Portable Stimulus Specification (PSS) and the Reuse Revolution

By:

Mike Bartley, Tessolve and
Sharon Rosenberg (Cadence)

Abstract

This PSS paper is collaboration between an expert user, Mike Bartley (Tesolve), with a senior solution architect, Sharon Rosenberg (Cadence) to provide a complete picture of the problem statement, a solution paradigm and technology, and the user benefits and impact of applying the technology.

TESSOLVE
A Hero Electronix Venture

The Challenge

As products become more complex and market windows continue to shrink, efficiency in product development can translate directly into competitive advantage. We understand what improves efficiency in verification: abstraction and reuse; but how to achieve that? Accellera are looking to address this through the Portable Stimulus Specification (PSS).

The reuse revolution started in design IP and has contributed hugely to design efficiency. Verification IP soon followed in the form of eVC and latterly UVC but these miss the main point: the currency of verification is stimulus (and checkers, of course). Also, verification has to deal with multiple dimensions of reuse:

- Hierarchy (block, subsystem, SoC, system)
- Platform (simulation, emulation, FPGA prototype, Silicon)
- Project (reuse from one project to the next)

PSS tries to deal with all these through abstraction.

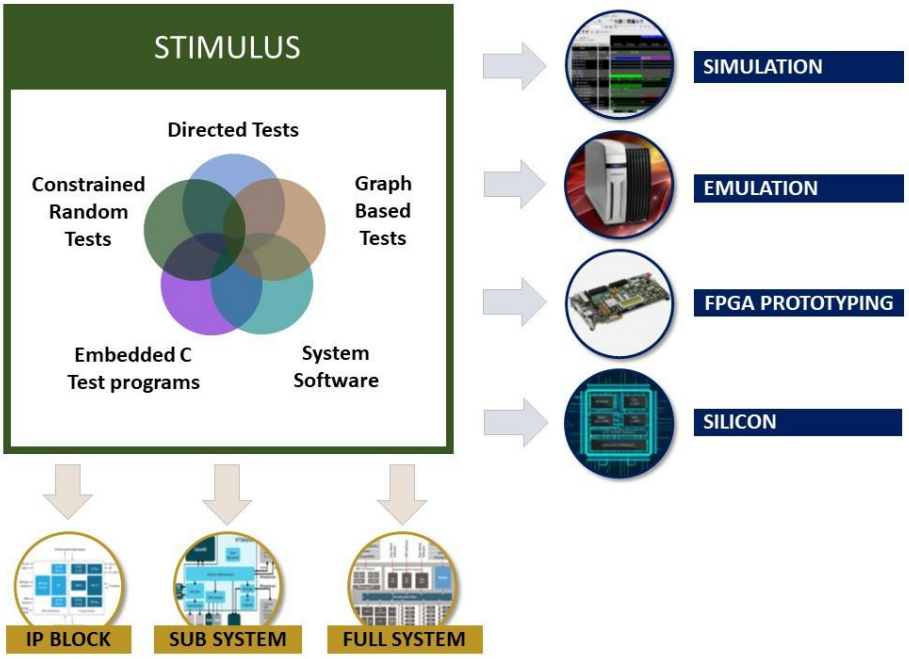

FIGURE 1: THE NEED FOR PORTABILITY

For demonstrating the PSS capabilities lets introduce a known and common industry challenge - the cache and IO coherency challenge. For optimization, both CPU-cores keep copies of the memory in close-by fast internal caches. If a CPU core cached and updated its memory block copy, it is key to ensure that no other core updates the common memory, resulting and incoherent cache. In order to maintain coherency, sub-systems must communicate between them (e.g. using snoops) to ensure no device is invalidating an already modified cache. Achieving such scenarios requires highly-parallel SW driven tests which can target both the inter-cluster intra-cluster, and IO traffic.

