Portable Stimulus Models for C/SystemC, UVM and Emulation

Mike Andrews, Mentor Graphics: mike_andrews@mentor.com
Boris Hristov, Ciena: bhristov@ciena.com

Abstract- The methodology and standards for modeling stimulus generation and tracking functional coverage are currently not easily ported between the various verification environments that are used during the development of a SoC. The use of constrained random stimulus is mostly confined to SystemVerilog testbenches or hybrid environments such as mixed SystemC / SystemVerilog. While methodologies such as the SystemC Verification Library (SCV) do allow for randomization of stimulus and seed management, the limitations and the performance have been disappointing, and so they have not been widely adopted. The real issues though are the effort required to maintain independent stimulus models and the lack of random stability between those models.

There is therefore a growing interest in having a stimulus modeling capability that is independent of the language and methodology used. For example, when developing algorithms that will eventually be synthesized in a High-Level Synthesis (HLS) flow, a common desire is to fully verify the functionality in a pure C environment but with advanced capabilities like constrained random stimulus and functional coverage metrics included. Ideally the same stimulus model can then be re-used within a SystemVerilog testbench to validate the RTL output of the synthesis flow. Furthermore, if random stability can be achieved across the languages and testbench environments then the results from each model can be compared easily and problems found in one environment can be debugged on any or all of the available design models.

This paper describes a methodology based around a graph-based stimulus modeling language which can provide all of these capabilities. The same model can be used to generate repeatable stimulus in a pure C or SystemC simulation environment, in a SystemVerilog UVM testbench or any other environment, such as hardware emulation, that can be driven from transaction level stimulus generated on a connected host. The graph based model consists of a combination of rules that resemble a Backaus-Naur form and SystemVerilog-like algebraic constraints. The model can originate from many sources, including extraction from a System Verilog UVM sequence item, or generation from a high-level synthesis tool that has analyzed the design under test. This model can be compiled into a binary form that can drive repeatable constrained-random stimulus, controlled by common seeds, into any environment that has a C/C++ application-programming interface (API).

In addition, by extending the modeling capability of algebraic constraints typically used within a SystemVerilog randomize-able class object with the rule language syntax, more complex verification scenarios can be described in the graph-based model compared to, for example, a SystemVerilog sequence item, without compromising the plug-and-play nature or reusability of that model.

I. INTRODUCTION

With the capabilities of mixed language simulation, and the availability of libraries dedicated to inter-language or inter-methodology adaptation/translation, it may seem that common stimulus models could already be made available in most environments. This may be true in theory, but connecting, for example, a SystemVerilog constrained random stimulus model to an abstract algorithmic model in C or C++ brings an unfamiliar language and possibly performance issues into that domain. Other options like manual or automated translation of a stimulus model from one language to another bring their own issues in the areas of maintenance overhead and debugging issues in the translated models.

At the same time, those domains now need to upgrade their verification methodology with the capabilities of a SystemVerilog UVM based environment in order to keep up with the growing complexity. The reasons are identical to those that drove the evolution of constrained-random, coverage-driven environments for SystemVerilog, i.e. primarily the need for automation in the generation of stimulus, when directed test generation can no longer keep up.

II. ALGORITHM VERIFICATION IN C/C++

A pure C-based verification environment is traditionally used for checking the algorithm performance and for comparing floating vs. fixed point models. The algorithm's relative performance due to varying design parameters and other trade-offs can be determined in this early architecture phase without the need to re-verify them in RTL, providing that equivalence between the original C and subsequent RTL implementations can be established. This is done either formally or by functionally simulating both models using the same stimulus, or a mixture of both.

Historically, functional coverage in algorithm model verification has been mainly limited to code coverage, and some use of assertions, but more complex models involving concurrent combinational and sequential processes introduce a lot of cross product space that code coverage alone isn't able to distinguish. This is leading organizations that rely on fully verifying these models in C-based environments to look for SystemVerilog style functional coverage modeling, i.e. covergroups with coverpoints and crosses, and advanced automated stimulus generation to target them.

III. EXAMPLE OF A PORTABLE STIMULUS MODEL

The term 'portable stimulus' was chosen by a recently formed Accellera Proposed Working Group (PWG) that was tasked with determining whether a new standard for stimulus modeling should be formed and to identify what its requirements should be. The Portable Stimulus working group will commence in March, with the main requirements being that the model should be both self-contained and independent of any existing languages.

