White Paper

Taking the Pain Out of UVM
Leveraging Portable Stimulus to Eliminate Classic UVM Issues

Version: 1.0
February 2019

Breker Confidential

Abstract
The Universal Verification Methodology (UVM) has proven to be highly effective in establishing common testbench coding methods, enabling reuse and improving the comprehension of tests. However, the methodology still has limitations that particularly impact complex block verification. The Accellera Portable Stimulus Standard (PSS) allows for many of these limitations to be eliminated, while still leveraging existing testbenches, thereby not wasting legacy effort. PSS, through the Breker tools, enables a white-box approach to test authoring, allows complex multi-threaded, synchronized sequences to be generated from single scenarios, automatically provides scoreboard checks and coverage models, and generally improves test reuse and verification use models. This paper demonstrates how PSS may be leveraged in UVM environments to realize these and other advantages.

Breker Verification Systems, Inc.
www.brekersystems.com

Copyright Breker Verification Systems, Inc. ©2019 All Rights Reserved
1. Introduction: Augmenting UVM

As designs have grown in size and complexity, verification teams have been under increasing pressure to increase quality without a corresponding change in cost. They have managed to do this using languages and tools pushed far beyond their intended application. SystemVerilog and UVM were built to tackle much simpler blocks than we see today, and engineers now need an environment that will help them increase their productivity further. Portable Stimulus tools, coupled with UVM deployment technologies, can provide this productivity improvement.

If one asks an experienced, UVM-literate verification engineer what their top three time sinks are when using the methodology, invariably they will say:

• Writing synchronized complex sequences for multiple ports
• Writing checks and the scoreboard
• Writing functional coverage models

All three of these, somewhat independent, tasks in UVM are ripe for improvement.

The Accellera Portable Stimulus and Test Standard (PSS) approach, coupled with the right tooling, can significantly alleviate the associated aggravation. In fact, there are many benefits to applying PSS to a UVM environment that we will make clear.

Luckily, using this approach there is no need to throw away current skills and accumulated Verification Intellectual Property (VIP). The UVM Deployment Flow allows the surgical replacement of areas of UVM weaknesses with new, more powerful, approaches operating on existing UVM testbenches.

This paper will show how Breker tools, using the PSS, can produce a model that greatly simplifies complex sequence development, together with associated scoreboard checks and coverage models, leading to automated coverage closure. It then allows these tests to be reused in full chip simulation and emulation, along with post-silicon validation.

2. Introducing Portable Stimulus

The Portable Stimulus Standard (PSS) was first released by Accellera in 2018 and defines a new verification model that encapsulates both control- and data-flow aspects of an IP or SoC. This standard was built, in part, on foundational work refined by Breker over a number of years and relies on the solid mathematics of graph-based approaches.

The PSS model is read by a tool that synthesizes test cases to target verification environments including simulation, emulation, FPGA prototyping and real silicon. The tool employs randomization to pick the scenarios that will be synthesized. In
many PSS tools the end-user has to manually deploy generated tests into a specific environment by writing additional code that details these tests. Breker provides a range of automated deployment options that alleviate this manual coding by synthesizing this C or SystemVerilog code using additional services, see figure 1.

![Diagram](image)

**Figure 1: The Breker Tool Flow including the PSS Tool and Deployment Options**

A PSS model can be written in two languages that are semantically equivalent to each other. One of the languages is called the domain specific language (DSL), while the other utilizes C++ and an associated library. The DSL is somewhat similar to SystemVerilog and may be the preferred language for engineers who are used to working in a UVM environment. If they wish to replace UVM sequences with Portable Stimulus, as demonstrated in this white paper, they may find this language to be more familiar. C++ may be the preferred language for system architects, software engineers, chip bring-up teams and others who already use it extensively in their jobs today. The two languages are semantically equivalent, and with the Breker tools can be freely intermixed.

### 3. The Benefits of Applying PSS to UVM Testbenches

Figure 2 shows the complete UVM-PSS environment, driven by a scenario model written in PSS and utilizing existing UVM components. Note that to provide real productivity without substantial manual coding, a portable stimulus synthesis tool such as TrekGen™, needs to be accompanied by a Deployment Optimization Flow (such as Breker’s TrekUVM™) that includes UVM sequence generation, automated
Taking the Pain Out of UVM

scoreboard and coverage modeling, and other capabilities that provide a seamless connection to the UVM environment.

