Automated Test Generation to Verify IP Modified for System Level Power Management

Christophe Lamard, STMicroelectronics

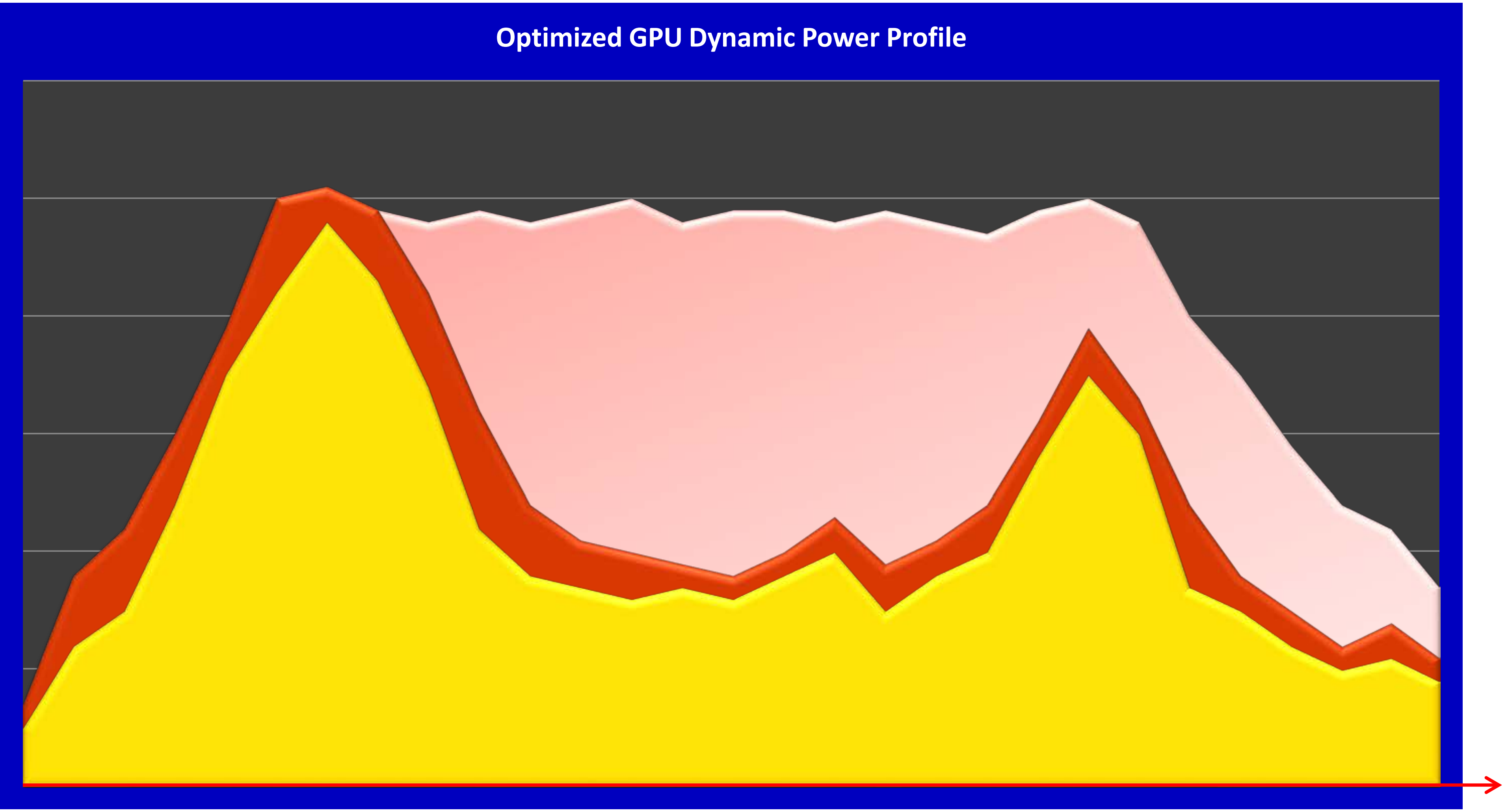
Frederic Dupuis, Cadence

GPU Power optimization

- We integrate internal and third party GPU IP
  - Replace generic Macro Block according to the technology
  - Tune power capability
  - Split hierarchy design according to layout team request
  - Adjust DFT structure

