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

Christophe Lamard
STMicroelectronics
Frederic Dupuis
Cadence Design Systems

Abstract- **Verification of IP modifications to support complex system level power management is a must, but it is typically left to verification engineers to write the software to change power programming while testing to ensure continued operation of the IP. This paper looks at how automation of test generation can radically change not only the time it takes to create complex tests, but also the efficiency of testing needed to achieve maximum coverage.**

INTRODUCTION

Today's SoCs require sophisticated power management capabilities to meet the myriad of power requirements for different use cases. Developers often have to modify purchased IP to support complex system level power management capabilities developed for the SoC, typically the verification of the added functionality is best done using bare metal software that can test the IP under different power management configurations and transitions. When this testing is created manually, it is a prohibitively complex task, especially if you need the testing to drive toward some coverage closure across not only all power states, but also all legal power state transitions. For these reasons, ST looked for ways to better automate the use-case based test development process using a coverage driven approach. This paper will show how automation enabled ST to improve test development by an order of magnitude, accomplishing testing goals with fewer people in days, what would have taken weeks, requiring more developers and achieving less coverage.

LOW POWER IP VERIFICATION CHALLENGES

In the last years we have seen more and more power control and structure added in IP designs. The power management is not anymore done only at SOC level. Complex IPs like GPU also have embedded power management control.

In ST's GPU team, we adjust power management structure and control according to the ST Technologies rules. Starting from third party GPU RTL, we tune or add power control according to the ST internal power specification. We swap generic switch with ST power switch and generic IP provider memory with ST memories. Some GPU providers already insert power structure in their RTL, but to meet ST technology requirements the power structure had to be adjusted. Many providers add clock and reset gate control to reduce dynamic power, in this case, ST generates a power island in line with the clock and reset gate path to reduce static power.

So our task consists of adding or tuning power retention, power island, power management state machine, memories, power switch and their controls to the RTL received. Of course, we also need to verify these changes to check that they are in line with power specification and that we haven't inserted any bugs.

On earlier projects, the power verification was pretty simple. There were only one or two power supplies to switch ON or OFF. We now have more and more complex power management with more power domains, more power values. The number of power states available in UPF increase drastically and the number of signals to manage them also increases. To move from one power state to another, some rules must be applied regarding the clamp, the retention sequence, the clock value, and more.

We need to completely verify the power states defined in the UPF, implementation tools use only this information and if a power state is missing in the UPF, but is mandatory in the power sequence we could be in trouble. Some protection or repeater cells could be added in layout tools and no trouble is highlighted by static tools, but in the silicon we may face a current peak or wrong control signal value.

We need to provide integration tests to SOC verification team, SOC validation team and also SW team. Methodology like UVM could not be used since the tests are not portable. All the tests are written in C and are run on a Host Code Execution (HCE) platform at IP level.

We created a C API to manage and control all the power functions. Some functions are directly accessing the power logic in the IP, while other functions are controlling input pins or issue some specific Verilog commands (to set power supply level for example). For these we have created a dedicated power control module in Verilog that can be controlled through the C code as shown in figure 1.

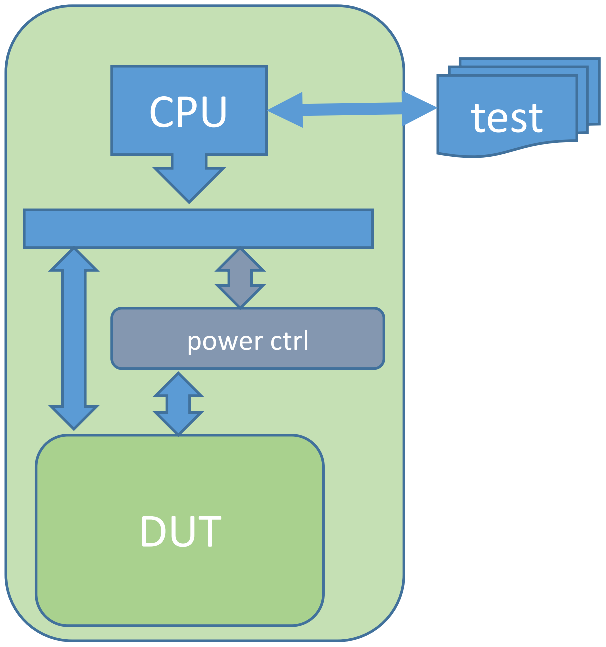


Figure 1: Power Verification Setup

Before describing the automation methodology and tool, let's have a closer look at our test requirements.