The development of stimulus modeling alternatives to traditional SystemVerilog or 'e' constrained random has been ongoing for a while with some graph-based approaches already in use in many verification groups in all areas of the globe. As the use of such techniques grows, the lack of a standard in this space starts to become more of a concern.

This paper will reference a graph-based stimulus modeling language as an example of the type of capabilities that such a portable stimulus model might be expected to have, and show how it fits into a broader verification methodology that is also portable across languages and environments.

Graph-based stimulus modeling is not new, with a long history of use in software verification and at least a decade of use in hardware verification. The graph in this case is a declarative description of legal stimulus scenarios, which is independent of any particular hardware verification language or environment. The textual description of the graph is based on rules that resemble a Backaus-Naur form, which declares how elements of the scenario can legally be combined into a sequence. Fig. 1 shows a simple example of such a set of rules.

```
rule_graph simple_protocol {
    import "decls.rseg";
    action init, do_idle;

    trans tr0;
    interface do_trans(trans);

    input state_vars curr_state;
    interface get_state(state_vars);

    simple_protocol = init repeat {
        get_state(curr_state)
        (if {curr_state.intf == ready} do_trans(tr0)) | (if {curr_state.intf == busy} do_idle)
    };
}
```

Figure 1. Rule example

The rules are hierarchical and contain three basic constructs: sequences, choices, and loops.

A graphical view generated from these rules gives a clearer indication of the intent. Fig. 2 shows the graphical form of this same example, with the interface 'do_trans' expanded to show its internal rule graph.

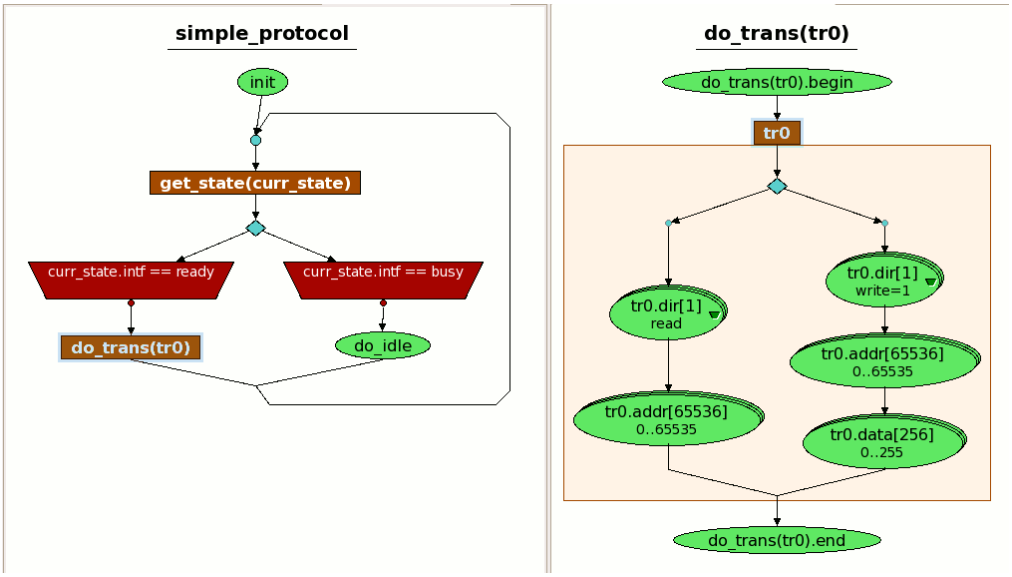

Figure 2. Example graph view

This simple example demonstrates what is meant by graph-based stimulus, and how this differs from a traditional UVM sequence/sequence item approach, which uses procedural code to define the scenario. The graph-based approach extends stimulus automation capabilities beyond generation of legal combinations of numerical quantities, to also include legal sequences of operations. The graph is therefore a description from which multiple legal scenarios can be determined.

The portability of this description arises from the ability for the nodes in the graph to be independent of any language or environment, but to contain all the necessary information for a language or environment specific implementation to be unambiguously derivable from them.

To be a viable alternative to SystemVerilog constrained random stimulus, the simple rule syntax must also incorporate the declaration of numerical quantities and algebraic constraints that define the legal combinations. The constraints in the case of the graph though can extend to encompass relationships between elements of concurrent or sequential transactions. Within a typical state-of-the-art UVM testbench environment, for example, the graph effectively becomes a source from which multiple virtual sequences can be generated. In that case, the numerical quantities and nodes within the graph become the fields of sequence items and the calls to the sequence item API respectively. Fig. 3 shows an example of a graph that defines one or more scenarios.