GPU power optimization

Optimized GPU Dynamic Power Profile

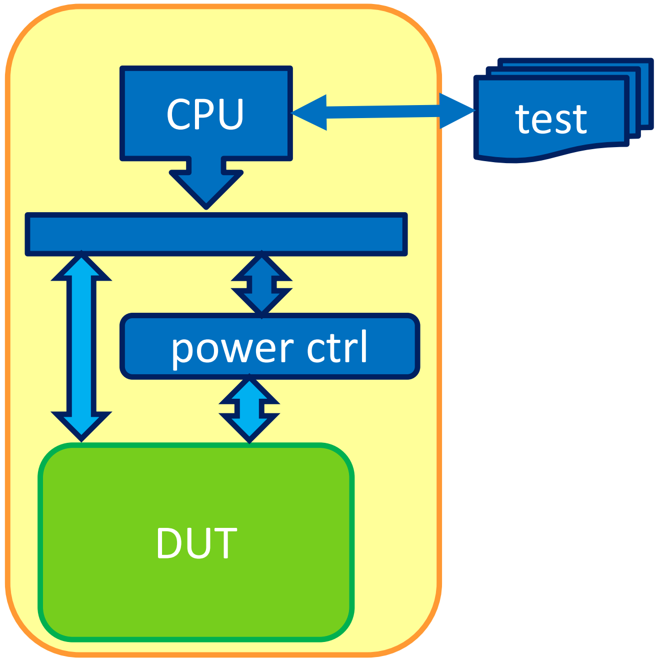


Dynamic Power verification

- New bug types comes with power management
  - Missing Isolation bugs
  - Control Sequencing bugs
  - Retention scheme errors
  - Memory corruption
  - Power sequence scheduling errors
  - Software/Hardware dead lock
  - Power On Reset bugs
  - ….
- Verification team need to manage dynamic power simulation:
  - Create Power test sequence
  - Run dynamic power simulation
  - Add power checker

Sub-system Integration

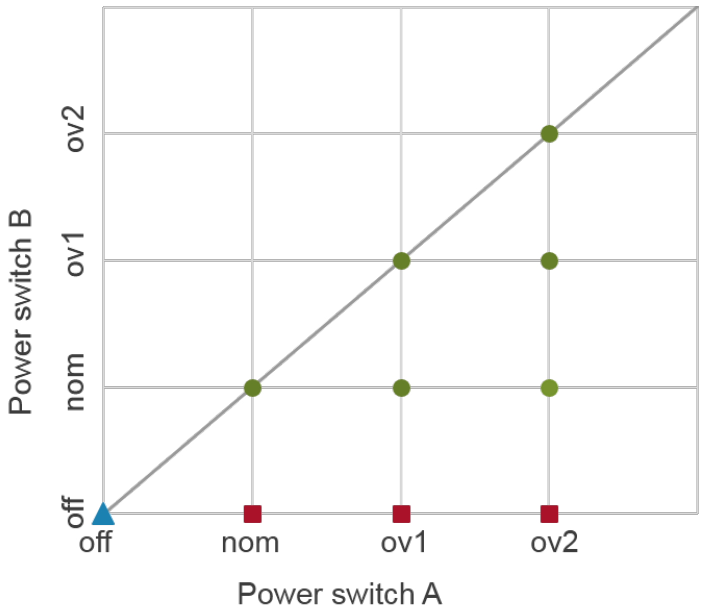
- We deliver our test to SOC, Validation and SW team
  - need for portable tests (not UVM)
  - Tests are written in ‘C’
- We have created :
  - A test bench API to control:
    - IP Powers state
    - IP top signal like clamp, reset, clock
  - An IP API to control IP power sequence
    - Retention sequence
    - Clock, reset and clamp control
  - Some Power monitor/checker