TEST REQUIREMENTS

There are several power elements to configure in order to reach a functional state, state in which the IP is ready for operation. Among these power elements the main ones are the different power switches.

If we consider for example, two different power switches: power switch A and power switch B. Both power switches can have different values: OFF, nominal value, overdrive1 and overdrive2. Most of the time these two power switches are not independent, they are linked by a rule that must be respected at any time of operation. Let s consider a simple rule: power switch B value must be less or equal to the value of power switch A. This is represented in figure 2.

In this graph, the blue triangle is the initial state, both power switches are turned off. The green points represent all the power functional states (6 different) and the red squares are possible transition states (power switch B is off).

Note that power switch value must be configured sequentially, so it is not possible in this example to go directly from OFF value to overdrive1 or overdrive2 for example. To reach one of the functional states, we need to configure the two power switches by creating step by step transitions, but always below the gray line (representing the rule: power switch B value <= power switch A value).

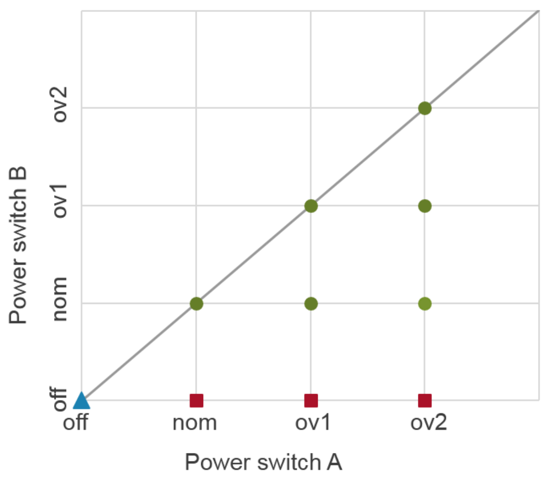


Figure 2: functional power state example

The tests we need to develop must all:

1. Move the design into a functional power state
2. Run a functional test on the design to check functionality has not been impacted by low power logic

For example we may decide to create a test in the functional state: power switch A = ov2 and power switch B = ov1. There are different possible path to reach this state. Two examples are provided in figure 3 and figure 4.

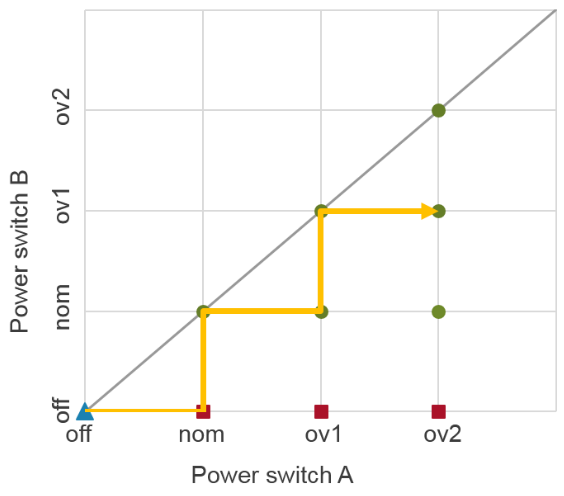


Figure 3: path to a functional state (1)

