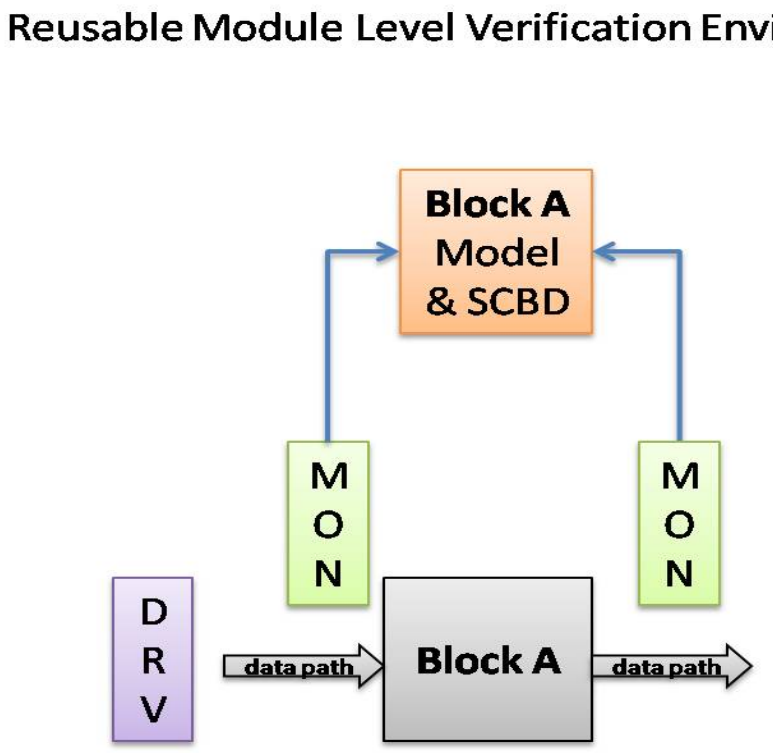
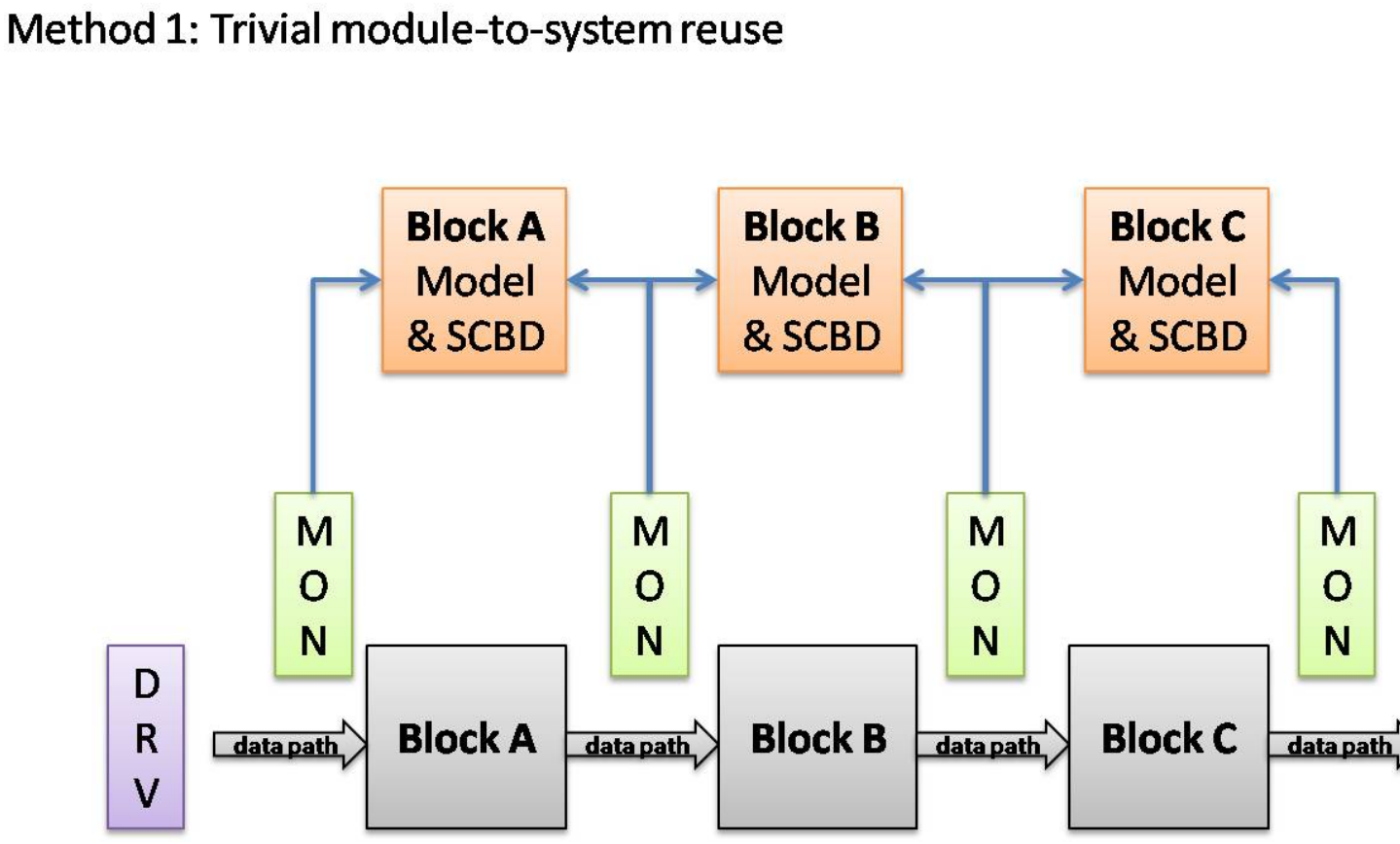
Time to talk about module-to-system reuse, a very important topic. If you plan your verification environment properly (using one of the common methodologies in the market today or your own) you'll be able to easily build a system level verification environment that reuses most of your module level environments (i.e. sub-environments). However, even if all your sub-environments are well suited for plug and play reuse at the top level, there are still considerations to be made regarding the overall topology. In other words, how do you go about connecting the sub-environments to each other to make an effective top level environment? Here are 3 methods that you can use.