This flow operates as follows. A scenario model is created using the PSS DSL or C++ language, or graphically via TrekDesigner™. The model may be a single, small scenario, a complete specification or something in between. A common way that these models evolve is for scenarios to be individually created and then combined, something that is easy to do via this graph-based approach.

TrekGen and TrekUVM parse this model and run the first pass PSS test synthesis operation, solving random decisions as with any PSS tool and uniquely synthesizing the SystemVerilog tests, scheduling multiple tests together given the I/O, registers, etc., contained in the design. Path constraints on the scenario model allow for an extra level of configuration over and above the PSS standard node-based algebraic constraints.

A first pass coverage, reachability and other analysis can be performed on the tests to ensure they will be good enough for reasonable functional coverage, prior to simulation execution. The simulator is executed with TrekUVM either running interactively, or in a post-processing batch mode. The scoreboard self-checks operate to indicate tests that either pass or fail, with this information displayed in TrekDebug™, and potentially passed to an integrated debugger, such as Verdi®. Coverage may be reanalyzed together with profiling information and other details, and then fed back into the scenario modeling process.

There are many reasons why a UVM team should consider integrating PSS capabilities into their testbench. Some of these enable immediate benefit while
others provide gains over time. We conducted a survey of some Breker customers using PSS with their UVM flows. Below are the survey findings of the advantages that can be obtained by adopting a UVM-PSS flow:

- **More efficient stimulus.** The coverage value of each UVM test must be assessed individually. Every UVM-PSS test is guaranteed to do something useful as it is driven from an intent specification, and the resulting coverage is known before the test is run. Fewer, efficient tests directly translate into faster regression times, maximizing hardware resource efficiency. In addition, complete tests can be generated ahead of time rather than having to regenerate the test during simulation. Both of these are especially important with emulators.

- **White-box versus black-box.** UVM is inherently a black box testing approach. UVM tests drive block inputs and the test author must figure out manually a path through the design to target internal areas of interest. PSS with the Breker tool suite takes a white box approach where the functional test scenario drives the appropriate UVM sequences, generated automatically to target specific design internals without the author having to define a path from the inputs.

- **More complex tests.** One of the difficulties with the UVM methodology is that each VIP is independent, making it difficult to coordinate activity within the design and across multiple inputs. This is often necessary to ensure aspects of the design are stressed. A UVM-PSS strategy also extends the notion of random test generation to include available concurrency within the design. One single scenario can automatically generate multiple sets of synchronized sequences across all the block I/O ports through multiple agents, eliminating the manual need to come up with those synchronization points.

- **Integrated scoreboard and checkers.** Verification doesn’t happen unless a check has been made. Knowing that all aspects of a test have been verified is difficult when stimulus, scoreboards and checkers are not unified. With PSS, everything is integrated, resulting in faster and more complete model development that includes stimulus and checks. TrekUVM is able to generate self-checking tests automatically from the scenario, saving a considerable amount of painful coding.

- **Integrated coverage.** The development of coverage models is time consuming and difficult in UVM, partly because functional coverage of the implementation is a proxy for design intent coverage. A PSS graph contains a direct measure of design intent coverage, in that all paths through the graph are inherent capabilities of the device. For many blocks it becomes possible to think about 100% intent coverage obtained from automatically generated test sets. True functional intent coverage can be defined directly on the graph using path coverage (with Breker), ensuring that exactly the right functional cross combinations and are covered, and alleviating the need to create a separate
coverage model.

- **Path Constraints.** Path constraints allow the control of white-box sequences to target specific areas of the specification. These can be directly placed on the graph (and only with Breker, path constraints across the graph) to eliminate capabilities that are not to be verified or control the high-level randomization process.

- **Reuse throughout design flow.** Models developed for block-level verification are directly reusable for sub-system or system-level verification. This enables a bottoms-up verification methodology to be put in place. At the same time, a model developed by an architectural or performance team can be reused for block-level verification. This can save a significant amount of recoding, and creates the notion of higher level, comprehensible VIP.

- **Retargeting.** The migration of tests between simulation and emulation is difficult. In fact, this was the primary motivation for the development of PSS. But the retargeting capabilities go further than this. A PSS model can be retargeted to virtual prototypes, physical prototypes and real silicon as well. This means that tasks that were once validation steps can be integrated into a verification flow, and regression suites can be run directly on first silicon when it comes from the fab.