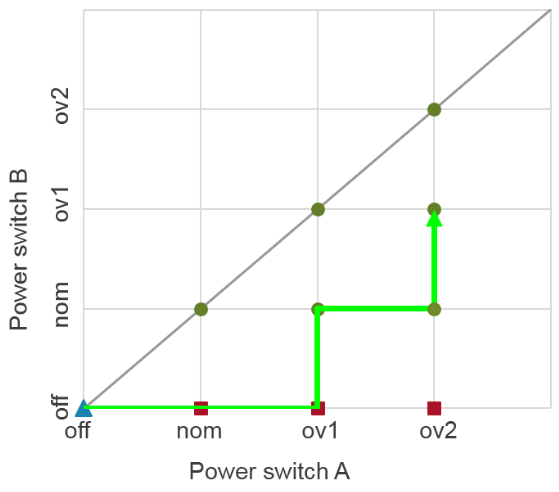


Figure 4: path to a functional state (2)

So far we talked only about the power switches, their values and rules that must be followed to reach a functional state. But there are additional power elements that must be taken into consideration and properly configured to be in a functional power state. The three other power elements are the clamp, the reset and the clock frequency, one for each power domain or power switch. Configuration of these elements must also follow some strict rules. For example the clamp and reset can be configured only when the associated power switch is not off (whatever the value). Some clock frequency can be set only if the power switch value is greater than a specific value.

Also in the examples, we are describing the path from initial state (all power switch off) to a functional state. But we have to describe the path back to initial state too in order to create more complex scenarios like: go in a functional state, then switch off the power, then go in a different functional state.