Closer look at tests requirements

- Let s consider 2 power switch A and B.
  - 3 different possible values: nom, overdrive1, overdrive2
  - They are linked by a rule :

value power switch B <= value power switch A



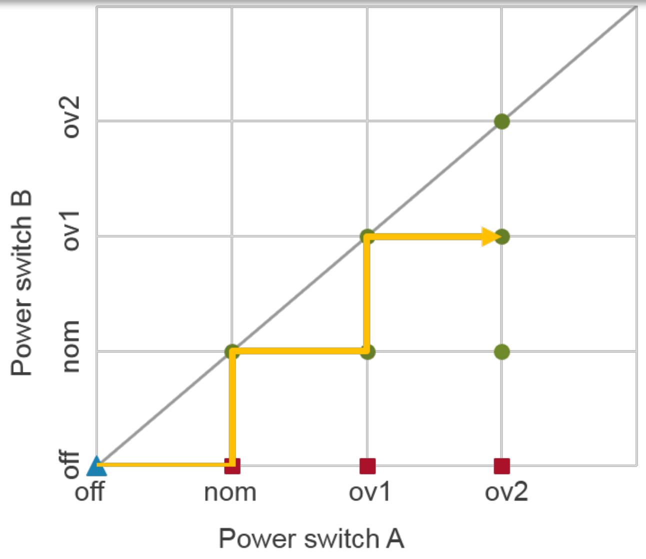
Blue triangle = initial state
Red Square = transition state
Green points = functional state

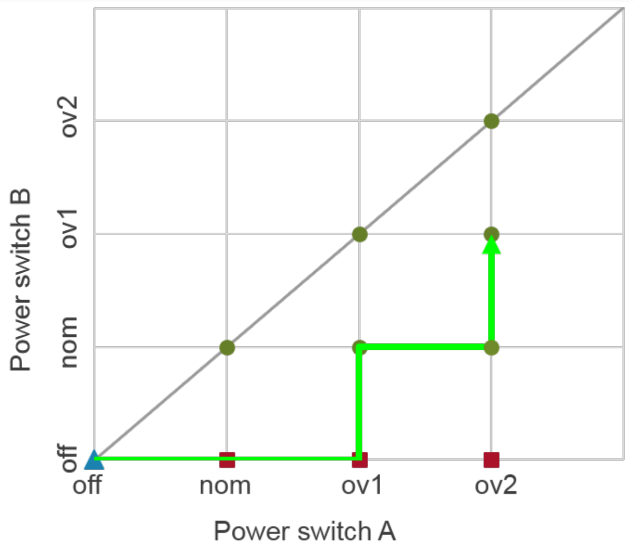
Gray line = rule
Transitions below the line are allowed.

Defining one test

- A test is always :
  1) Put the power logic in a functional state
  2) Run a functional tests on the IP

Functional state : Power switch A = OV2 and Power Switch B = OV1
User need to define transition from initial state to this functional state



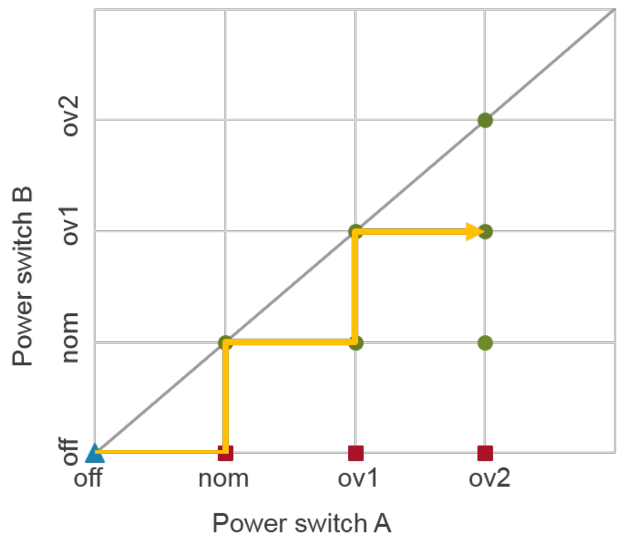


Defining one test (cont)

- **Additional power elements** must be configured to be in a functional state :
  - **Clock** for power domain A and B
  - **Reset and Clamp** for A and B
- **Different rules** also exist for these elements :
  - Can be configured when power switch is not off
  - Some clock frequency cannot be used with some power switch value
- Some more complex tests scenario are needed :
  - Go in a functional state,
  - Switch off the power
  - Go in a different functional state