- **Earlier verification.** The development of a UVM testbench takes a considerable amount of time. Development teams would benefit from starting verification sooner. With a UVM-PSS methodology, verification can start as soon as a single path can be found through a graph and UVCs are in place. Details can be added to the verification model over time, meaning that the important aspects of a system can be verified early, and less important capabilities left until later in the cycle. Vendors can also help by providing libraries of standard functions and targeted apps for quick-start, such as the ARM cache coherence app.

### 4. Defining Sequences

While SystemVerilog and UVM start with the inputs on which stimulus is to be applied, Portable Stimulus starts with the intended goals of verification. For example, you may want to ensure that an image can be displayed on a screen. SystemVerilog would just randomly wiggle input pins and if an image were to be displayed on the screen it would be pure chance. A solver for a PSS model walks backwards through the graph finding out how to satisfy a declared goal.

A diagram for a hypothetical model is shown in figure 3. To accomplish the goal, the display needs a certain amount of data, in one of several formats. Where does that data come from? This defines an action, a dependence, and some information that
can be randomized. The display dependence could be fulfilled by getting data from the image processor, from a camera or read from an SD card. These in turn have dependences.

![Diagram](image)

**Figure 3: Actions and dependences within an image sub-system.**

Once all of the dependencies can be met by the primary inputs of the device, test scenarios can be created. It may be possible that concurrently, another image is stored to the SD card, coming from either the image processor or from the camera. A tool that generates test cases is free to choose any legal paths through the dependence graph and create the necessary stimulus to ensure that the verification goals are met.

![Diagram](image)

**Figure 4: Scheduling a Test with Shared Resources.**
Generated scenarios may be constrained by available resources, such as the number of CPUs, constraints about simultaneous SD card reads and writes, or the ability to do encode and decode simultaneously. The test may schedule when each application scenario gains access to necessary resources. Figure 4 shows how tests may be put together over one thread and three threads.

An important philosophical change is that verification changes from being a black box test methodology to white box testing. A UVM-PSS testbench knows what should be happening in the design and can check that each module is functioning as intended. Contrast this to a pure UVM testbench where scoreboards and checkers have highly limited visibility to design internals and problems are usually found a long temporal distance from the real problem.

Once a scenario has been defined, it can also be retargeted towards different execution environments. Thus it is possible to fully develop bring-up tests using an emulator and retarget the same tests to chips when they come back from the fab.

Figure 5 shows how these ideas are translated directly to a UVM design. This real example shows a multi-block design with four I2C ports, all driven by an independent I2C agent. The top-level graph is also included. This single graph will be walked to generate strings of sequences, which are then brought together using scheduling synthesis such that multiple tests are executed concurrently. The synchronization points between the sequence streams are shown as well.

Figure 5 – A single graph translated to concurrent sequence streams (courtesy ADI)
5. Inclusive Checks, Scoreboards & Debug

The scenario defined in the PSS graph model includes verification goals, and these are easily translated to UVM checks and a scoreboard. When a test is run, it is fully self-checking because it contains not only the stimulus but the expected result as well. As such, TrekUVM is able to synthesize a scoreboard and perform comparisons as each test is executed.

The ability to include checks in the scenario graph and see these translated down to UVM alleviates a significant burden on a verification engineer to independently come up with a complete scoreboard and reference model for checking. By making all the tests self-checking, the possibility of missing a check is greatly reduced and the burden of writing these low-level checks is eliminated.

Individual checks are brought down to the granularity of the related individual tests, which makes tracking test failures and debug easier. Figure 6 shows the TrekDebug testmap view where passing tests are shown in green and failing tests, where the checks indicated a scoreboard mismatch, shown in red.

![Figure 6 – Testmap view showing failing tests (red) and integration with Verdi](image)

At this level it is easy to see which tests failed and the debug window also includes high-level information to indicate possible issues. The debugger is integrated with standard signal level debuggers such as Synopsys’ Verdi. Failing test information can be directly used in the signal debugger to track down the problem.

6. Coverage Analysis, Automation and Profiling

UVM functional coverage analysis involves defining a coverage model that indicates if the constrained random tests have hit all the important functionality under test.
This task is somewhat independent of the stimulus and checker authoring efforts as might be expected. It is also a laborious, error-prone and time-consuming effort, where the engineer must attempt to create SystemVerilog covergroups, etc. based on the original specification and the design.