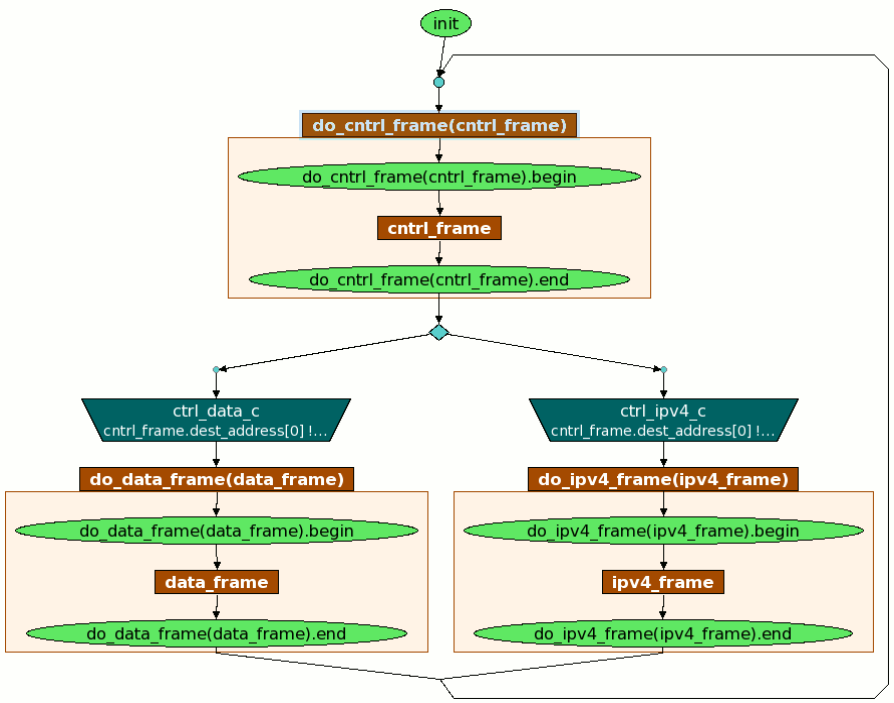

Figure 3. Example of a scenario graph

This example graph declares its scenario to be either an ethernet control frame followed by a data frame, or, alternatively, an ethernet control frame followed by an ipv4 frame. The inverted blue trapezoids represent new constraints that become active depending on the path taken, which create new dependencies on the transaction to follow. The brown rectangles are containers for the individual fields of each data structure, which map to fields of a sequence item in UVM. The green nodes are action nodes that represent calls to the sequence item API.When targeted at an abstract C testbench, those same quantities and nodes become class data members and appropriate function calls for that environment that reference them. The next section will show this in more detail using a real example.

IV. PORTABLE STIMULUS APPLIED TO A HLS FLOW

The growing use of High-level Synthesis (HLS) is one of the factors driving the need for a more advanced verification methodology in a pure C based verification environment, and also underscores the value of a reusable stimulus model that can also be applied to downstream verification.

A typical HLS flow starts with an algorithm implemented in C or C++ as the input to a high level synthesis tool and also as the Device-Under-Test (DUT) for verification. Along with one or many RTL implementation options, HLS tools can theoretically synthesize a stimulus model based on its analysis of the algorithm, and perhaps even infer some suitable verification goals & even specific coverage targets.

This is best explained by a simple example. Fig. 4 shows a C implementation for a FIR Filter with Loadable Coefficients.

```
#include "fir_filter_ld.h"
#include "stdio.h"

#pragma design top
void fir_filter_ld (ac_int<8> *inp,
                    ac_int<8> coeffs[NUM_TAPS],
                    ac_int<3,false> addr,
                    bool ld,
                    ac_int<8> *output ) {
  static ac_int<8> regs[NUM_TAPS];
  static ac_int<8> coeffs_int[NUM_TAPS];
  ac_int<19> temp = 0;
  int i;

  if(ld)
    coeffs_int[addr] = coeffs;
  else{
#pragma unroll yes
  SHIFT:for ( i = NUM_TAPS-1; i>=0; i--) {
      if ( i == 0 )
        regs[i] = *inp;
      else
        regs[i] = regs[i-1];
    }
  MAC:for ( i = NUM_TAPS-1; i>=0; i--) {
      temp += coeffs_int[i]*regs[i];
    }
    *output = temp>>SCALE;
  }
}
```

Figure 4. FIR filter function with loadable coefficients

Some numerical quantities that might form part of a stimulus model for this Device Under Test (DUT) are easily inferred from this code – they are the input arguments to the fir_filter_ld function: inp, coeffs (an array), addr and ld. The data types used would need to be translated into a more general form for the model to be portable.