Manually developed C code

- Here is one C code example to check this functional state :

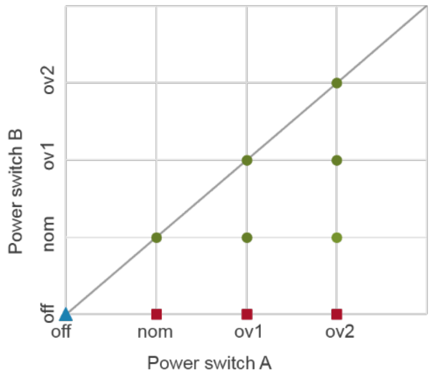


```
turn_on_power_switchA();
turn_on_power_switchB();
setup_clamp_reset_powerB();
setup_clock_powerB(freq0)
setup_clamp_reset_powerA();
setup_clock_powerA(freq1) ;
change_power_switchA(ov1);
change_power_switchB(ov1);
change_power_switchA(ov2);
run_functional_test();
```

- Obviously there are much more possible tests :
  - Change the path (different transitions)
  - Change clock frequency, change setup time of clamp/reset

Need for automation

- On this simple example we would need at least 6 tests (for each functional state)
- In our GPU design we have :
  - **More power switch** (up to 6)
  - **More complex rules**
  - **Specification may change** during project



- Developing a test requires **deep knowledge** of power spec
- Not possible to **create and maintain** all needed tests.
- We have developed **20** tests (targeting 20 states)
  - Most of the time same path is used (extension of previous test)
  - Other power elements often configured same time
- **Need for automation** to create tests for all possible state

Defining new methodology using automation tool

- Main contribution of the automation tool are
  - **Model based approach**

    simple and abstract way to define and constrain the power elements
  - **Goal directed test creation**

    Thanks to the use case based solver (describe what, not how). It means describing expected power state, not the transitions to reach it.
  - **Automated test generation**

    Simple way to achieve 100% coverage of specified goals. Goals here would be the complete list of functional state

Model based approach

- Do not describe path **from initial state to functional state**
- But:
  - Describe **all power elements** and their possible values
  - Define all possible **transitions** and their **relations** with other power elements
  - Map to each transition the **associated API** call
- Force to have a systematic description and completely understand the power specification

Describe all power elements

- If we use power switch B of our previous example
  - Possible values: off, nominal , overdrive1, overdrive2
  - List of transition, associated rules and API

Transition	Rules	Associated API
off_to_nom	pswitchA != off	turn_on_power_switchB()
nom_to_off	pswitchA != off (*)	turn_off_power_switchB()
nom_to_ov1	pswitchA > nom	change_power_switchB(ov1);
ov1_to_nom	pswitchA > nom (*)	change_power_switchB(nom);
ov1_to_ov2	pswitchA == ov2	change_power_switchB(ov2);
ov2_to_ov1	pswitchA == ov2 (*)	change_power_switchB(ov1);

(*) implicit rules (already put for power switch A) but added for clarity and debug purposes

Goal directed test creation

- User can define a test in a **goal directed** way.
- Define **what** (the expected functional state) and not **how** (the path to reach it)

Power switch B == OV1 and Power switch A == OV2

- Tool will **automatically find a path** from initial state to this state.
- Each generation may create a different path.
- It is not possible to create a test that contradicts the rules

Power switch B == OV2 and Power switch A == OV1 (illegal state)

  – Tool will report an error.
  – No time spent on trying to run/debug a wrong test

Automated test generation

- User can also request tests in **all possible** functional state :

Power switch B == “all value” and Power switch A == “all value”

- Only legal tests will be created (following the rules)
  - No need to know all of them, the tool with find them
  - Tool will also report the non valid case, useful for debug
- In our case, tool has been able to create **192** tests, all reaching a **different functional state**.
  - Different path have been used and the different power elements have been configured at different point of time
  - Coverage of all possible path might be possible too but this was not our main requirement