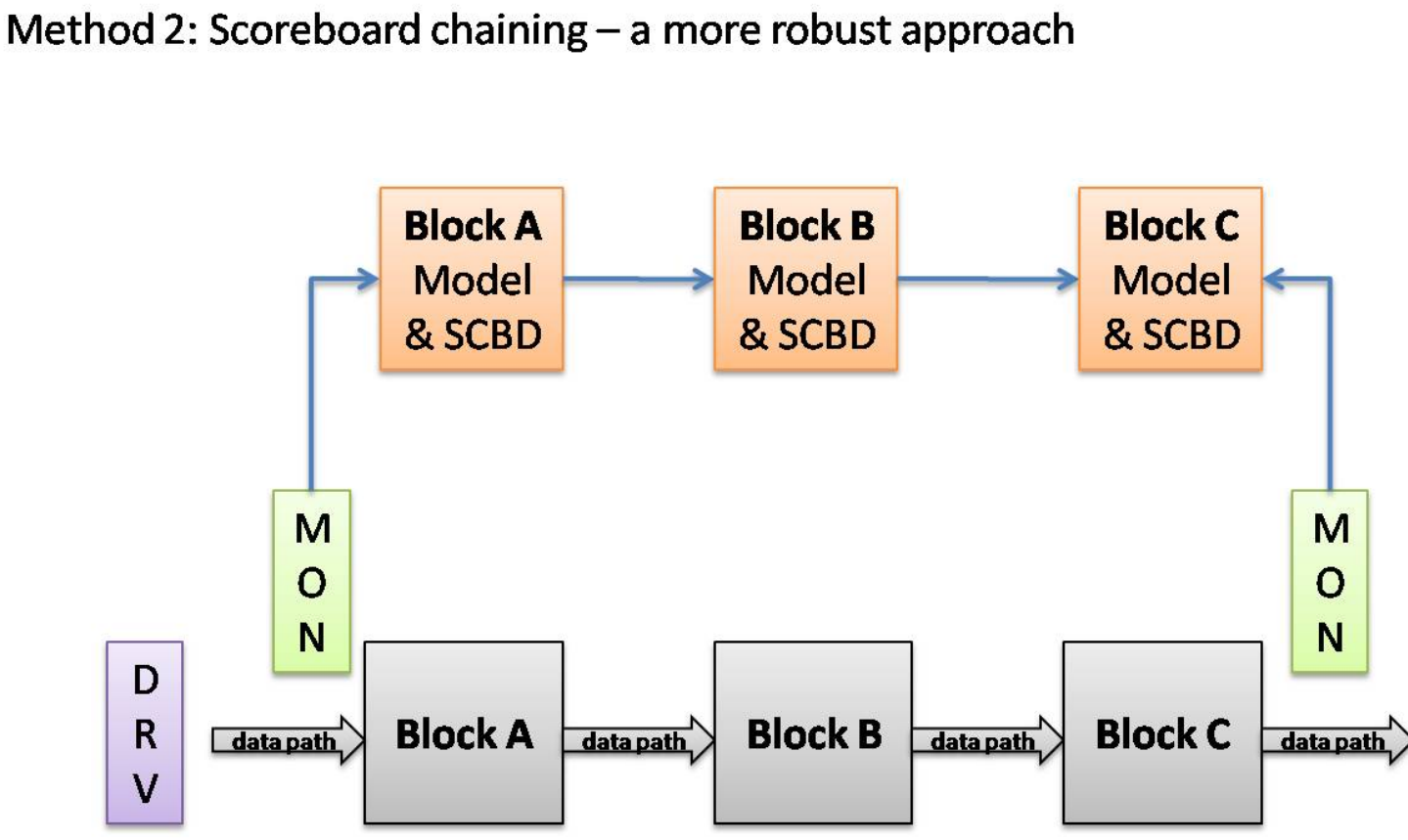
It is assumed that all of your module level environments look more or less like this:



The first method is the classic reuse method where you're virtually instantiating all your sub-environments at your top level environment – simple, quick but a bit redundant. Note that you can save some code if you only instantiate a single monitor (instead of two) on internal interfaces. In fact, if you've been a good boy at the module level, you might have thought about this in advance and placed a monitor **reference** rather than **instance** to facilitate smoother reuse. But even if you didn't, this shouldn't be much of an effort. Eventually you'll get something like this:



If there are many blocks in your data path and the internal busses that connect them use non-standard protocols then you might find out that having multiple monitors along the data path could introduce a significant amount of false alarms. New RTL releases and bug fixes often introduce minor timing differences or other changes to internal signals that don't necessarily affect the end-to-end data path, but might mean that you'll have to adapt your internal monitors with each revision. Ouch! From experience, when you're doing top level verification – the only thing that counts is the overall functionality. If internal logic has to be modified to make the chip work – that's what they're going to do, and you're stuck with a bunch of out-of-date monitors. The solution is such cases is fairly simple and is called scoreboard chaining. In this method, the scoreboards (or reference models) are daisy-chained together to create a virtual end-to-end scoreboard. Here's what it looks like:



One disadvantage of the previous method is that by eliminating “internal” monitors you’re also eliminating precious debug information. In case the end-to-end scoreboard reports an error, you’ll have to dig in the code to locate the problem and potentially go through several interfaces until you reach the offending one. Good news - there’s a third method that you might want to try in that case. I’ve used this method successfully in one of my projects (after the first two turned out to be less efficient). It’s kind of like a way to enjoy both worlds. You keep all (or some) internal monitors alive, but you don’t get any false alarms because the the scoreboard/reference models are fed by the external monitors only! The internal monitors are simply there to monitor signals and provide debug information (to log file or something). They no longer have the power to affect test result, but they still can help you locate and track data items that flow through the system. Now before you shout out at me, let me clarify - this method is not ideal, it’s just a practical approach that worked successfully for me in the past and might work for you too. But as always with verification, the trick is to match the most efficient solution to your specific problem. But anyway, here’s what it looks like:

Method 3: Scoreboard chaining with extended debug capabilities