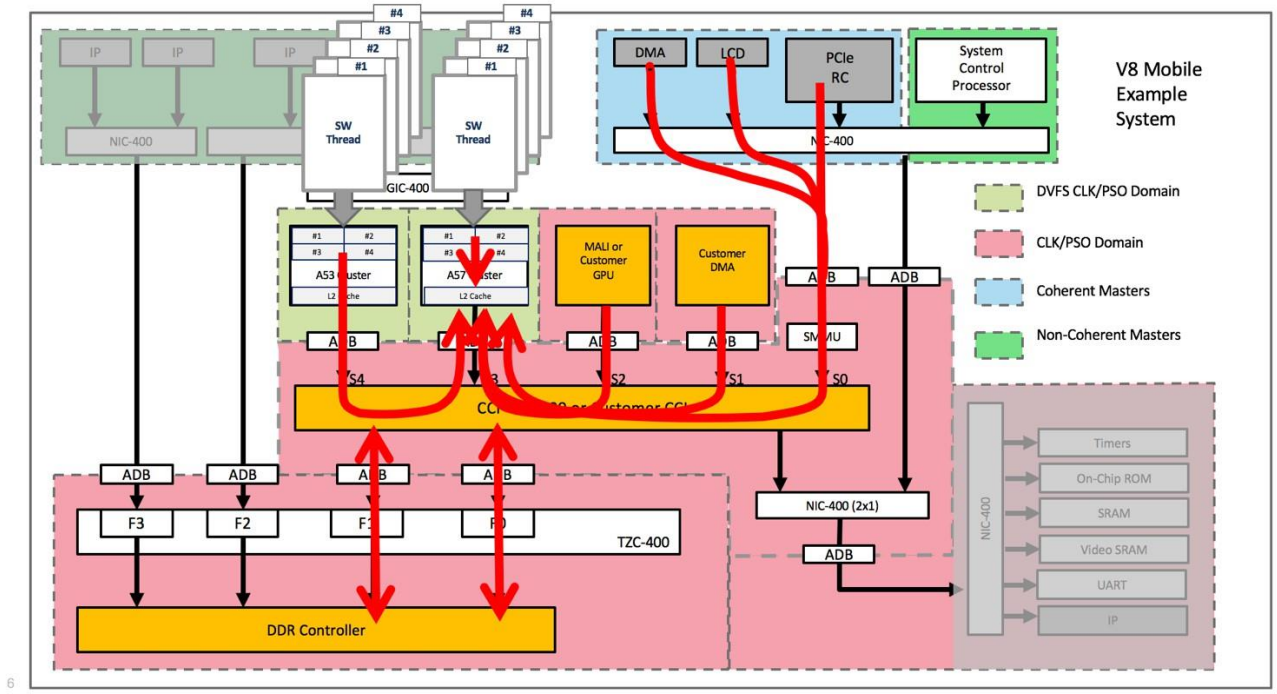

FIGURE 2: SOC COHERENCY TRAFFIC

The diagram above shows a representative system in which the well synchronized traffic (the red arrows), comes from the CPU cores, and peripherals to stress the system coherency mechanism. SoC designs often have 3rd party IP for their CPUs, GPUs, DSPs, and IO devices, but cache coherent interconnects involve custom, often complex integration.

The PSS Solution

To better understand how PSS solves coherency challenges, let us discuss PSS usage flow and concepts. As regards flow, in the PSS use model the user uses a two-step approach. In the first step, the user captures the use case behaviors in the form of actions and their combination rules / dependencies. In the second step, the user leverages the actions of the first step to define the desired use cases using PSS activities. The activities need not be complete; they can be partially specified. A PSS compatible tool can leverage the action's combination rules or dependencies to do any of the following:

Validate the correctness of the user activity request
Vonstraint randomization for legal attribute value assignment
Infer and add missing actions
Determine the legal scheduling given the available system resources.