Once the scenario is described, we need to put this scenario in a C code. Typical C code following path defined in figure 3 could be:

```
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 different C code could be written to check this functional state. We can for example change the transition and the path from initial state (following path defined in figure 4 or creating a new one). The place where the clamp and reset are configured can be changed anytime between the turn on of the power switch and the call to run_functional_test. The clock frequency could also be configured at different time and with different values.

On this simple example we already see some limitation of the traditional manual test development flow. The number of functional power state in example from figure 2 is 6. So we would need to develop at least 6 tests to check all of them. We would also need to develop the tests using different path, going through the non-functional state. Configuration of other power elements including using different clock frequency could also be changed.

Here the rule between the two power switches are pretty simple, we often face other cases like the voltage difference must not exceed a specific value. Moreover, when additional power switches are present, they may all have link and rules that must not only be applied between power switch A and power switch B but also between power switch B and power switch C for example. Finally, during the project, changes in the power specification could occur, changing the possible values of a power switch, and/or changing the rules between them.

In our GPU design, the number of power switches could be up to 6 with complex rules between them and complex additional power elements to configure. Creating a single test, reaching a functional state and going through only valid transitions, requires a deep knowledge of the entire power specification. Time is spent on creating the test and also in debugging it when wrong transition are created. Due to time and resource constraints we cannot develop and maintain test for all the possible transitions to all possible functional states manually. We would have to make some trade off and arbitrarily select a few of them. Also, with manual development a lot of tests are always using the same transition path since a new test might only be an extension of an existing one.

In summary, with the traditional manual approach, we don't have a clear picture of all the possible combination of functional state and other power element like clock frequency. A systematic approach would be needed to list all of them and create a test for each of them. Because of limited time and resources, it was infeasible to do this, so we created and maintained 20 tests manually. Not all functional states could have been covered this way: the 20 tests could cover a maximum of 20 functional states. We were looking for a different approach that would allow us to reach all the possible functional state in order to fill the holes we knew we had in our flow.

NEW METHODOLOGY USING TEST GENERATION AUTOMATION

The new methodology we developed based on automation tool offered us 3 important capabilities:

- *Model based approach: An easy and abstract way to define and constrain the SoC power management capabilities*
- *Goal directed test creation: A use-case based solver technology to automatically identify path to reach a functional state*
- *Automated generation of tests to achieve 100% coverage for the specified goals. Reaching all possible functional states and randomizing transition paths*

In addition to these three capabilities, we also found it helped us maintain the test database and with creating more complex use cases. This paper focuses on how automated test generation improved our coverage which was ultimately measured as productivity.

