
                long_name="I/O Placement (Low Util)",
                flow=self,
            )
            self.start_step(io)
            step_list.append(io)
            gpl = OpenROAD.GlobalPlacement(
                fp_config,
                state_in=io.state_out,
                id="gpl-lowutl",
                long_name="Global Placement (Low Util)",
                flow=self,
            )
            self.start_step(gpl)
            step_list.append(gpl)

        self.end_stage()
```

```
        success("Flow complete.")
        assert gpl.state_out is not None  # We should be done with the executi
        return (gpl.state_out, step_list)
```

Error Throwing and Handling

Steps may throw one of these hierarchy of errors, namely:

- `openlane.steps.StepError`: For when there is an error in running one of the tools or the input data.
  - `openlane.steps.DeferredStepError`: A `StepError` that suggests that the Flow continue anyway and only report this error when the flow finishes. This is useful for errors that are not "show-stoppers," i.e. a timing violation for example.
  - `openlane.steps.StepException`: A `StepError` when there is a higher-level step failure, such as the step object itself generating an invalid state or a state input to a `Step` has missing inputs.

As a rule of thumb, it is sufficient to forward these errors as one of these two:

- `openlane.flows.FlowError`
  - `openlane.flows.FlowException`

Which share a similar hierarchy. Here is how `SequentialFlow`, for example, handles its `StepError`s:

```
                try:
                    current_state = step.start(
                        toolbox=self.toolbox,
                        step_dir=self.dir_for_step(step),
                    )
                except StepException as e:
                    raise FlowException(str(e)) from None
                except DeferredStepError as e:
                    deferred_errors.append(str(e))
                except StepError as e:
                    raise FlowError(str(e)) from None
```

As you may see, the deferred errors are saved for later, but the other two are forwarded pretty much-as is. If no other errors are encountered, the deferred errors are logged then reported as a `StepException`.

Copyright © 2020-2023 Efabless Corporation and contributors
Made with Sphinx and @pradyunsg's Furo