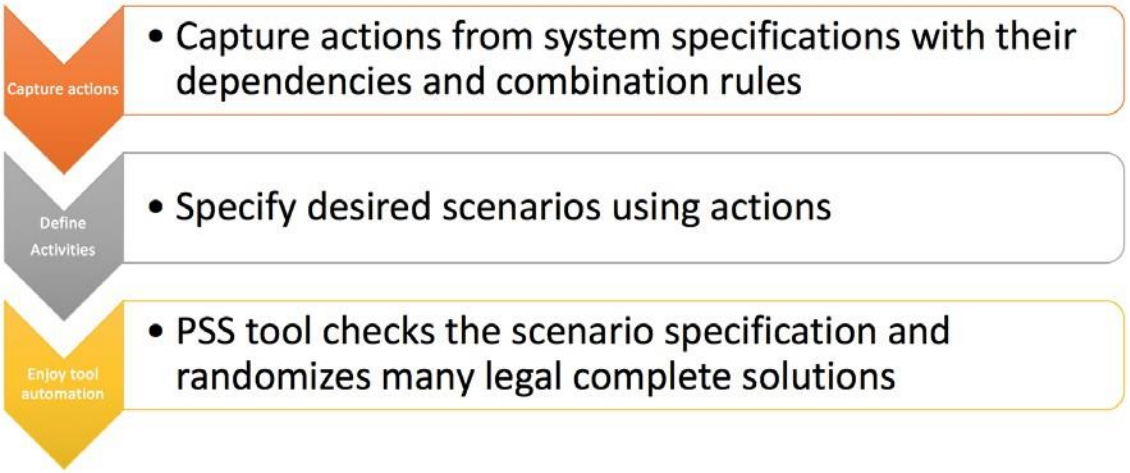

FIGURE 3: PSS USAGE FLOW DIAGRAM

PSS introduces several powerful concepts that can be combined to address multiple industry challenges. The figures below illustrate PSS actions that can serve many teams and use-cases on different execution platforms.

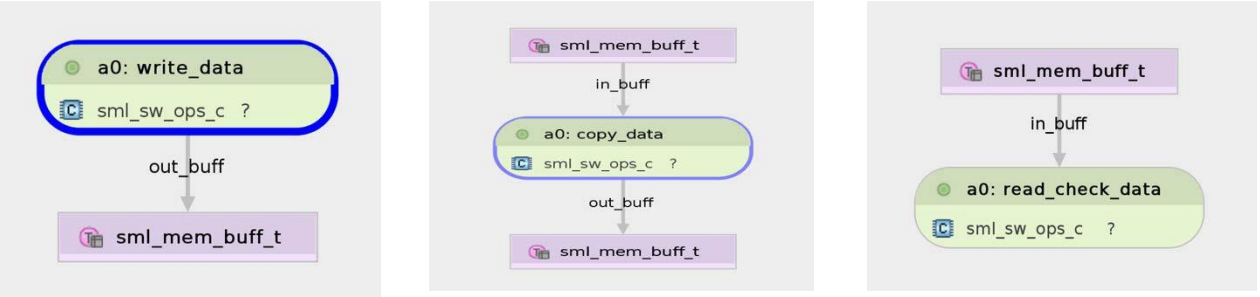

FIGURE 4: CPU OR TB BUS TRANSACTOR ACTIONS

Note that the same write_data action can be implemented by a master on the BUS or via CPU core to write the memory buffer. At the same time, the read_check_data action requires a memory buffer as an input in order to be executed. This is a dependency of the read_check_data. Let us now see an example of a scenario that can be achieved with this minimal vocabulary of three actions that we have just created. Requesting a PSS tool to executed a read_check_data action as a desired scenario will result in the following solution:

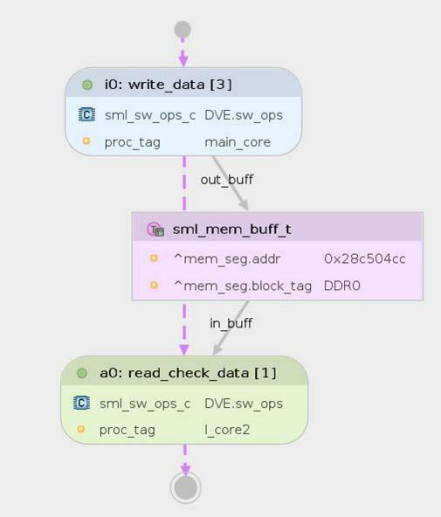

FIGURE 5: PSS COMPLETE SOLUTION FOR A READ_CHECK_DATA SCENARIO REQUEST

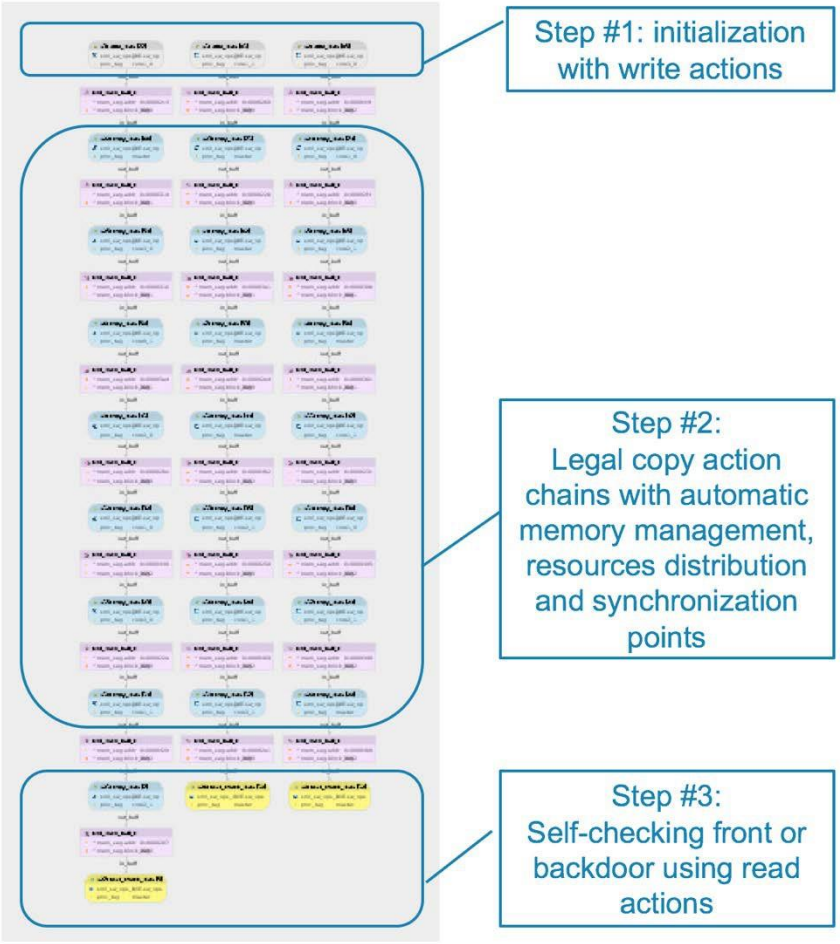

FIGURE 6: PARALLEL COPY CHAINS (TIMING AND DATA-PASSING DIAGRAM)

Using the action model and combinations rule, a PSS tool inserted the write_data action to be executed before the read_check_data. It will randomly assign processors or a master to execute the write and read actions and insert sync points between them to make sure that the reading begins only once the initialization is done. PSS allows the requesting of sophisticated use cases and

scenarios using compound actions. Compound actions have an activity block in which users can request more sophisticated, procedure-like flows. For example, to stress and interconnect, the user can request three initializations, followed by multiple legal copies and a read_check_data at the end, and get something like this:

In the above example, action types are shown in different colors to make it easy to see that we start with three initialization write actions (in gray), followed by multiple copy actions (in cyan) and finishing with check actions (in yellow), exactly as requested. The memory buffer and the attributes are colored in pink. The participating processors and the accessed memory locations are all carefully selected to enable this highly parallel scenario.

But how are these three actions relevant to cache and IO coherency?

PSS allows layering constraints from above at both the actions and the flow-objects (in this case the memory buffers). This means that I can constrain the memory buffer location to be in a specific cache region, and stress the cache mechanisms by forcing the CPU cores to snoop and invalidate other CPU cache lines. While extremely powerful, this is not enough for coherency verification. PSS allows self-checking that user can implement in both front-door (the CPUs are checking the result), or back-door (a backdoor task wakes-up at the right time to perform the check). If all the processors randomly write in an uncontrolled manner to the same location, how can the testbench predict the expected result?