With the model based approach, focus is not anymore on defining completely a test from the initial state to the expected functional state. The solution is to describe independently all the power elements available in our system. The description must contain all the possible value of a specific elements (possible clock frequency, possible values of a power switch …). It must also contains all the possible changes or transition of this element. Power switch may go from OFF to nominal value, from nominal to overdrive1 … The rules under which the transition can occur must also be described. Finally, the corresponding API call to this transition must be associated to it.

This approach is completely different than traditional test development flow but really allowed us to focus on the details of each element. We had cases where listing all possible transitions enabled us to find holes or unclear points in the power specification.

If we look at the power switch B of our previous example (figure 2), we can extract the following information:

Power Switch B can take value: OFF, nominal, overdrive1 and overdrive2. The list of transitions, rules and associated API call are defined in table 1:

TABLE 1

POSSIBLE LIST OF TRANSITION OF POWER SWITCHB

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);

Note that the rules with the (*) are implicit rules. These rules are not mandatory since an equivalent rule should be put on power switch A. But we decided to write them in any case for clarity and debug reasons.

Describing all these transitions is the longest task in this methodology. However, the time to define this was shorter than developing the tests manually. This task has to be done once for all the tests and it really forces us to have a clear and complete understanding of the entire power specification. We do not need to know or remember all of it, just the specifics of a power element when we want to describe it. All these descriptions/models will be shown later in the paper.

Once this task is completed, we can now use the goal directed test approach. This means that to describe a test we just have to ask for a specific functional state and the automation tool will identify a path, from the initial state to the expected state, following the rules defined in the model. For example, we can request a test to be created for the functional state: power switch B = ov1 and power switch A = ov2. We ask for what we want and not how we want to reach it.

The goal directed test creation is very powerful in the sense that each generation may create a different path to reach this state (assuming several paths exist, which is typically the case). The test itself, is reduced to the simplest expression: what is the expected state.

The other advantage is it is not possible to create an invalid test. If a user asks for an invalid state (state not reachable due to the rules), in this case the tool will immediately report an error saying that this state is not reachable. The tool will also give you the reason why based upon the rules. So by construction, no illegal transition can be created, the tool will allow you to force generation of an illegal test if you want to perform negative tests.

After having created a few tests to debug the model, we can then proceed to use the automatic generation capability to fill all our goals. In our case we can ask the automation tool to create a test for all possible functional states. Here the request would be: power switch B = all possible value and power switch A = all possible value. The tool will then try to create a test for all possible combination. If some are not possible due to the constraints, no test will be created and message will be printed. This is again very useful for debug. Note that we don t need to know or think upfront about all the possible values. The tool will automatically identify them.