A typical C testbench for this example might randomize or hard code values passed as the arguments using available functions such as rand(), and perhaps using a while or for loop to produce sequences of input arguments. If we re-arrange this testbench to conform to something closer to a UVM-like architecture, the arguments would be collected into a class or struct, analogous to a UVM sequence item, and the code that takes those class/struct members and calls the function with their values becomes the analog of the agent / bus functional model (BFM).

Fig. 5 shows an example of such a stimulus class, in this case auto-generated from the C function in Fig. 4.

```
#include <stdio.h>

class fir_filter_ld_stimulus
{
public: // data (one class member per formal argument to the function under test)
  ac_int<8, true > inp;
  ac_int<8, true > coeffs[8];
  ac_int<3, false > addr;
  bool ld;

public: // interface
  fir_filter_ld_stimulus() {}

  void initialize() {
  }

  void randomize() {
  }

  void next() {
    printf("Creating next value.");
    if (inp != 0) inp = 0;
  }
};
```

Figure 5. Stimulus Class Example

Organizing the C testbench in this fashion makes it easier for a common stimulus model to be used in the initial pure C simulation environment, and then later, when the RTL is available, to be re-used in a UVM or other SystemVerilog testbench that includes that RTL and also uses a class or struct of similarly named fields. In both cases, the portable stimulus model is used to populate the field values within the context of a scenario that might, for

example, perform some necessary initialization and then loop through a series of constrained sets of inputs to the function. The series of inputs produced might simply be targeted to achieve some cross coverage goal of the input space, and / or might setup scenarios that relate subsequent sets of these values, analogous to SystemVerilog transition coverage. Fig. 6 shows the stimulus model, which was also generated from the original C implementation.

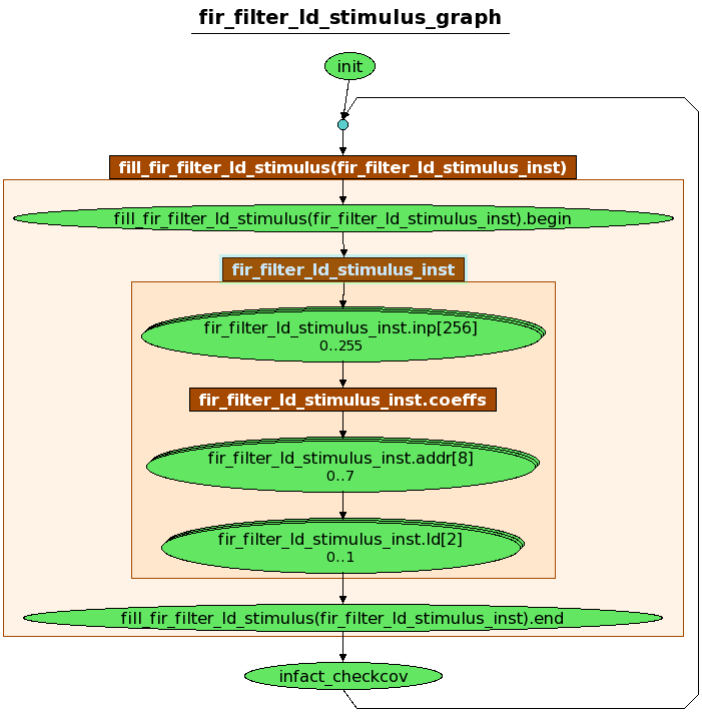

Figure 6. Graph Stimulus Model for the Fir Filter Example

For the coverage goals that the stimulus model targets, these can either be inferred by the HLS tool as mentioned before, or manually determined by the verification engineer, who may also be the developer of the algorithm at this abstraction level.

For this FIR filter function we could infer, for example, a cover point on “ld” and on a range of “inp” and on the cross of these to form the SystemVerilog style ‘covergroup’. Internal to the function we could also put an assertion on “temp” to check if it goes out of range or wrap/clip occurs.