Indeed, functional coverage is often augmented with other forms of coverage, including code and mutation coverage, which inspects the implementation of the design itself in an effort to understand if the entire code base has been fully exercised during verification. This thinking suggests that if the entire design has been tested, then it is likely that the functionality it represents has also been fully exercised. Of course this is a questionable assumption.

It should be noted that all these coverage forms may still be run even if a PSS model has been used to produce the testbench. However, PSS enables a more effective coverage mechanism that may reduce the need to reply on these other methods, and in fact can eliminate the need for a UVM coverage model all together.

![Figure 7 – Examples of Cross Coverage and Path Coverage on a scenario model](image)

In a full PSS scenario graph model we attempt to define a verification intent specification for the functionality of the design. At this level of abstraction, it is easier to comprehend the entire functional specification and make it as complete as possible. Once this graph has been created functional, or scenario/intent, coverage may be obtained simply by ensuring that all the paths on the graph have been fully explored.
As the graph becomes large, this would result in a high number of tests, so instead provision has been made to allow coverage points to be established along with coverage targets (equivalent of SystemVerilog covergroups) simply by labeling the graph for key combinations of actions, and path coverage where specific points along a path may be defined as important to hit. Figure 7 on the right shows a cross coverage specification where all the ports in A need to be hit, followed by all the ports in B, followed by a cross coverage between A and B, resulting in 8 individual paths through the graph. On the left is a path coverage definition ensuring an overflow on port 0 with an error condition on large packets is fully tested.

This is a far simpler process than defining an independent coverage model, and leads to the ability to test the coverage on generated tests even before they have been synthesized from the scenario model. The graph is translated to a coverage specification, which is then checked during verification. This generates a list showing the coverage achieved and areas on the graph still uncovered, see figure 8, that may be downloaded into a verification manager tool or a spreadsheet as required. It also highlights the coverage deficiency on the graph.

![Figure 7 - Cross coverage specification](image)

![Figure 8 - List of coverage points, as well as highlighted missed coverage path](image)

This capability can lead to other coverage and profiling benefits, not available in regular UVM. For example, coverage closure may be easily automated where a coverage hole on the list is indicated, and a new test added from walking the appropriate path indicated by the identified hole. Reachability analysis may be performed prior to verification to make sure that all the nodes in the graph can be reached during test synthesis.

Profiling can also be performed after verification, where the size of the test is adjusted to reflect the number of clock cycles that the test required. When multiple
tests are scheduled together, leveraging common resources, etc., bottlenecks may be directly observed in terms of resource overload, etc., see figure 9.

Figure 9 – Post-execution profiling using superimposed test lengths on the test map

7. Connecting and Running the Environments

There are multiple ways in which the Trek tools can run, depending on the appropriate deployment flow. For example, if the solution is driving a full SoC test using software running on processors in a “Software Driven Verification” flow, C-tests and transactions will be synthesized in advanced of the verification run, which may be on a simulator or emulator. Results, including checks, coverage, profiling & debug information are extracted and may be analyzed independent of the verification tool execution.

When verifying a design that does not contain a processor, or the user chooses to run in this manner as is typical of most UVM block-level verification runs, the test is driven from outside of the design at the transaction-level. This is known as the “headless” mode, suggesting no processor is available to drive the system, see figure 10.

There are still two possible ways in which the test can be generated. One is the interactive generation mode, where the test generator is running in lockstep with the simulator. This is similar to the way in which SystemVerilog and UVM test cases typically run and enables test generation to be dependent on design state. It is also possible to pre-generate a complete test case, which is then loaded into Trek ahead of time and it is not necessary to run the solver in parallel. Result data is still captured for post run checking. This is more efficient and saves simulation time.
There are three fundamental connection points between the two environments. Establishing the channels between the environments, sharing the memory map, including register field designations and then synchronization between the two environments. In addition, the resources available to a test need to be defined. This may include the number of threads and available memory. It should be noted that aspects of this are TrekUVM specific. This cannot be avoided because you are integrating a UVM model to a tool rather than an interface between two models. It also makes use of the hardware/software interface (HSI) to provide a set of services, which alleviate the user from having to perform integration tasks.

8. A Portable Stimulus UVM Example
The simple design used in this example includes a couple of CPUs, two UARTs, a DMAC and an AES block, as depicted in Figure 11. The example is a public domain code base and a full distribution of it is included in every Breker software installation, enabling the example to be tried out.

