Portable Stimuli Over UVM

Using Portable stimuli in HW verification flow

Efrat Shneydor, Cadence Design Systems, Israel *(efrat@cadence.com)*
Slava Salnikov, Ben Gurion University & Texas Instruments, Israel *(slava.s@ti.com)*
Liran Kosovizer, Texas Instruments, Israel *(lirank@ti.com)*
Dr' Shlomo Greenberg, Ben Gurion University, Israel

Keywords—Verification; PSS; Portable Stimuli; UVM; Specman

I. INTRODUCTION

A. Overview

In the Hardware verification world, we see continuous increase in the size and complexity of designs under test. The complex designs present new challenges to the verification engineers, and especially to the integrators and tests writers. Two of the main challenges are creating a multi-channel stimulus which requires synchronization of multiple sub-systems, and ensuring that test plans were achieved. In this paper we will present a project we did, in which we used *Portable Stimulus and testing Standard* (PSS) to write tests that run over a HW verification environment implemented in UVM. The verification environment that we used is implemented in *e* and the PSS model is implemented in SLN. For generating the model we use Cadence Perspec™ System Verifier.

B. The Device Under Test

The device under test is a WI-FI component, including multiple CPU cores, power domains & HW hierarchies. To model WI-FI flows, a PSS state machine was created, capturing rules of select frame exchange sequences such as connection establishment & optimization, data transmission, and power save scenarios. In essence, the PSS model emulates interaction with a WLAN router. In this paper, we will show examples from the connection establishment phase of the state machine (figure 1).

Figure 1: The connection establishment state machine

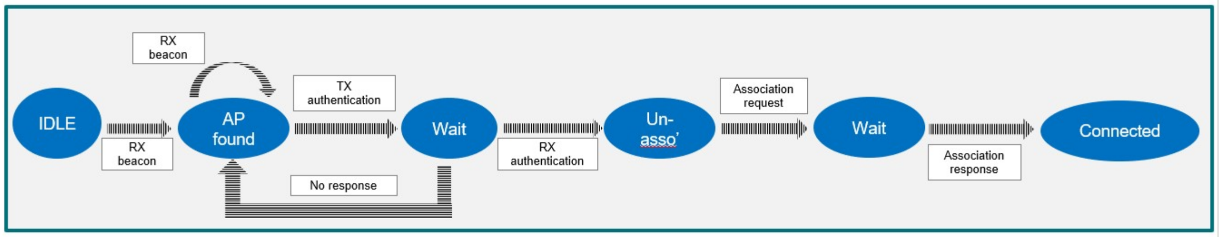


Table 1: Terms and Acronyms

Acronym	Stands for	Description
e	e hardware verification language	An aspect oriented verification language, used by Cadence® Specman simulator. Contains constraints solver, temporals language, coverage model, and also implementation of UVM methodology (sequences, objections, and more)

DUT	Device Under Test	In hardware verification, this refers to the design that is being tested.
FLI	Function Level Interface	Specman *e*-C function level interface. Enabling to call *e* functions from C code, and to call C functions from *e* code.
PSS	Portable Test and Stimulus	Accellera standard of a portable test and stimulus specification language, that can be used to generate stimulus for multiple target implementations.
SLN	System-Level Notation	One of the Portable Stimulus Syntax languages that are supported by Perspec™ System Verifier. Was developed by Cadence® Design Systems, Inc.
TI	Texas Instruments	A technology company that designs and manufactures semiconductors and various integrated circuits. The headquarters are in Dallas, Texas, United States, and the company has dozens of offices, including Europe, Israel, India, Russia, and more.
UVC	Universal Verification Component	A reusable verification component, which follows the Universal Verification Methodology. .It contains the definition of stimulus generation, monitoring, checking and coverage model that are needed to verify a design component, interface or protocols.
UVM	Universal Verification Methodology:	A standardized methodology for verifying integrated circuit designs. An Accellera standard with support from multiple vendors
WI-FI	Wireless Fidelity	wireless networking technology that uses radio waves to provide wireless high- speed Internet and network connections. commonly used for the wireless local area networking (WLAN) of device
WLAN	Wireless Local Area Network	Wireless computer network that links two or more devices using wireless communication. Most modern WLANs are based on IEEE 802.11 standards and are marketed under the Wi-Fi brand name.