Pros/Cons of the methods

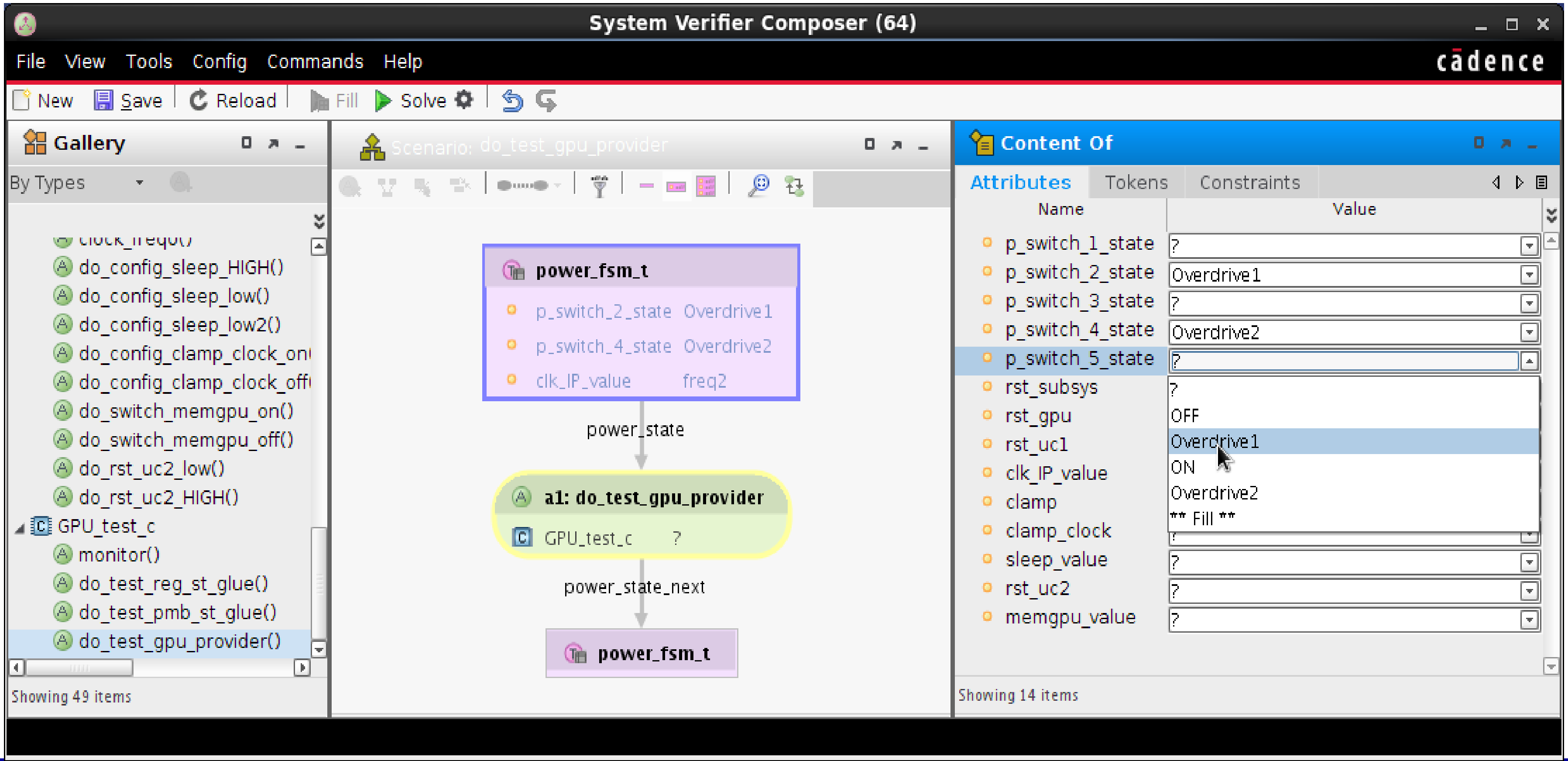
- Pros :
  - Enable to **divide power specification** and focus piece by piece
  - Create a kind of **embedded power specification**
    - Usable and readable by anyone
  - **Changes** to power specification could easily be reported
    - Add/remove a transition, add/remove/change rules
  - Goal directed test creation is very efficient (develop a new test in seconds)
- Cons :
  - Extra work for first tests.
  - Need to develop a new model and learn a new language
  - Model has to be exhaustive