On the GPU IP, we have created 192 tests to meet our coverage goals with this approach. All the tests are different by construction and reach a different functional power state (or a cross with a power state and a different clock value for example). We also know that paths used to reach all of these state were randomly selected and that the other power elements have been configured at different point of time. We don t have real coverage on the other power elements configuration since this was not our requirements but we manually analyzed some of the test and noticed the randomization was also applied here.

In summary, the methodology has a lot of advantages. First it enables us to divide the power complexity and focus on each of the elements one by one. This modeling of the power elements creates a kind of embedded specification of the power specification making it easily readable or usable by anyone even without deep knowledge of it. This gives us the ability to reuse the power transition scenarios for other IP if needed. In case of changes in the power specification, instead of rewriting or analyzing changes required for all developed tests, here we can simply report these changes in the model (adding/removing rules, power elements …) and create new test suite matching the new specification. The goal directed approach is very efficient and can create new tests in seconds.

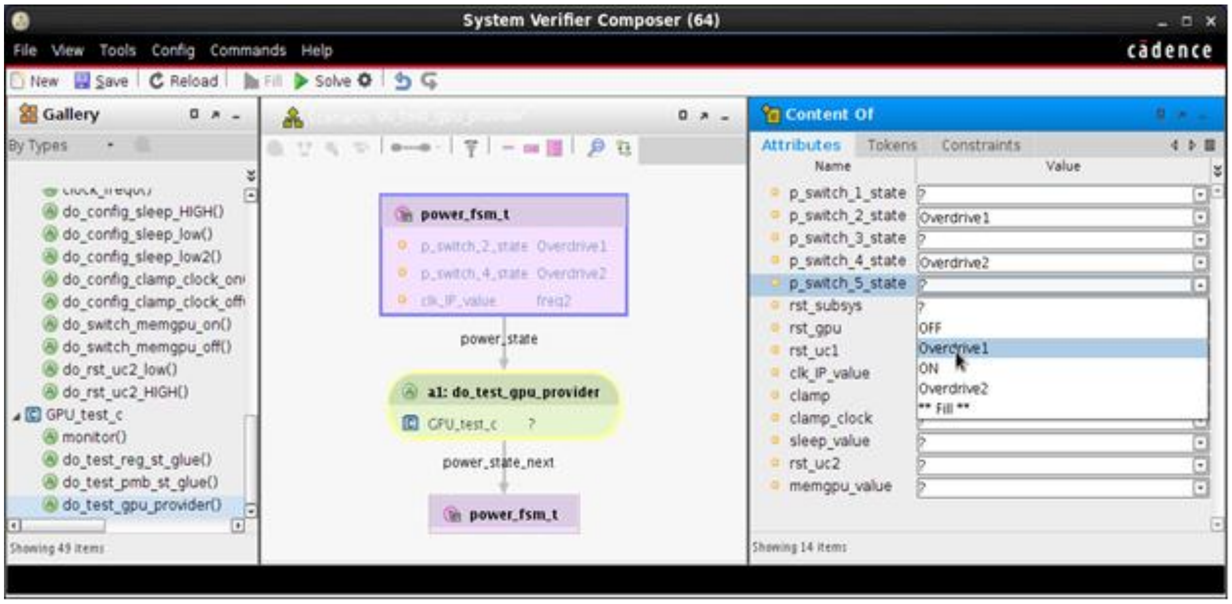
The methodology has also some constraints. It requires extra work for the first test. It also means developing a new model and learning a new language. The entire power specification needs to be modeled to get the most of the automation capabilities.

CHOICE OF AUTOMATION TOOL: PERSPEC SYSTEM VERIFIER

We developed and use this methodology with new Cadence tool: Perspec System Verifier. This choice was done mainly because of the model based approach that fit our requirements. The generated test is also graphically represented in a UML activity diagram showing clearly all transition from initial state to functional state. This graphical representation is very useful to analyze and debug the model developed and the test. It is also a good way to communicate and exchange with design and architecture team that don't necessarily write or work with c code for example. Also we found the C code generated very linear and simple to read and analyze, again this helps us in the debug phase.

To apply the methodology, we had to create the power specification model with the input language of the tool. The first model has been developed with local application engineer support in a few days. Since there is not much language to learn, other models were developed by ST with limited support.

Then, as in figure 5, the test can be created in a goal directed way by selecting graphically the expected values for switch A and switch B as well as reset, clock and clamp parameters.



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	?

Figure 5: goal directed test creation with Perspec System Verifier Composer

Once the goal is selected, the user can ask the tool to create a solution. Figure 6 represents an extract of the entire solution. A UML activity diagram from initial state to requested state is created.

Then to create the entire test suite, reaching all possible functional states, the user can ask the tool to FILL (create a test for all possible values) on attributes of interest as shown in figure 7. The result of this will be a list of different test cases, all different for reaching different states.

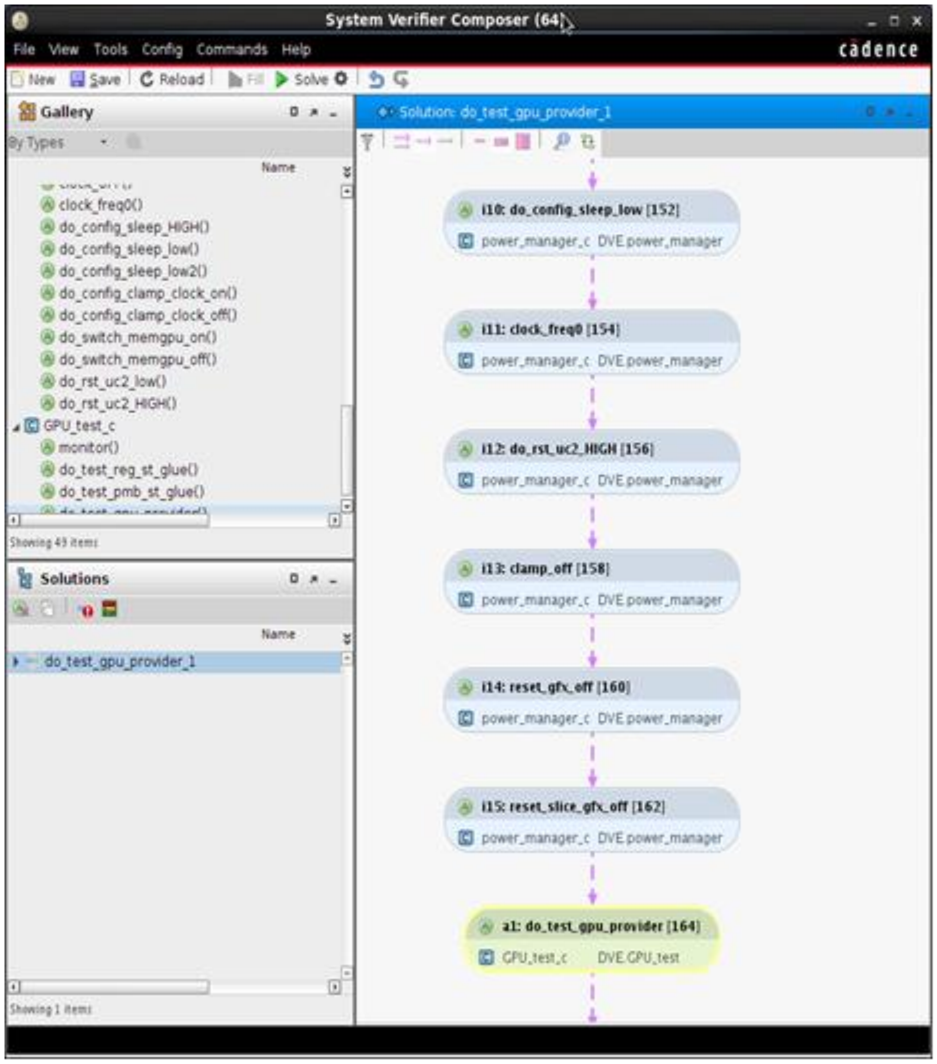
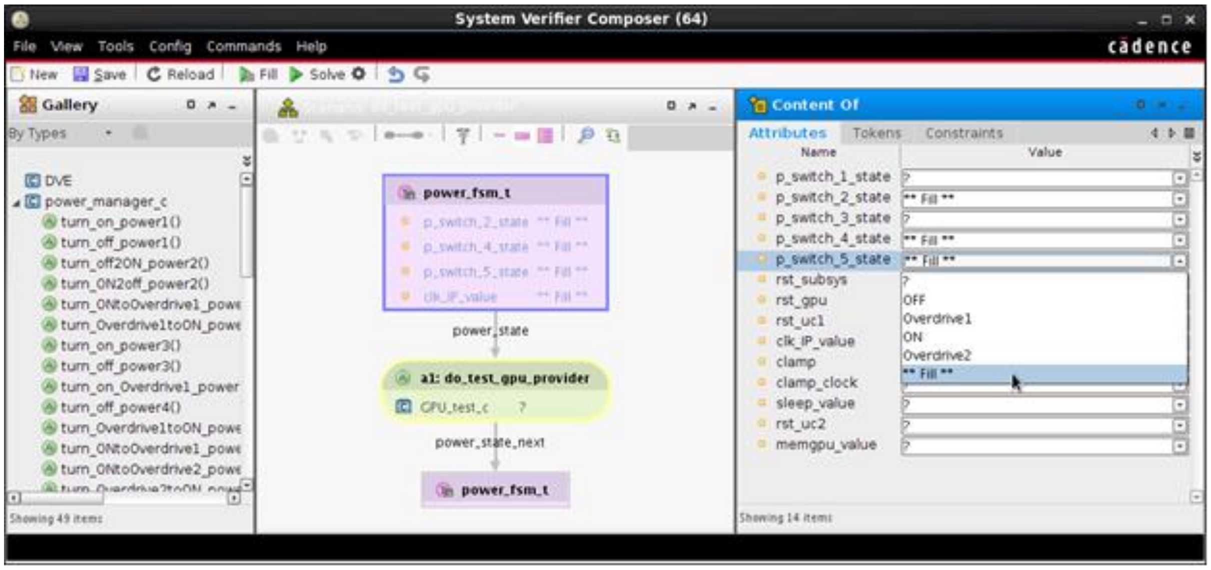

System Verifier Composer (64)
File View Tools Config Commands Help
cadence
New Save Reload Fill Solve
Gallery
By Types
Name
clock_freq0()
do_config_sleep_HIGH()
do_config_sleep_low()
do_config_sleep_low2()