II. Test Generation

A. UVM Sequences vs. PSS actions

Using UVM, the test is described with **sequence**s. UVM sequences give the capabilities of constraints solving and smart scheduling done by the sequencer – multiple sequences can run in parallel, sequences can have priorities and grab the sequencer, and more advanced capabilities. But implementing system level scenarios using UVM virtual sequences can be challenging. The core reason for that is that some of the sequence characteristics are controlled declarative, using constraints. Other characteristics, mainly timing, are controlled procedurally in the sequence body. Relations between sequences are controlled using both constraints and actions, and defining dependencies between various sequences is one of the biggest challenges in tests writing. For implementing the connection establishment (shown in figure 1), for example, there should be a sequence that sends an authentication packet to the device. But to model the packet arrival in the right context (access point with supported parameters found, an authentication frame sent to it), and more importantly – correct handling if something goes wrong along the way (retry, maybe look for another access point), a UVM test writer must work very hard.

In the end of 2018, TI started designing the verification environment for the next project, a WI-FI component including multiple CPU cores and power domains. The verification environment is based on existing UVCs, and for the new project it was required to implement a virtual sequences library – sequences running n top of the existing sequences, defining system level scenarios. The team decided to look for advanced solutions for defining system level scenarios, rather than implementing complex virtual sequences. Another motivation for looking for new tools and methodologies was the plan to integrate the environment with SW tests, in the future.

For the new and improved verification environment, TI decide to evaluate the usage of portable stimulus tests (PSS). As opposed to UVM nature of describing test **steps** ("do this, wait for something, then write that"), the PSS actions talk about **states** and relations between them. SLN is one of the PSS languages supported by Perspec, and it resembles *e* – Specman language. As this verification team works with Specman, SLN was the natural choice for this project. With SLN, the basic building blocks are **actions**, and they exchange **flow objects** between them. The flow objects are the mean to define the action input (action precondition) and output (state assumed after this action is taken). Using them, dependencies between actions are straightforward. The tool generates a test, which is practically a list of actions, in which the output of each action is the input of the next action. (The code example below uses the old syntax of using 'tokens').

This is how we describe an action that takes from the *ap_found* state to *authentication* state, with SLN:

code example 1: SLN: Defining an action

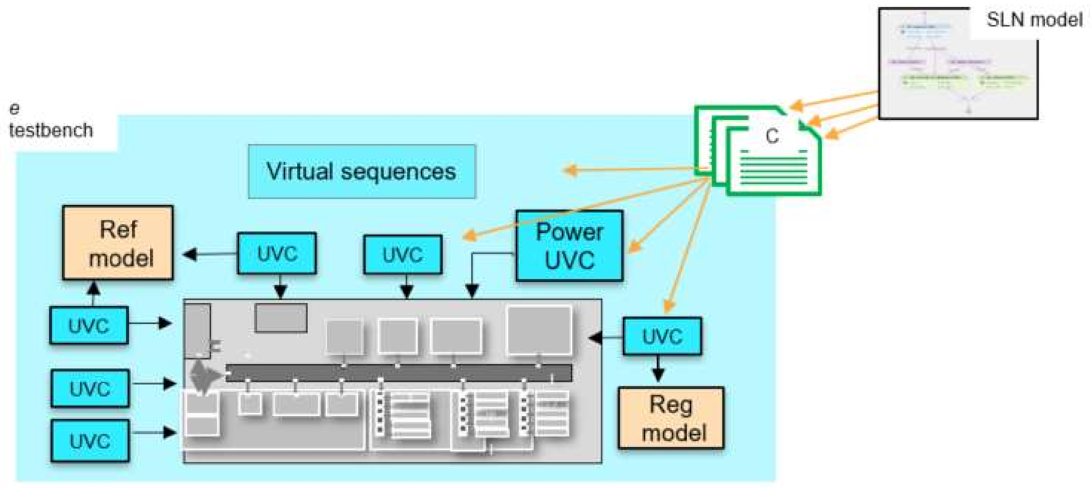
```
// The protocol/DUT state machine
type env_state_e : [idle, ap_found, wait_auth] //… and more..

// the token
state token env_state_t {
    state: env_state_e;
};
component env_component {
   // the token used by all actions
   child state_var: memory[1] of env_state_t;

   // Define action for each state transition
   action tx_auth {
      input prev  : from state_var;
      output next : to state_var;

      // define the input and output of this action
      constraint prev.state == ap_found;
      constraint next.state == wait_auth;
   };
};
```