The implementation of the coverage model in a pure C environment is possible through the use of the Unified Coverage Interoperability Standard (UCIS) API calls, the detailed code for which could be generated by the HLS tool, if it inferred the coverage goals, or generated through some friendlier user interface or convenience API as part of the verification process and toolset.

Stepping back a little from the details, we can see that we have most of the elements required for a constrained random, coverage-driven flow, all implementable in a pure C simulation environment.

The portable stimulus model itself is abstract enough to either be translated into any HVL language, or for higher-value stimulus generation tools to be linked into the C environment with their own C/C++ API. One implementation of such a tool provides a single API call to the stimulus model that replaces the SystemVerilog item.randomize() call with its own analogous function, method or task call for each of the environments and languages that it supports. The common UCIS coverage model allows coverage results to be collected, analyzed and merged alongside similar coverage from a traditional SV/UVM toolset.

Returning to the FIR filter stimulus model in Fig. 6, in the tool flow used in this example, there is an interface defined which operates on an instance of the stimulus class, called ‘fill_fir_filter_ld_stimulus’. This interface has an implementation in C++/SystemC, and in SystemVerilog. The stimulus class in Fig. 5 is easily also defined as a SystemVerilog sequence item, as in Fig. 7 below, keeping the variable names consistent.

```
class fir_filter_ld_stimulus extends uvm_sequence_item;

       `uvm_object_utils(fir_filter_ld_stimulus)

  rand bit [7:0] inp;
  rand bit [7:0] coeffs[8];
  rand bit [2:0] addr;
  rand bit ld;

endclass
```

Figure 7. SystemVerilog Sequence Item

The original verification of the C implementation is performed in a C++ testbench, wrapped in SystemC. Eventually, the synthesized RTL is instanced in a UVM testbench. If we compare an excerpt from each of these two testbenches, as in Fig. 8 & Fig. 9 below, we can see the interface method or task being called in both cases, passing a handle to an instance of the class or its analogous sequence item.

```
while (true) {
  wait();
  iteration_count++;

  // call graph to obtain next values
  fir_filter_ld_gen->ifc_fill_fir_filter_ld_stimulus(&stimulus);

  // Call C++ design
  fir_filter_ld(&stimulus.inp, stimulus.coeffs, &stimulus.addr, &stimulus.ld, &output);

  if (iteration_count > 15) sc_stop();
}
```

Figure 8. C++ Testbench Excerpt 'while' Loop

```
task body();
        fir_filter_ld_gen.ifc_fill_fir_filter_ld_stimulus(stimulus);
        start_item(stimulus);
        finish_item(stimulus);
endtask
```

Figure 9. SystemVerilog Testbench Excerpt Sequence Body Task

The benefit in this example may seem trivial, but if we consider a more typical situation, such as the transaction level object that models a complex bus protocol with dozens of variables and a large number of complex constraints, the value of moving all of that into an independent form is clearer. In this example tool flow the language dependent object just needs to contain the fields, and those are easily derived. The portable stimulus model contains the constraints and as shown, can be called from any language to provide a legal set of values for the next transaction.

V. RESULTS CHECKING & DEBUGGING

For the results checking, i.e. score-boarding, this will typically be done in the pure C environment via the use of another golden reference model, perhaps a Matlab model that is linked into the simulation. Leveraging this downstream requires us to consider other possible benefits that might be derived from an independent stimulus model.

If the technology used in stimulus generation can also provide random stability across these environments then this allows the results produced in the C simulation to be used as a reference when validating the functionality of the RTL version of the DUT. If performance considerations limit the relative amount of stimulus that we can apply to the lower level models, then the coverage targeted in the C domain would be a superset of what is expected to be achieved downstream. This suggests a methodology where the functional coverage for the C model is built in a layered way, with perhaps fewer bins, or crosses of fewer variables being included in the metrics that must be achieved at both levels. This is quite appropriate though for our intent of using this lower-level 're-verification' in conjunction with formal techniques as a means to establish equivalence between the initial C algorithm and the synthesized RTL, as mentioned earlier in section II.

For debugging issues, any random-stability provided can also allow a problem found in one domain to potentially be debugged in the other, using the ability to replay the same stimulus sequence in each case using a seed mechanism or something similar.