There are two strategies to overcome the self-checking challenge. The first one is called “false sharing” and the second one is “true-sharing.” In the false sharing mode, though all the cores are writing at the same time to the same lines, each CPU core is writing to its own dedicated bytes, so that there are no actually simultaneous writes to the same bytes. The figure below shows a false sharing scenario:

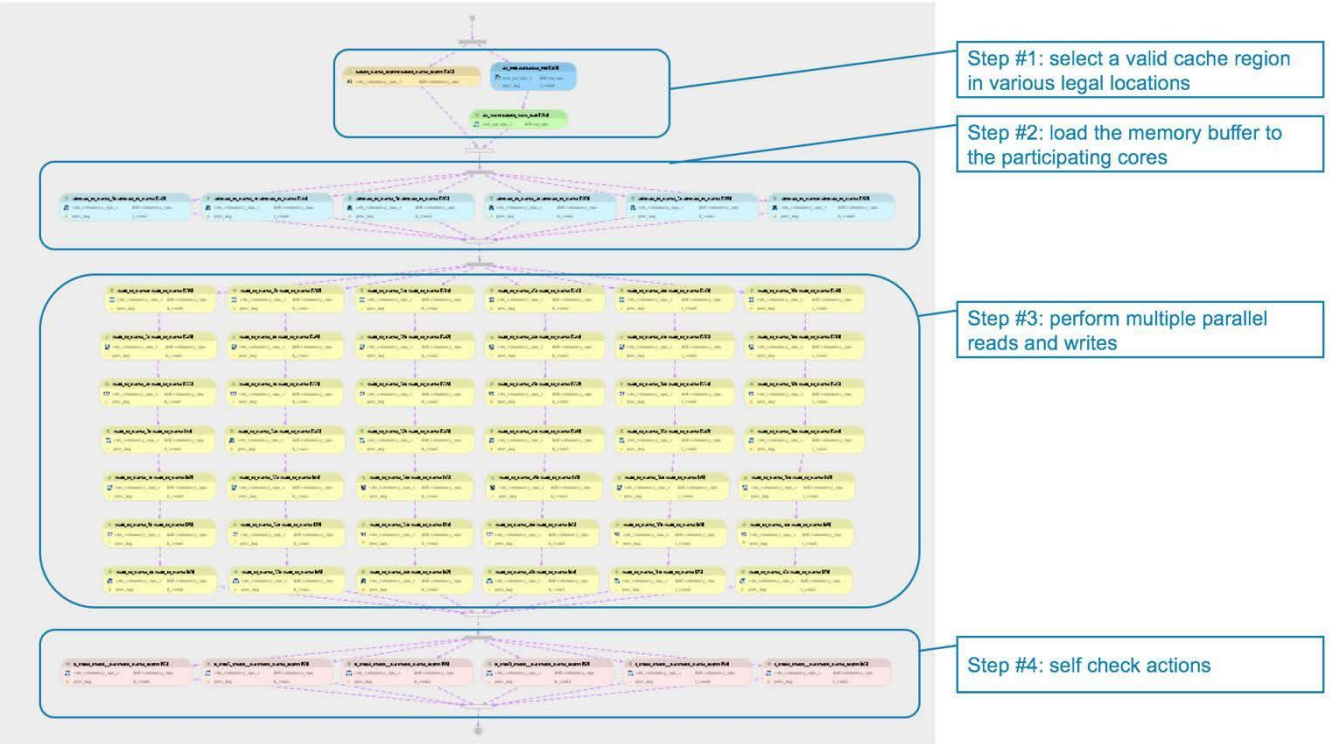

FIGURE 7: COHERENCY FALSE-SHARING SCENARIO (TIMING-ONLY DIAGRAM)

In the diagram, you can see the execution steps, where the model

Selects a valid cache region,
Loads the memory to the participating CPU caches,
Performs multiple parallel reads and writes, and
Finalizes with a self-check at the end.