As previously noted, the Accellera PSS makes use of two formats, the Domain Specific Language (DSL) and a C++ class library. These two formats are semantically equivalent. With Breker a scenario may be written in either format, or using both formats mixed together. Further more, full C++ functions can be called from DSL code allowing the use of procedural coding. In this example we have chosen to predominantly use the DSL format.

Six steps are required to integrate TrekUVM PSS tests into an existing UVM testbench and execute it.
1. Identify UVM interfaces, including TLM interfaces, software interfaces and memory. Configure the tool and integrate to UVM.
2. Create PSS register type descriptions. This can be done manually, through HSI register definitions, or converted from an IP-XACT description.
3. Identify the overall PSS model/representation for the design (components, actions, resources, etc.)
4. Provide details for each ‘action’. These are defined in terms of portable primitives that can synthesize to either TLM or SDV tests.
5. Compile the model, synthesize test cases and run UVM simulation.
6. View and debug results, and analyze coverage.

**Figure 11: The Example Design**

**Step 1. Identify UVM interfaces and integrate tool**

The example design has two UART VIPs, each of which need TLM blocks for configuration and for transmitting and receiving data. The design also has two CPU ports, which are driven by APB VIPs.

TLM transactions and TLM ports are defined for the UART VIPs and processor agents configured in the TLM mode are defined for the APB VIPs. A memory resource is defined for use by the DMAC operations.
Taking the Pain Out of UVM

// transaction to configure the UART VIP to match UART DUT
class uart_config : public tlm_transaction {
PSS_CTOR(uart_config, tlm_transaction);
  tlm_field<bit> char_length ("char_length");
  tlm_field<bit> nbstop ("nbstop");
  tlm_field<bit> parity_en ("parity_en");
  tlm_field<bit> parity_mode ("parity_mode");
};

// UART payload transactions for both Rx and Tx
class uart_frame : public tlm_transaction {
PSS_CTOR(uart_frame, tlm_transaction);
  tlm_field<bitset> payload ("payload");
};

// instantiate Portable TLM ports
put_port <uart config> cfg_port {"cfg_port", tb_path + "cfg"};
put_port <uart frame> tx_port {"tx_port", tb_path + "tx"};
check_port <uart frame> rx_port {"rx_port", tb_path + "rx"};

// Instantiate processor agent in TLM mode
Processor cpu0("cpu0", 2, Processor::TLM);
Processor cpu1("cpu1", 2, Processor::TLM);

// Memory resource for use by DMAC
memory_resource ddr0 ("ddr0", 0x4000);
A DPI file called `libtrek.so` is also linked into the simulator. This DPI library is dormant unless the `+TREK_TBX_FILE` flag is set, as show below.

**Step 2. Create or convert register descriptions into an HSI PSS register description**

Most blocks contain descriptions for internal memory and registers. These definitions may be listed in an IP-XACT file or another format, or created by the test author. IP-XACT is a commonly used XML format that defines and describes individual, re-usable electronic circuit designs to facilitate their use in creating integrated circuits, and this example has an IP-XACT definition available. Breker provides a utility (`trekhsi`) that converts an IP-XACT register representation into an HSI register definition.

Register descriptions for each of the three components (UART, DMAC, AES) are easily created using `trekhsi` from the IP-XACT files released with the design. Field names can be modified for readability.

The HSI register definition for the UART (`hsi_uart.h`) becomes an ‘`hsi::reg_block`’. The field names were modified from the raw IP-XACT specification for readability.

```cpp
// Register Block
class uart_regs : public hsi::reg_block {
public:
  uart_regs(
    const pss::scope& name,
    hsi::reg_addr base
  ) : hsi::reg_block (this)
  , map("map", base)
  {
    map.add_reg { UART_RX, 0x0000000000000000 };
    map.add_reg { UART_RX, 0x0000000000000000 };
    ...
    map.add_reg { UART_DIVISOR_MSB, 0x0000000000000001 };
  }

  reg_UART_RX UART_RX { "UART_RX" };
  reg_UART_TX UART_TX { "UART_TX" };
  ...
  reg_UART_DIVISOR_MSB UART_DIVISOR_MSB { "UART_DIVISOR_MSB" };

  hsi::reg_map _map;
};
```

**Step 3. Identify the overall PSS model/representation for the design**

Review the design to identify the components of the system. For this design the main IP blocks are the UARTs, DMA, and AES which each become a ‘PSS component’. Each block has core functionality that can be described as ‘actions’ and are represented as a ‘PSS action’. The key functions (actions) of these blocks could be
defined as:

- UART – configuration, receive, transmit
- DMAC – output data, input data
- AES – encrypt, decrypt
- CPU – output data, input data

Note that when first writing a PSS model, it is not important that all actions are defined. At first, only the most important need to be defined and, as the verification task progresses, additional, secondary actions may also be defined. This does not impact any of the verification already performed, it just makes more sequences possible.