do_config_clamp_clock_on()
do_config_clamp_clock_off()
do_switch_memgpu_on()
do_switch_memgpu_off()
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
Solutions
do_test_gpu_provider_1
Showing 1 items
Solution: do_test_gpu_provider_1
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

Figure 6: Excerpt of Concrete Use-Case Scenario



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	?

Figure 7: Setting FILL Option to Create Use-Case Solutions to reach 100% Coverage

RESULTS

With automation we were able to generate tests that covered all legal power states and power state transitions to achieve higher coverage than manual test development in less time. The model based approach enables the solver to identify paths or state transitions a user may not think of or did not have time to implement.

Table 2 describes the productivity gain, comparing the number of tests, the number of lines of code for all these tests and the time to get the tests. It also provides information about the gain in case of specification changes.

TABLE 2

ANALYSIS OF PRODUCTIVITY GAIN REALIZED

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

With less number of coded lines, user can create more tests and increase productivity from 1 test per day to around 20 tests per day. These data represents the time to create a functional test, so it includes debug time.

Note that to reach the same level of coverage with manual test development, we would need to develop these 192 tests, estimated to take 192 days. Thanks to automation it took only 10 days

The coverage of generated tests has been confirmed by running the entire test suite and analyzing the effective power state coverage obtained in simulation. This is a very effective way to cross check the model developed with the UPF.

Some bugs have been found in the internal checkers. These checkers should verify that no invalid states are reached. By stressing the design in some specific states or transitions, the checker were proven to be not correct and were reporting false errors.

Also the UPF file was not consistent. Some states were defined but were not reachable, while others were not described at all. Both could be found in simulation by either analyzing the coverage report or by automatically creating the checkers.

CONCLUSION AND FUTURE WORK

Thanks to the automatic generation of all possible use cases we increased our confidence in the low power logic added on this IP and dramatically improved both our coverage and productivity. Due to the complexity of today’s low power verification, traditional manual test creation cannot cover the entire power space and automation is a must. The tool used, Perspec System Verifier test creation enabled us to even tackle a more complex verification task and at the same time improved our productivity by an order of magnitude.

The methodology described here can be applied to any low power verification and more generally to any complex FSM verification. This approach could also be beneficial for the SOC verification teams. As of today we provided the SOC team some integration tests (this might be a subset of the entire test suite to match additional SOC constraints in possible low power sequences). The model based approach allows combining different models and creating more complex use cases. So, in the future we could provide to the SOC team the model we developed, they can add additional information and simply create advanced SOC level low power scenarios.

In our team, we have different kind of IPs to verify. Some might be even more complex. We have cases were 2 instances of one IP are present, both have the same LP logic added and they are independent. Complexity increases here, but using such an automation tool will definitely simplify our work and gives us more options on how to best verify use cases for these complex combinations.