When the user will request to generate a *tx_auth* action, Perspec will first generate an action that its output state is *ap_found.*

B. *Combining Sequences and Actions*

We decided to employ the advantages of both methodologies. The PSS model defines the actions – the dependencies and order between states, and the steps that must be taken for moving from state to state. This model is Transaction Level Model – it describes the high level of the transaction. The UVM sequences, implemented in *e*, generate the bit level data and sends it the DUT. We can view this as PSS being a layer on top of the traditional UVM environment, as shown in figure 3. Based on the actions that are defined in the PSS model, Perspec creates C test files. These test files run on top of the *e* based testbench.

Figure 3: PSS over UVM test environment



For implementing this flow, we should decide of the separation of duties. What is to be decided and generated by the PSS model, and what – by the testbench. This is discussed in the section “The Action-to-Sequence flow”.

Then – we must implement the mechanism for C tests to activate the *e* based testbench, and this mechanism is described in the section "The languages layers".

1) The 'Action to Sequence' flow

We decided not to duplicate in PSS all the knowledge that is already implemented in the *e* verification components. The generation of data items and sequences will continue to be done in the *e* code. The PSS model should pass to the *e* component the directives of what data item should be generated. Such directives are, for example, the direction and the kind of the next transaction. As the testbench can send a burst of transactions, another directive is the number of transactions that the next burst should contain. The *e* function, when called by the C test, should generate the next item/s based on the directives it got. We defined a config struct, holding all the information that the PSS model passes to the testbench. Code example 2 shows the definition of this struct. SLN model can import *e* files, so this struct is defined once and used both by SLN and *e*. Note that in the future, when running tests on other platforms – other utilities will get the some struct and handle it according to the platform they run on.

code example 2: e: Defining the configuration struct

```
struct config_s {
    direction        : direction_e;
    // the kind of each transfer in the next burst
    transfer_kinds   : list of transfer_kind_e;
    ack_req          : ack_kind_e;
    // … additional fields
};
```

For executing the action defined in code exmaple 1, the model defines these constraints on the config struct. The **exec body**, the execution of this acrion, calls the *send_next* function:

code example 3: SLN: Constraining & driving the configuration struct

```
extend tx_auth {
    // TX one transfer, and its kind should be AUTHENTICATION
    constraint cfg.direction               == TX  ;
    constraint cfg.transfer_kinds.size() == 1;
    constraint cfg.transfer_kinds[0]      == AUTHENTICATION;

    exec body {
       // The function that executes the action
       send_next(cfg);
    };
};
```

The testbench method that executes the action, generates the required transaction, according to the config it got:

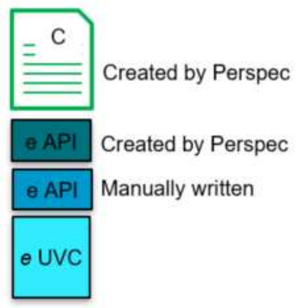
code example 4: e: Getting & handling the configuration struct

```
extend env_driver {
    send_next(config : config_s) @sys.any is {
        if config.direction == TX {
           for each in cfg.transfer_kinds {
               gen transaction keeping {
                   .kind   == it;
                   // All other fields are generated randomly, according
                   // to the constraints defined in the UVC.
               };
           };
           execute_item(transaction);
        } else {
           // in case of RX, do something else…
        };
    };
};
```

2) The languages layers

After Perspec generates the actions, and thus defines the test scenario, it creates the test file, which is C code. As the verification environment is implemented in *e*, we had to implement an API layer between the *e* UVC to the C tests. The API consists of a small number of methods exported to C using Specman Function Level Interface (FLI). Using FLI, *e* methods can be called from C functions. The header of the API is created automatically by Perspec, and must be extended by the verification engineer to add technical details, known only to the testbench developers.

Figure 4: The languages layers



For defining an *e* method named *send_next()* to pass the config struct from the C test to *e,* we wrote in the PSS model these two lines, defining the header of the imported function, and declaring that it is imported from *e*.

code example 5: SLN: Importing an e function

```
function send_next(config: config_s);
import E function send_next();
```

Based on this declaration, Perspec creates an *e* file, and in it writes this function header:

code example 6: e: Automatic created API, exporting an e function

```
unit perspec_thread {
    @export_fli_c()
    send_next(config: config_s) is empty;
};
```

The verification engineer must extend this method, per their understanding of the environment, calling the relevant method/s in the UVC:

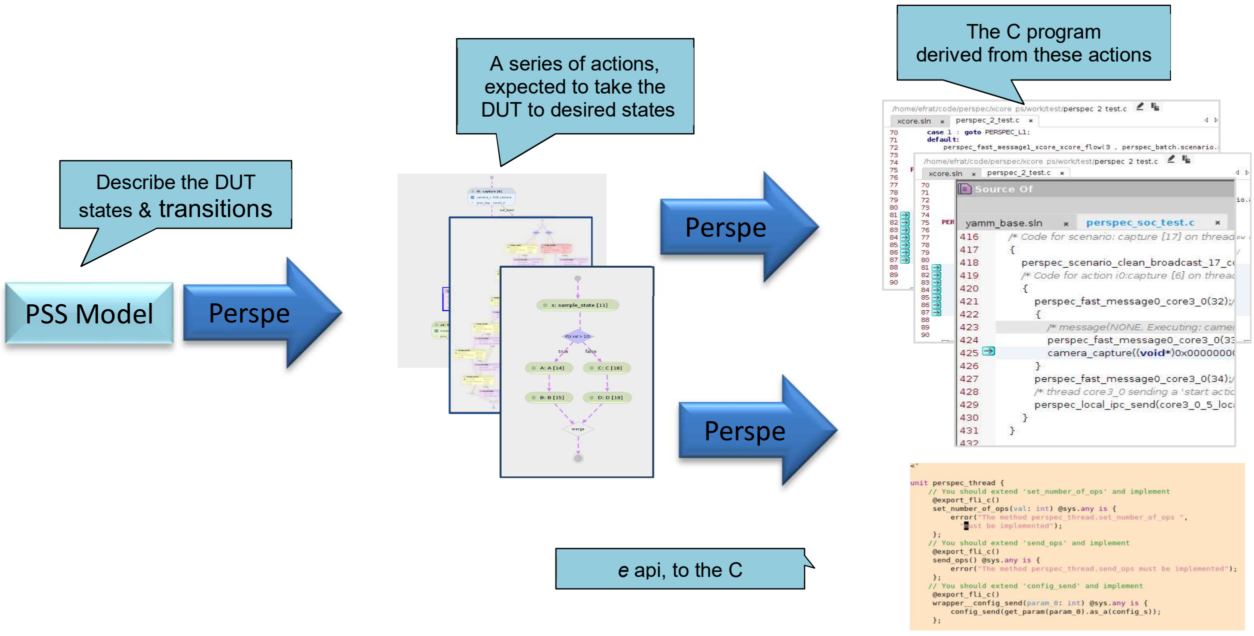
code example 7: e: Manually created API, implementing the exported function

```
extend perspec_thread {
    send_next(config : config_s)@sys.any is only {
        for each env_driver starting from sys {
            if it.available {
                it.send_next(config : config_s);
```

```
                    break;
                };
            };
        };
};
```