Note that this screen shot is showing only the timing information of the scenario without the flow-objects that are passed between the actions. Here the cores do reads and write operations without any synchronization between them.

The true-sharing coherency strategy allows the CPU cores to write the same memory addresses (hence true sharing). To allow checking, the tool provides the needed sync point to predict the expected result.

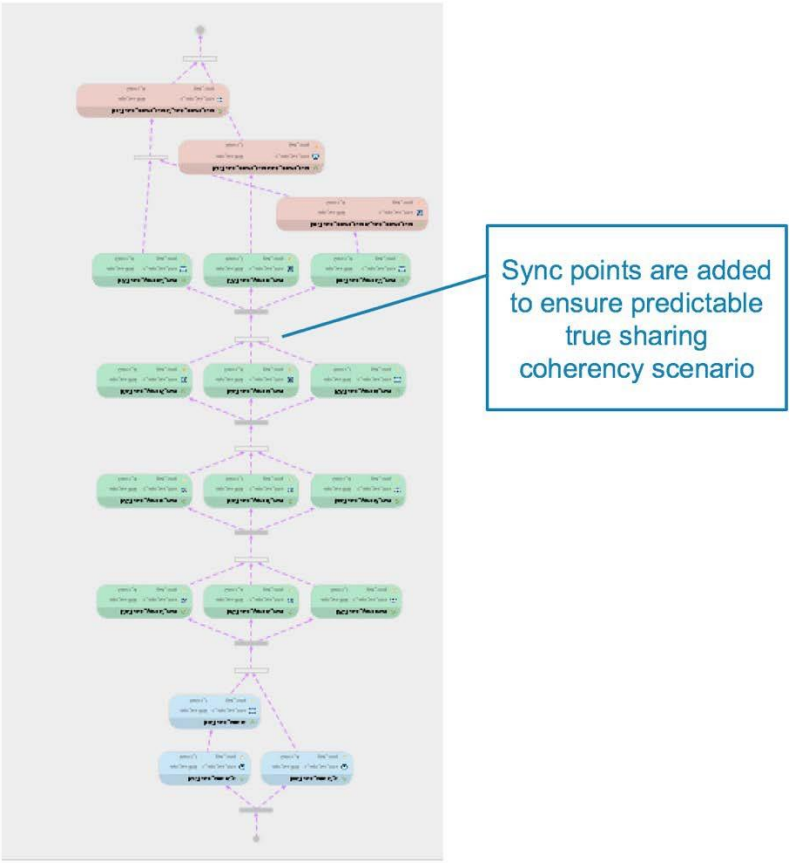

FIGURE 8: COHERENCY TRUE-SHARING SCENARIO (TIMING-ONLY DIAGRAM)

In the true-sharing scenario above, after loading the memory buffer into the CPU-caches (cyan), the activity between the CPU cores are well synchronized (green). This allows prediction of the expected result (pink). But what if I need to connect my I/O device to participate in the coherency scenario. For example, the PCIE with its own SMMu might write to a cacheable region. Will my system support this scenario?

This is where the PSS concepts are extremely useful. Just like any other action, the Root complex and Endpoint devices can read and write memory buffers. Modeling the PCIE devices in PSS provides seamless integration with coherency scenarios.

