Parameter Passing from SystemVerilog to SystemC for Highly Configurable Mixed-Language Designs

Bishnupriya Bhattacharya, Samik Das, Zhiting Duan, Chandra Sekhar Katuri, Pradipta Laha

Cadence Design Systems,
2655 Seely Avenue, San Jose CA 95134
{bpriya, samikd, zhiting, csekhar, plaha}@cadence.com

***Abstract*-The value of electronic system level (ESL) models is greatly improved if they can readily and easily be configured. At present, the Accellera Configuration, Control and Inspection Working Group (CCI WG) is actively working on the CCI parameter standard for instrumenting SystemC models to be configurable. The CCI (draft) standard adds the ability to configure a SystemC simulation model at creation-time and run-time, not just at compile-time. In this paper, we extend SystemC CCI parameters to the SystemVerilog-SystemC boundary, and define the syntax and semantics of passing parameters from a SystemVerilog parent model to a child SystemC model instantiated inside the SystemVerilog parent. This work seamlessly blends the native parameter feature of SystemVerilog with the CCI parameter (draft) standard of SystemC in a natural and intuitive fashion that is organic to both languages, thus enabling an enhanced user experience and a powerful mixed-language modeling paradigm that is highly configurable. We have implemented our work in the Incisive® simulator, and illustrate it with a mixed-language parameter-passing example.**

I. INTRODUCTION

In recent years, both SystemC [1] and SystemVerilog [2] have emerged as mature, and popular languages for design and verification in the EDA industry. In today's complex designs and test benches, it is no longer an aberration, but rather the rule, to encounter mixed-language usage. An IEEE or Accellera standard does not exist yet for mixed-language simulation semantics. However, commercial simulators have supported mixed-language simulation between SystemVerilog and SystemC for some time now [8, 9].

In the mixed-language design and verification space, there are two primary use models

- Function calling from one language to another
- Cross-language instantiation between the two languages

Function calling between SystemVerilog and SystemC can naturally build upon the DPI-C function calling standard specified in IEEE-1800 [2], and intuitive extensions have been proposed to support function calling between SystemVerilog and C++/SystemC [7].

For the instantiation-based mixed-language use model, there is no formal mixed-language standard to build upon. However, a natural strategy is to follow the instantiating language's syntax and semantics to the extent that it can be translated to equivalent semantics in the instantiated language. In the instantiation use model, both in SystemVerilog and in SystemC, ports are one important part of the instantiated module's interface, which needs to be bound to wires/signals by the instantiating module. Another important part of a module's interface are its parameters, which is our focus area in this work.

SystemVerilog defines a formal syntax and semantics of parameter declarations and overrides [2]. The SystemC language [1] does not yet define parameters as a formal language element. Consequently, parameters in SystemC today are mostly modeled as a SystemC module's constructor arguments. In order to bridge this gap, the Accellera Configuration, Control and Inspection Working Group (CCI WG) has taken up the charter of formally defining a

parameter specification in SystemC [4]. The CCI WG has already published a draft Language Reference Manual (LRM) for internal review [5], and is actively engaged at this time in finalizing a formal Accellera standard for SystemC parameters. We are early adopters of the CCI parameter (draft) standard in SystemC. In this paper, we extend SystemC CCI parameters to the SystemVerilog-SystemC boundary, to address the parameter passing requirements in the mixed-language instantiation use model. Specifically, we define the syntax and semantics of passing parameters from a SystemVerilog parent module to a child SystemC module instantiated inside the SystemVerilog parent. Instead of ad-hoc and customized parameter passing solutions between SystemVerilog and SystemC, this work innovatively builds upon the standard parameter features in both languages, and brings them together. The mixed-language parameter specification proposed here is designed to seamlessly blend the native parameter feature of SystemVerilog with the CCI parameter (draft) standard of SystemC in a natural and intuitive fashion that is organic to both languages, thus enabling an enhanced user experience and a powerful mixed-language modeling paradigm that is highly configurable.

The rest of the paper is organized as follows: In Section II we provide a brief overview of SystemC CCI parameters. In Section III, we present our contribution and define the semantics of parameter passing from SystemVerilog to SystemC. In Section IV, we illustrate our work with a mixed-language parameter passing example in the Incisive simulator [8]. Finally we conclude in Section V with promising directions for future work.

II. Overview of CCI Parameters

The Accellera (draft) Configuration, Control and Inspection LRM [5] establishes a standard way to instrument SystemC models to be configurable. Today, parameters are mostly modeled as SystemC module constructor arguments. CCI adds the ability to configure a SystemC simulation model at creation-time and run-time, not just at compile-time. This adds flexibility for users, as they do not have to modify source code and recompile to try various parameterized configurations.

The (draft) CCI LRM defines the syntax and semantics of parameter modeling in SystemC, and how parameters interact with a simulation tool. A brief summary is provided here that is relevant for this work.