Figure 5 illustrates the flow – Perspec reads the PSS file and generates actions. If the user is happy with the generated scenario – they create the test, which consists of two files – a C file defining the test scenario, and an *e* file which is the interface to the UVC.

Figure 5: The test flow


Describe the DUT states & transitions
PSS Model
Perspe
A series of actions, expected to take the DUT to desired states
s: sample_state [11]
true
false
A: A [14]
C: C [18]
B: B [15]
D: D [19]
merge
Perspe
The C program derived from these actions
Perspe
e api, to the C

/home/efrat/code/perspec/xcore ps/work/test/perspec 2 test.c

xcore.sln perspec_2_test.c

```
        case 1 : goto PERSPEC_L1;
        default:
            perspec_fast_message1_xcore_xcore_flow(3 , perspec_batch.scenario.
```

Source Of

yamm_base.sln perspec_soc_test.c

```
    /* Code for scenario: capture [17] on thread
    {
        perspec_scenario_clean_broadcast_17_c
        /* Code for action i0:capture [6] on threa
        {
            perspec_fast_message0_core3_0(32);/
            {
                /* message(NONE, Executing: came
                perspec_fast_message0_core3_0(3
                camera_capture((void*)0x0000000
            }
            perspec_fast_message0_core3_0(34);/
            /* thread core3_0 sending a 'start actic
            perspec_local_ipc_send(core3_0_5_loc
        }
    }
```

```
<'

unit perspec_thread {
    // You should extend 'set_number_of_ops' and implement
     @export_fli_c()
     set_number_of_ops(val: int) @sys.any is {
         error("The method perspec_thread.set_number_of_ops ",
           "must be implemented");
     };
    // You should extend 'send_ops' and implement
     @export_fli_c()
     send_ops() @sys.any is {
         error("The method perspec_thread.send_ops must be implemented");
     };
    // You should extend 'config_send' and implement
     @export_fli_c()
     wrapper__config_send(param_0: int) @sys.any is {
         config_send(get_param(param_0).as_a(config_s));
     };
```

III. Coverage

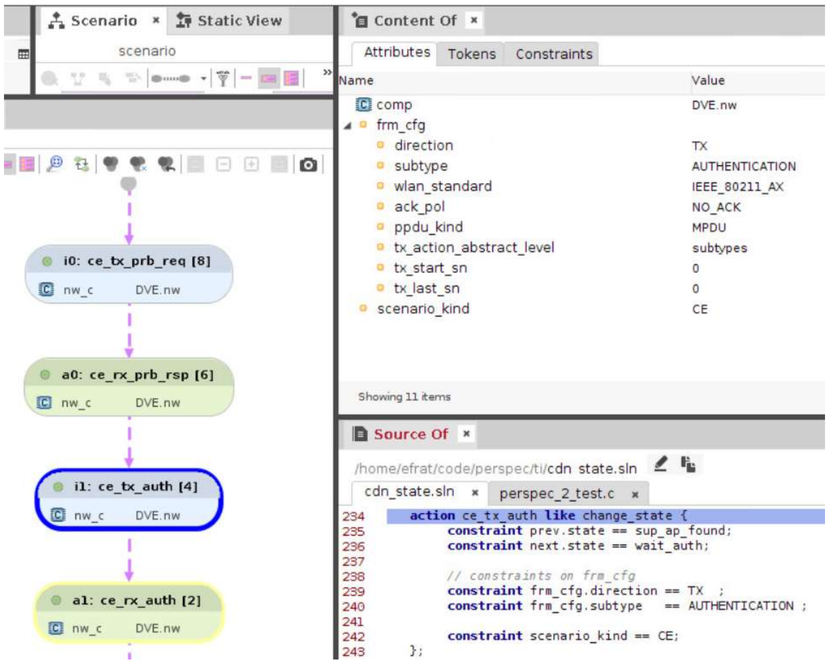
In addition to the coverage models defined in the UVC, we can define coverage of the PSS model. This coverage model consists of reached states (hence – 'internal coverage'), and passed flow objects (The *send_config* struct that is generated and sent to the UVC). There is some duplication between the two coverage models, but as the PSS model coverage is collected at gen-time (upon test scenario composition – doesn't need run time information), we gain the advantage of knowing expected coverage before the tests are run. This allows the user to adjust the test suite before running a single test, and running only after ensuring that coverage will be satisfactory. The coverage model of the UVC provides details that are not described in the PSS model (collected at run time).