If both domains also use the common UCIS standard and tools that support it for coverage collection and analysis, then a continuous verification management flow can also be maintained.

VI. RECAPPING THE PROPOSED HLS VERIFICATION FLOW

Fig. 10 below shows the flow that is enabled by incorporating these elements into a coherent methodology.

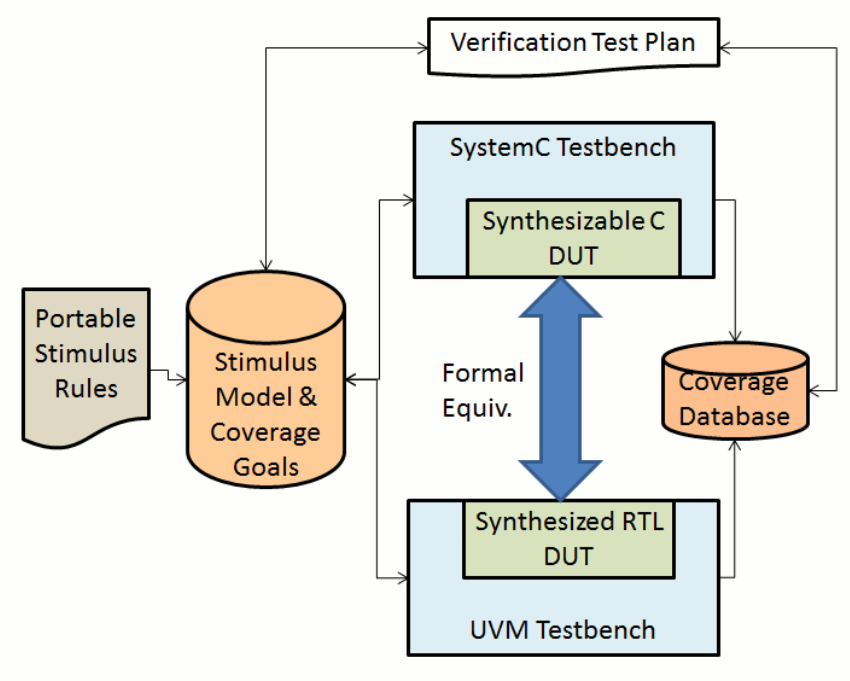

Figure 10. Flow diagram

To recap the capabilities, and the level of automation, that could be enabled by such a flow, this could include:

a) Generation of stimulus model and initial coverage goals from HLS tool's analysis
b) Re-usable stimulus and reference results across model abstractions and verification languages
c) Common, merge-able coverage results across model abstractions and environments

VII. SCALABILITY TO OTHER VERIFICATION ENVIRONMENTS

It is not enough to merely serve one flow with these methodology and model changes, so we must test this against others potential applications. Use of emulation is another area where finding common stimulus models between different domains can be a challenge. Again, SV/UVM solves the problem for simulation, and in some cases, testbench acceleration with links from UVM to a hardware box is a feasible option. In other cases the performance requirements or interface limitations dictate that a fast C-based stimulus model is needed, so our portable stimulus model may again be used here, with similar benefits, i.e. re-use a single stimulus model in both domains, and then again debug in either domain using random stability if available.

VIII. CONCLUSION - LOOKING TO THE FUTURE

The key advantage of this approach is reusability of the stimulus model across multiple representations of the same design. The design verification (DV) engineer can now focus on achieving architectural verification goals at higher levels of abstraction, and then take the same stimulus model to verify implementation details at the appropriate lower level, while proving the functional equivalency between design abstractions.

Furthermore, metric driven verification concepts, established and proven in the RTL domain, are extended to electronic system level (ESL) providing higher confidence in the design and greatly improving verification productivity.

The stimulus model itself can be fully verified at the ESL level and the DV engineer does not need to re-write or translate multiple stimulus models for the test benches used at the lower abstraction level. The ESL stimulus model

can be further refined by a DV engineer for any lower abstraction level specifics, especially related to closure on functional coverage metrics that are not practical or possible to verify at ESL.

This additional opportunity for re-use, helped of course by standardization through Accellera, of the portable stimulus model description will improve the overall DV flow and increase overall project productivity, targeting the verification part that is increasingly becoming the bottleneck for large contemporary designs.

It also has the possibility to bridge two teams, those working at ESL and the implementers, who currently speak different languages, with a formalized stimulus model and a coherent view of functional coverage.