Inside a SystemC module, parameter objects are declared using the *cci_param* class. The *cci_param* class has two template parameters. The first template parameter represents the data type of the parameter, and the second template parameter represents its mutability property.

```
namespace cci { namespace cnf {
 enum param_mutable_type {
      mutable_parameter = 0, // parameter value can change anytime during simulation
      immutable_parameter,// parameter value is constant throughout simulation
     elaboration_time_parameter, // parameter value can change only during elaboration
     other_parameter // Vendor specific/other Parameter type
 };
 template<typename T, param_mutable_type TM = mutable_parameter>
 class cci_param : ….. {
 public:
    cci_param(const char* name, const T& val); // constructor that creates parameter in parent scope with a default value
    …
    virtual void set(const T& val);
   virtual const T& get() const;
  ….
 };
}}
```

A *cci_param* constructor optionally accepts a default value. Upon construction, a CCI parameter is typically assigned a hierarchical name similar to other SystemC objects, and belongs to the scope of its parent module/process. However, it is also possible to create *toplevel* CCI parameters that do not belong to the parent scope. The *cci_param* class provides APIs to read/write the parameter value.

Configuration brokers hide the parameter implementation details from an application that uses parameters. Name-value pairs in the application are mapped by a broker to corresponding parameter objects. Every CCI implementation provides a global broker. The broker provides API *set_init_value()* to set up the initial value of a parameter object before the parameter is constructed. The initial value is specified as a string using the JSON format [6].

```
namespace cci { namespace cnf {
 class cci_cnf_broker_if {
 public:
  virtual void set_init_value(const std::string &parname, const std::string &json_value) = 0;
  virtual void lock_init_value(const std::string &parname) = 0;
  …
 };
}}
```

During top-down design hierarchy construction, *set_init_value()* calls are processed temporally by the broker. A later call overwrites an earlier call, provided the initial value is not locked. If a module higher up in the design hierarchy wants to ensure its initial value is not overridden by another module further down the hierarchy, it has the option of calling the broker API *lock_init_value()* after calling *set_init_value()*. When a parameter is constructed, it looks up the broker for any initial value that was specified for it, and on a match, assumes that value. If an initial value has not been specified, it assumes the default value specified in the *cci_param* constructor. Apart from programmatic APIs, parameter initial value overrides can also be specified in an external configuration file that the simulation tool processes and applies with the highest precedence, overriding any programmatic *set_init_value()* calls.

Callback functions may be registered with a parameter. A registered callback function is automatically called when the parameter is accessed. A callback function has the signature below.

```
callback_return_type callback_name(cci_base_param& param, const callback_type& cb_reason);
namespace cci { namespace cnf {
 enum callback_type {
  pre_read,
  reject_write,
  pre_write,
  post_write,
  create_param,
  destroy_param
 };
 enum callback_return_type {
  return_nothing,
  return_value_change_rejected,
  return_other_error
 };
}}
```

III. Parameter Passing from SystemVerilog to SystemC

In this section, we describe our specification for interfacing SystemVerilog parameters with CCI parameters in SystemC.

For SystemVerilog instantiating SystemC, we specify the semantics both for shell-based instantiation as well as ‘direct’ instantiation without a shell. In the shell-based instantiation use model, a SystemVerilog shell module exists corresponding to the SystemC module. This shell module presents the same interface (ports and parameters) in SystemVerilog as the original SystemC module. We treat the shell module purely as a syntactic artifact, and no semantic or behavior is associated with the shell.

Our parameter passing specification is designed to be orthogonal to the shell module, and works both for shell-based instantiation as well as for direct instantiation.

A. Parameter Declaration Syntax

In the SystemC model, declare CCI parameters natively. No special syntax is necessary to indicate these are visible in SystemVerilog. Thus the SystemC model is a 'normal' SystemC model that can be instantiated either in SystemC or in SystemVerilog.

All three kinds of CCI parameter mutability are supported at the language boundary:

- *immutable*: value is constant throughout simulation
- *elaboration_time*: value can change only during elaboration
- *mutable*: value can change anytime, including during simulation

In SystemVerilog, parameters are like a constant and parameter value is resolved by the end of elaboration. It may be a more natural fit with SystemVerilog to only support *immutable* and *elaboration_time* CCI parameters at the language boundary. However, as shown in Section II, the default value of the second template parameter for *cci_param* class is *mutable_parameter*. Typically, the user will declare his parameter with the data type specified, but without explicitly specifying the mutability, e.g. *cci_param<int> p1*. In that case, the parameter implicitly becomes a *mutable* parameter, although the user may not actually plan to change the value of that parameter at simulation run-time. If *mutable_parameter* is not supported at the SystemVerilog-SystemC boundary, such common usage will become illegal. Keeping this in mind, all three kinds of mutability are supported at the language boundary. Refer to Section III.D for further discussion of *mutable_parameter* at the language boundary.