IV. Debugging

One of the challenges we encountered was in debugging the tests, which requires a different approach and tool from what we use when debugging pure *e* verification environments. When debugging UVM environment, we can set break points and proceed with source debugger step by step, and issue **trace sequence** commands, and understand which sequence initiated the current transaction. Debugging a simulation in which the tests are implemented in C raised the question – do we need to debug the full flow, the C code as well as the *e* code? One approach is to use debugging tools that enable us track actions from C to *e*. This approach is challenging, as

the C code was created by Perspec, and much of it contains synchronization between threads. Lots of code that the average engineer is not familiar with (and is not expected to be).
The other approach is ”two folded debugging”: debug the top level using PSS point of view, that is – getting high level understanding of what the test is about. What states it is about to enter, what actions are about to be taken in this test, and why. Figure 6 is a screenshot from Perspec, showing the information of the generated test. One can see that this test will perform four actions (each can result with more than one *e* method or sequence, of course). For each of the actions, we can see its source code and the generated fields, mainly the values of the config struct.

Figure 6: A generated scenario, with four actions



Name	Value
comp	DVE.nw
frm_cfg	
direction	TX
subtype	AUTHENTICATION
wlan_standard	IEEE_80211_AX
ack_pol	NO_ACK
ppdu_kind	MPDU
tx_action_abstract_level	subtypes
tx_start_sn	0
tx_last_sn	0
scenario_kind	CE

```
    action ce_tx_auth like change_state {
        constraint prev.state == sup_ap_found;
        constraint next.state == wait_auth;

        // constraints on frm_cfg
        constraint frm_cfg.direction == TX  ;
        constraint frm_cfg.subtype   == AUTHENTICATION ;

        constraint scenario_kind == CE;
    };
```

This gives us a basic understanding of what the test is about to do, what kind of transfers are expected, which states the DUT is expected to enter.

V. Conclusions

Our expectations that implementing a PSS model would be more efficient than a sequence library were met. In less than a week, we modelled a simple, flexible set of actions that cover all modelled states, the code is much shorter than equivalent sequences, and once done – we could reach 100% coverage immediately, using Perspec ‘Fill’ option. Another nice advantage of this flow is that it greatly simplifies sharing the model with management, architects & designers. While most are not familiar with UVM sequences, all can understand a UML flow chart illustrating state transitions.
Defining the model takes some conceptual shift, and we needed the help of a Perspec expert. Without that – it would have taken us much more time to implement the model, and most likely it would be cumbersome, missing much of PSS advantages.
Connecting the outcome of the PSS model (the C tests) to the *e* verification environment went smoothly. Few changes had to be done, for example – connecting the C tests to the UVM Testflow. Some of them done as extensions to Perspec, in a loaded sln file, and some were done in *e* files. The plan is to further improve the support of testflow phases, to be more configurable.
The last and not least was changing the regression flow. The running scripts should compile the generated C tests, and we have to connect Perspec to vManager.