Choice of automation tool : Perspec System Verifier

- We developed and used this methodology using **Perspec System Verifier**:
  - **Model based** approach of the tool
  - **Graphical representation** of the generated test
    - UML activity diagram showing all transition from initial to final state
  - Generated C test is linear and readable
    - Simplify debug
- First model was developed with local AE support
- No need for deep knowledge of the language
  - Learning curve is in days
  - Not the complexity of UVM for example

Goal directed test creation

- User can ask the tool to create a test in a specific functional state :


System Verifier Composer (64)
File View Tools Config Commands Help
cadence
New Save Reload Fill Solve
Gallery
By Types
Scenario: do_test_gpu_provider
Content Of
Attributes Tokens Constraints
power_state
power_state_next
Showing 49 items
Showing 14 items

Gallery:
- clock_freq()
- do_config_sleep_HIGH()
- do_config_sleep_low()
- do_config_sleep_low2()
- do_config_clamp_clock_on(
- do_config_clamp_clock_off(
- do_switch_memgpu_on()
- do_switch_memgpu_off()
- do_rst_uc2_low()
- do_rst_uc2_HIGH()
- GPU_test_c
  - monitor()
  - do_test_reg_st_glue()
  - do_test_pmb_st_glue()
  - do_test_gpu_provider()

Scenario:

power_fsm_t
- p_switch_2_state Overdrive1
- p_switch_4_state Overdrive2
- clk_IP_value freq2

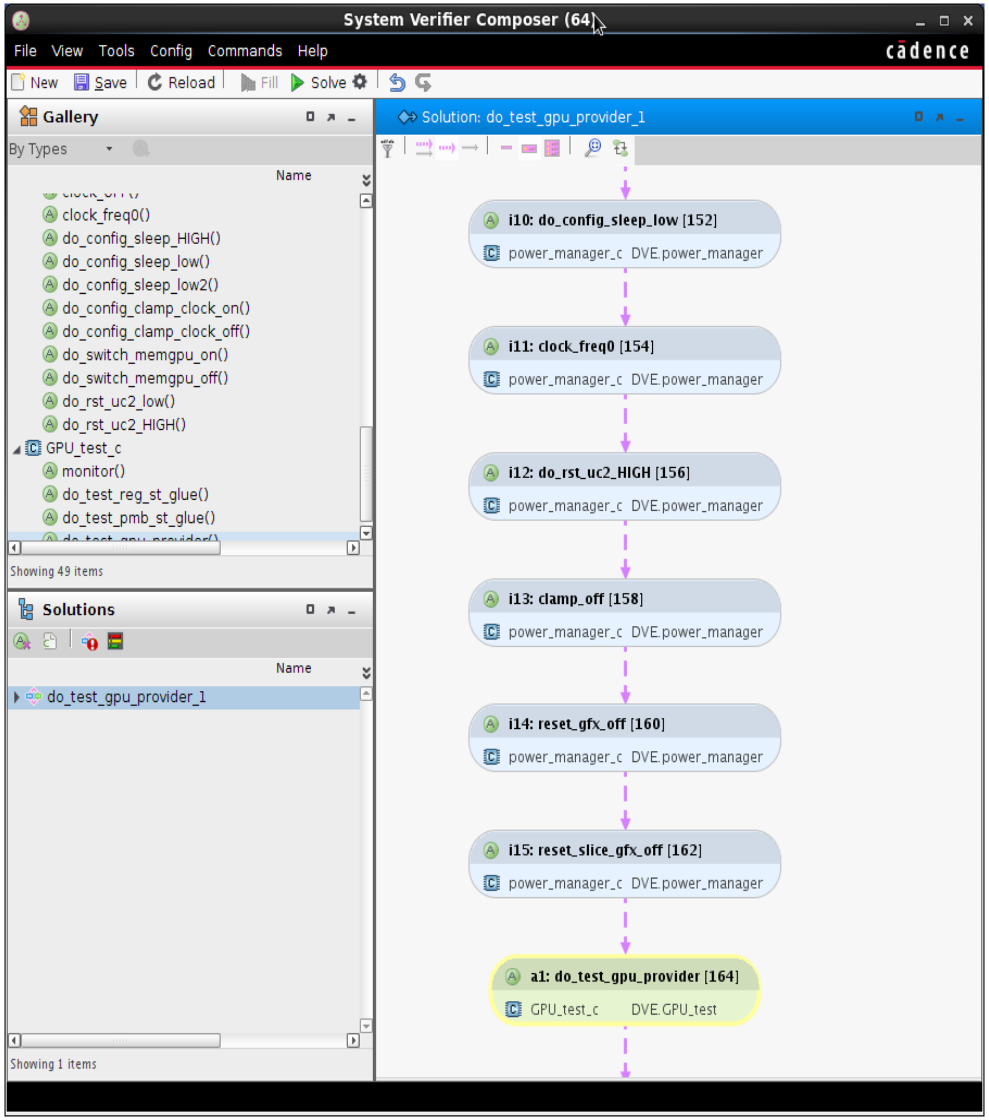
a1: do_test_gpu_provider
- GPU_test_c ?