A resource pool is created for each of the computational elements (UART, DMAC, AES).

The interfaces into the blocks are defined using flow objects (FIFO, Reg) and a corresponding ‘pool’ created for each.

Finally, the PSS locks control of the shared or exclusive use of resources. The scheduler will use these to ensure that it does not attempt to make the hardware perform mutually exclusive actions at the same time.

Breker’s TrekDesigner was used to create the model depicted in figure 12. In that model, ‘components’ are green boxes, ‘actions’ are light blue boxes, ‘resources’ are dark blue diamonds, and ‘locks’ are grey boxes associated with an ‘action’. Inputs and outputs to ‘action’ blocks are depicted with blue input/output ports.

TrekDesigner is a graphical user interface (based on UML) that supports PSS 1.0, the current version released by Accellera. TrekDesigner may be used to quickly create or extend PSS models to define and test design intent. The tool automatically generates PSS code from the graphical representation input by the user. The model could also have been entered textually using PSS. The Trek tools can also be used to visualize a design that was entered using PSS.

The Entry action (top) will schedule two UART scenarios, an Encrypt and a Decrypt operation, concurrently. The UART scenario (bottom left) will pick a configuration for the DUT, configure the VIP to match and do a number of receive and transmit operations in parallel. The Encrypt and Decrypt operations are fed by DMAC transfers (bottom right). Resource locks are used to ensure that two operations on the same hardware block are not allowed to execute at the same time.

PSS code for the entire model is generated by the tool. Each generated action has a pair of // Start of user code and // End of user code markers within which the detail operation of the action may be entered. Code within these markers are preserved when the model is regenerated.
Step 4: Provide details for each ‘action’ -- regardless of the type (TLM or SDV)

Each IP block or ‘component’ specifies its corresponding functionality in an ‘action’. Unique to Breker is the to synthesize stimulus and checks for each action regardless of the transaction type, namely whether the design is TLM or SDV.

Consider the UART transmit operation in figure 13. It is hierarchically broken into three steps as shown below.
Detailed code to drive transmit data to the UART DUT is shown below. The length of the transfer is randomized between zero and 16 bytes. Note that the input data is forwarded to the next consumer action.

```vhln
testbench 
#ifdef __DATASRC_H
#define __DATASRC_H

#include "fifo.h"

action DataSrc 
output Fifo out0;
// Start of user code Action_DataSrc
rand bit [4] count;
rand bit [8] data;

exec body 
for (bit i = 0; i < count.val(); ++i){
    iwrite8(data.val(), out0->addr);
    out0->expect.push(data.val());
data.solve(); // get new randomized values
}
// End of user code
};
#endif // __DATASRC_H
```

Detailed code, to check that the correct data is received by the UART VIP, is shown below. Note that results are checked against expected data from the producer to this action.

```vhln
testbench 
#ifdef __UARTVIPCHECK_H
#define __UARTVIPCHECK_H

#include "fifo.h"

action UartVipCheck 
input Fifo in0;
// Start of user code Action_UartVipCheck
ref uart_block blk;

exec body 
while(!in0->expect.empty()){ 
    uart_frame rx "{rx}";
    rx.payload.set(in0->expect.front());
    blk->rx_port.check( rx ); // blocking
    in0->expect.pop();
}
// End of user code
};
#endif // __UARTVIPCHECK_H
```

Code for the UART scenario is automatically generated (below):
The top-level entry code is also generated (below):

```c
#include "uartCfg.h"
#include "uartVipCfg.h"
#include "uartDoTx.h"
#include "uartDoRx.h"

action UartScenario {
    Uartcfg uartCfg0;
    UartVipcfg uartVipCfg;
    UartDoTx uartDoTx;
    UartDoRx uartDoRx;

    activity {
        uartCfg0;
        uartVipCfg;
        parallel {
            uartDoTx;
            uartDoRx;
        }
    }
};
```

**Step 5: Compile mode, synthesize test cases and simulate UVM**

The following two lines compile the model and synthesize a test into the file `test_uvm3.tbx`:

```bash
% $HOME/bin/trekcc test_uvm3.pss -o test_uvm3.exe -src
% ./test_uvm3.exe --seed=0x1234 --entry=gss_top.entry --output test_uvm3.tbx
```