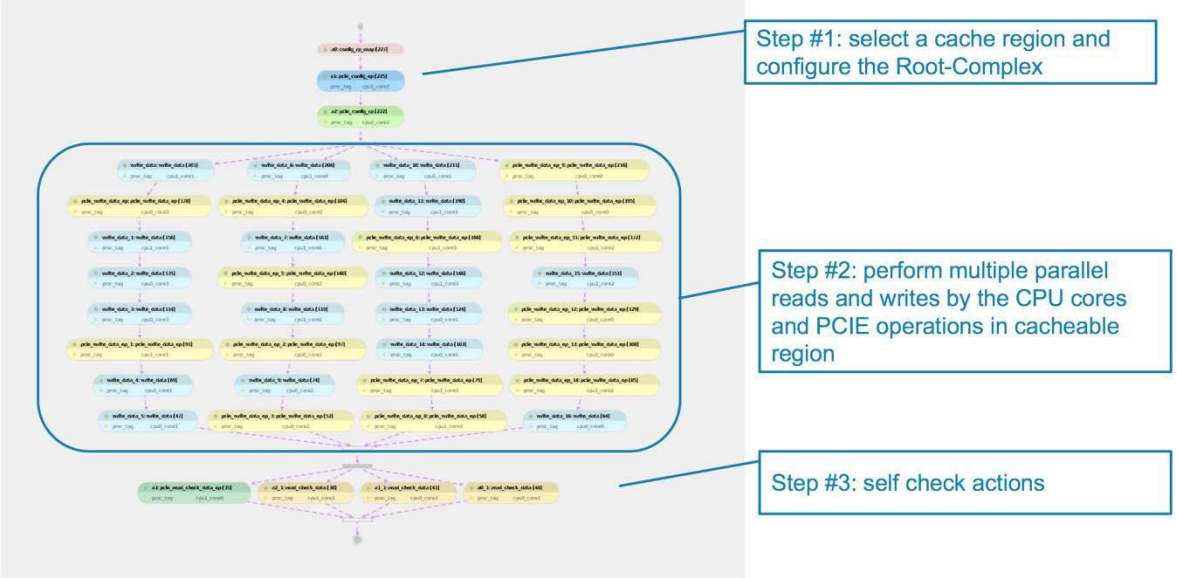

FIGURE 9: CACHE AND I/O COHERENCY WITH THE ARM AND PCIE ACTIONS

Packaging Actions for Reuse

True-sharing and false-sharing are just two examples of actions that could be packaged into a reusable library. For example, Cadence provides SOC action libraries that allow coherency, low-power, memory virtualization, connectivity, and performance scenarios. Some of the actions are generic for any architecture, and some are tuned for ARM architectures, generating combinations of highly-parallel C and assembly tests. Cadence also provides a PCIE library that can validate the integration of the PCIE controller and could be mixed with the SOC actions to achieve other interesting SOC scenarios. All libraries can be configured for specific project needs and be mixed with other user-defined actions to efficiently achieve project specific scenarios. The PSS action dependencies and combination rules allow the PSS tool to produce legal SOC scenarios for both SW driven and host driven (transactional) configurations.

Conclusion

The most noticeable impact of using PSS is the automation and organized process that this approach injects into SOC projects. Test creation in SOC environments is often manual work that can be both error-prone and time consuming following an iterative development process. PSS allows harnessing technology, such as Perspec, for automating the task of test generation. The tests are correct-by-construction as the action model ensures legal completion of the multiple SOC dependencies. Users can use functional coverage to achieve a well-defined process of setting a goal and ensure timely convergence of the verification process. PSS allows for projects that started with a compromised test plan that was limited by their test-creation capabilities transition to a more desired, thorough plan when they adopted the automated approach. Alternatively, users can decide to reduce the overall verification time to achieve the same level of thoroughness. Many times, a library provides more use-cases that the user initially aimed for, including coverage, a verification plan that can be adapted according to needs and self-checking tests without writing a single PSS line.

The other value to be mentioned is the portability and reuse aspect. As mentioned at the start of this blog, verification has to deal with multiple dimensions of reuse: Hierarchy; Platform; and Project

(reuse from one project to the next). PSS tries to deal with all these through abstraction. We have discussed this in the context of memory coherency where multiple CPU-cores and peripheral devices are sharing memory whilst also keeping locally cached copies for performance. Without a library, our first step is to define some core memory-related actions (such as write, read-and-check) with rules for combining and synchronizing them. We can now define the CPU cores and peripherals onto which those actions can be mapped as well as adding constraints (e.g. defining memory regions to force true data sharing). Synchronization allows us to know the actual ordering of reads and writes and hence to know the expected results for self-checking tests. The actions and produced scenarios can be mapped to either transactions or software making them portable across multiple simulation and emulation environments. The software tests can also be ported to silicon. Thus, PSS has allowed us to define truly portable test through abstraction.