power_fsm_t

Name	Value
p_switch_1_state	?
p_switch_2_state	Overdrive1
p_switch_3_state	?
p_switch_4_state	Overdrive2
p_switch_5_state	?
rst_subsys	
rst_gpu	
rst_uc1	
clk_IP_value	
clamp	
clamp_clock	?
sleep_value	?
rst_uc2	?
memgpu_value	?

?, OFF, Overdrive1, ON, Overdrive2, ** Fill **

Goal directed test creation


System Verifier Composer (64)
File View Tools Config Commands Help
cadence
New Save Reload Fill Solve
Gallery
By Types
Name
Solutions
Name
Solution: do_test_gpu_provider_1

clock_freq0()
do_config_sleep_HIGH()
do_config_sleep_low()
do_config_sleep_low2()
do_config_clamp_clock_on()
do_config_clamp_clock_off()
do_switch_memgpu_on()
do_switch_memgpu_off()
do_rst_uc2_low()
do_rst_uc2_HIGH()
GPU_test_c
monitor()
do_test_reg_st_glue()
do_test_pmb_st_glue()

Showing 49 items

do_test_gpu_provider_1

Showing 1 items

i10: do_config_sleep_low [152] — power_manager_c DVE.power_manager
i11: clock_freq0 [154] — power_manager_c DVE.power_manager
i12: do_rst_uc2_HIGH [156] — power_manager_c DVE.power_manager
i13: clamp_off [158] — power_manager_c DVE.power_manager
i14: reset_gfx_off [160] — power_manager_c DVE.power_manager
i15: reset_slice_gfx_off [162] — power_manager_c DVE.power_manager
a1: do_test_gpu_provider [164] — GPU_test_c DVE.GPU_test