The UVM simulation, as in figure 14, is run as normal with the addition of a flag `+TREK_TBX_FILE=test_uvm3.tbx`. The flag causes the TrekBox runtime component to start and read the `test_uvm3.tbx` file. Transactions are sent and received as required to the various UVM VIPs to drive stimulus and check results.
Step 6: View and Debug Results, Check Coverage

As shown in figure 15, Breker’s TrekDebug provides visualization of the multi-threaded sequences generated by TrekUVM (or TrekSoC™ and TrekSoC-Si™). TrekDebug has been integrated with standard debug environments, such as the Synopsys Verdi debugger. This tight integration with well-known industry simulation environments provides useful capabilities, such as cross probing, breakpoints, etc.

With TrekDebug, you have the ability to see and trace the activity within the tests. This includes the tracing of all the processors and threads, the memory regions (contents of all locations), and the tracing of activity, including dependency across time between processors and threads.

This run shows the UART being configured and then a number of UART operations being performed. Green boxes are operations that completed successfully, while the red boxes indicate that an error occurred. In this case, a mismatch between a payload on a Tx operation and the expected outcome was found. The blue boxes are actions that have not yet taken place.
There are many coverage and profiling operations that could be performed on this example. Note in future white papers we will detail extensively the many coverage options available to users of the tool. In this case we will perform a simple coverage analysis that can be displayed at the touch of a button. This analysis shows that several paths in the graph were not fully exercised in this simulation, as depicted in both the graph and list views shown in figure 16. This list view may be downloaded into a standard CSV file, a spreadsheet, and several verification manager solutions available on the market, thereby combining this coverage results with others.
From this view an auto-coverage closure option may be selected to close specifically identified paths, where additional tests will be generated to create this coverage.

Path constraints and path coverage may also be defined to specifically identify areas in the graph to be covered, and this could also be highlighted on the coverage view. Analysis may be performed that includes reachability across the graph and expected coverage prior to the tests even being executed. Future papers will explain in detail the coverage options available with PSS and the Breker Trek tool suite, or for more information, please contact the company.

Profiling may also be executed where the length of the test in executed clock cycles is superimposed on the test map view, as shown in figure 17. Here you can clearly see the same tests either lengthened or reduced in size. Combined with the configurable scheduling synthesis to schedule multiple tests with their dependencies, it is possible to stress test parts of the design to receive an accurate measure of potential resource bottlenecks or timing and power issues. In this example a critical path in the profiler is displayed across the sequence threads.

![Figure 17: Profiling the UART design](image)

9. Summary
In this whitepaper, we demonstrated how a UVM environment can be upgraded using the new Portable Stimulus Standard. A PSS model is a unified description that brings together stimulus, checkers, scoreboards, coverage and constraints to enable simple and efficient coverage closure. While this example focused on the generation and checking of stimulus, future whitepapers will talk about how coverage is
integrated into the same graph used for generation, as well as addressing portability, scalability and adapting existing UVM methodologies to maximize the increased capabilities made possible by the PSS integration.

In this paper we attempted to show the overall benefits that may be derived from using a single, easy-to-understand scenario model to synthesize complex, multithread synchronized sequence sets, together with checks and coverage analysis. We showed how PSS has some inherent advantages over UVM, which allow for an easier, white box modeling style, and a shift in abstraction level to focus on the intent specification of the verification program. Increasing value comes with complex systems where scheduling and resource allocation can make the construction of testcases very difficult and tedious.

It should also be evident that these modular PSS models may be easily ported to other phases of the verification process, in particular Software Driven Verification (SDV) with a simulator or emulator, and Post Silicon Validation. Future white papers will also address these use models.

Although the PSS allows for an appropriate level of abstraction in creating these models, much of the benefit of this approach lies in the tooling used to synthesize the tests, schedule them correctly, and then provide portable services which eliminates hand coding in the tests to integrate them into the UVM environment. This also includes key verification activities such as coverage analysis and closure, debug and profiling. Breker’s experience working with UVM teams has allowed us to build a set of tools that offloads the manual effort associated with using the PSS standard in its raw form.

For further information or to see the PSS tools in action, please contact Breker Verification Systems at our website. Also, look out for future white papers cover other flows and focused details such as coverage analysis and closure.