Scenario is partially represented

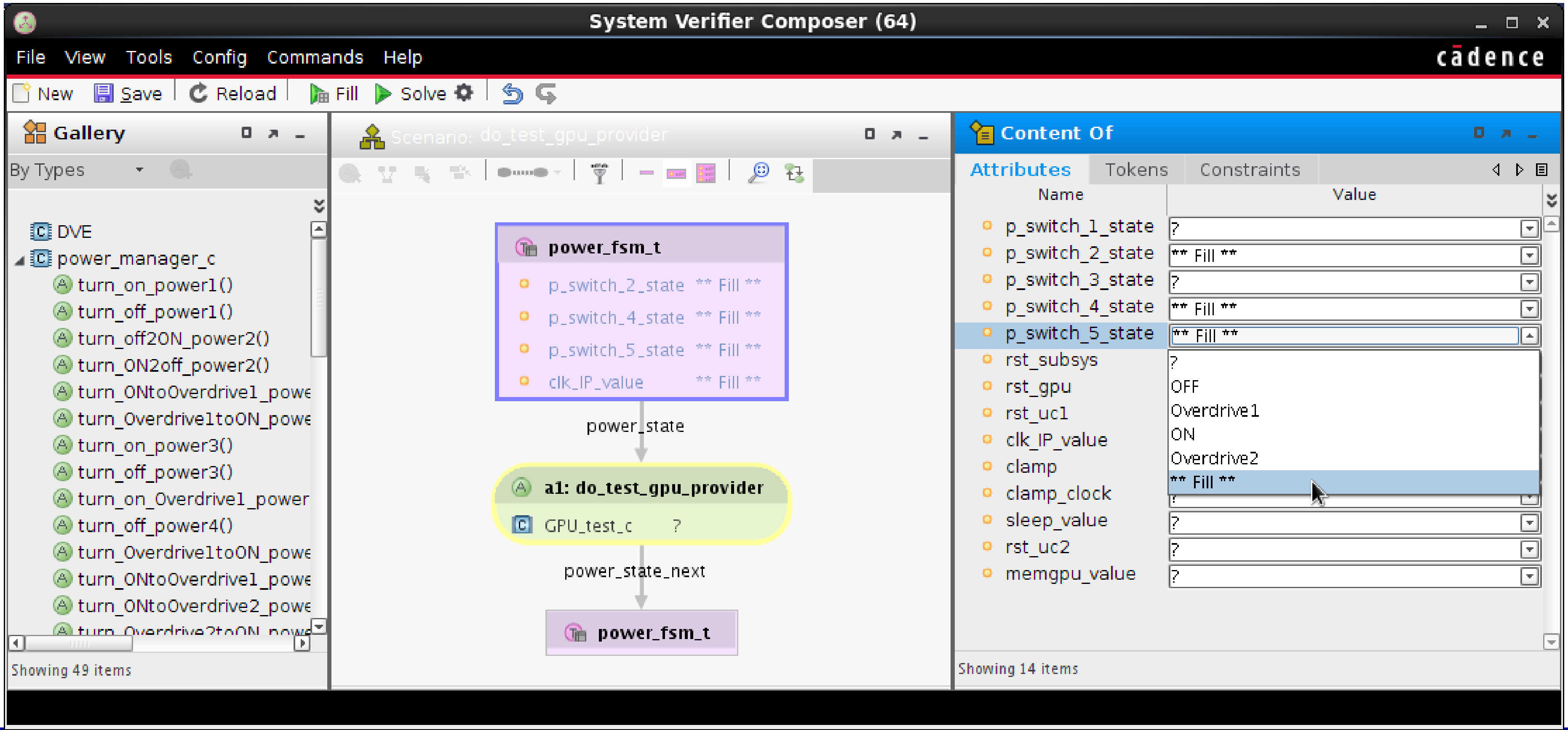
Tool identified a path, and created a scenario with all the transitions from initial state up to final state

The graphical representation also enables:

- **Quick analysis of the solution, and identification of model bugs**
- **Exchange /discuss with other stakeholders like design and architects.**
- **Simplify debug analysis of failing tests**

Automatic test creation

- User can ask the tool to create a test in all specific functional state : 192 tests in our case



Name	Value
p_switch_1_state	?
p_switch_2_state	** Fill **
p_switch_3_state	?
p_switch_4_state	** Fill **
p_switch_5_state	** Fill **
rst_subsys	?
rst_gpu	
rst_uc1	
clk_IP_value	
clamp	
clamp_clock	
sleep_value	?
rst_uc2	?
memgpu_value	?

? / OFF / Overdrive1 / ON / Overdrive2 / ** Fill **

Results

- **Higher** coverage in **less time** than manual tests development
  - All 192 generated tests are different and cover all states
  - Covering transition we did not think off
  - Estimated manual effort to reach same coverage: 192 days

	Nb tests	Lines of code	Develop ment	Maintenance (each change)	Nb tests /Day	Nb test/day in case of 5 changes
Manual	20	2k (100x20)	20 days	3 to 4 days	1	0.57
Perspec	192	800	10 days	1 day	19.2	12.8
Ratio		**0.4**			**19.2**	**22.4**

Conclusion and future works

- Coverage confirmed by PST coverage during simulation
- Identified bugs in :
  - Embedded design checkers
  - UPF file (missing/unreachable states)
- Increase confidence in power sequence supports
- Methodology could be applied to any LP verification

- **Future works :**
  - **Reuse** model at SOC level to create system level LP tests
  - Deploy on even more complex IPs
  - Model based enable to **combine LP and functional tests**