In SystemVerilog, using native SystemVerilog syntax, declare corresponding parameters in the SystemVerilog shell model that represents the SystemC model. The name of the SystemVerilog shell parameter has to match the SystemC name of the CCI parameter. This is the name specified when constructing the CCI parameter, as opposed to the C++ variable name of the parameter. In the example in Fig. 1, the C++ variable name of the CCI parameter is *scp1*, but its SystemC name is *p1*. Consequently, in the SystemVerilog shell model, the parameter has to be declared with name *p1*. If a SystemVerilog shell is not used, then the name matching happens during direct instantiation of the SystemC module in SystemVerilog, and name-based value override has to be used. In Fig. 1, when instantiating SystemC module *sc1*, the parameter *p1* is overridden using a name-based association. This name has to match the SystemC name of the CCI parameter. Refer to Section III.D for a discussion on name-based parameter value override vs. positional value override for the direct instantiation use model.

For the rest of this document, we refer to a CCI parameter in a SystemC module instantiated inside a SystemVerilog parent, and visible to the SystemVerilog parent as a 'boundary' CCI parameter.

B. Parameter Data Type Mapping

We define the parameter data type mapping at the SystemVerilog-SystemC boundary following the DPI-C data type mapping for function/task arguments specified in IEEE 1800 [2], and enhance it further to include SystemC types. The data type mapping is shown in Table 1. For multiple mappings, the default mapping is indicated with an asterisk. SystemVerilog *type* parameter, or parameters of type unpacked array and dynamic array are not supported at the boundary.

In SystemVerilog, explicitly declaring the type of a parameter is optional. If a type is specified during declaration, it is strongly typed, and any assignment of a value to the parameter that is not of the declared type is converted to its declared type and then assigned. If during declaration, a type is not specified, it is weakly typed and its type is assumed to be the type of the value assigned to it. For weakly typed parameters, if a *defparam* statement [2], or an instantiation override specifies a different value type than the type of the initial parameter value, the assigned type wins.

At the SystemVerilog-SystemC boundary, to reduce complexity and ambiguity, we only support strongly typed SystemVerilog parameters. This implies, in the shell-based use model, the SystemVerilog parameters must be declared with an explicit type in the SystemVerilog shell module. For the direct-instantiation use model, the type of the SystemVerilog parameter is inferred from the SystemC data type according to the mapping in Table 1, and the rules for strongly typed SystemVerilog parameters are applied.

TABLE I
DATA TYPE MAPPING BETWEEN SYSTEMVERILOG AND SYSTEMC PARAMETERS

SystemC data type	SystemVerilog data type
char	byte*, byte signed
unsigned char	byte unsigned
short	shortint*, shortint signed
unsigned short	shortint unsigned
int	int*, int signed
unsigned int	int unsigned
long long	longint*, longint signed
unsigned long long	longint unsigned
double	real
float	shortreal
bool	bit*, bit unsigned
std::string	string
sc_bv<N>	bit[0:N-1]*, bit unsigned [0:N-1]
sc_uint<N>, sc_biguint<N>	bit[0:N-1]*, bit unsigned [0:N-1]
sc_int<N>, sc_bigint<N>	bit signed [0:N-1]
sc_logic	logic
sc_lv<N>	logic[0:N-1]

C. Use Case Restrictions for Boundary CCI Parameters

SystemC allows CCI parameters to be created anytime, both during elaboration and during simulation. In SystemVerilog, parameters are like a constant and parameter value is resolved by the end of elaboration. To converge between the two languages, for boundary CCI parameters, we restrict creation to the constructor of the owning SystemC module. So, CCI parameters created in the constructor of the owning SystemC module are visible at the boundary inside the parent SystemVerilog module. Conceivably, this can be extended to CCI parameters created in the other elaboration phases in SystemC like *before_end_of_elaboration* and *end_of_elaboration* also. CCI parameters created during simulation are not visible at the SystemVerilog-SystemC language boundary.

CCI parameters can be created as *toplevel* parameters (refer to Section II). However, *toplevel* parameters are not supported at the boundary. These parameters do not belong to the scope of the parent module and do not align well with SystemVerilog parameters that always belong to the parent module's scope.

Note that it is legal for a boundary SystemC module to contain *toplevel* CCI parameters, or CCI parameters created during simulation, however, these won't be visible in the SystemVerilog parent module.

D. Semantics of Parameter Value Propagation

Default Value of CCI Parameter

The default value of a boundary CCI parameter is owned by SystemC. In the shell-based use model, SystemVerilog syntax requires specifying a default value for the parameter declared in the SystemVerilog shell module. This default value specified in the SystemVerilog shell is ignored (e.g. see line 4 in *top.v* in Fig. 1). The default value specified in the SystemC module prevails.

CCI does allow the default value of a CCI parameter to be left unspecified in SystemC, i.e. the user may not provide a default value when constructing a SystemC parameter. In pure SystemC designs, if the default value is unspecified, and there is an attempt to read that parameter, a C++ exception is thrown that the user can catch in his SystemC source code. For mixed-language semantics, it is inappropriate to throw a C++ exception if SystemVerilog tries to read the value of a SystemC boundary parameter with an unspecified default value. Hence for boundary CCI parameters, if the user has left the default value unspecified, a default value appropriate for that data type is assigned. An implementation can print a warning in such cases.

Overriding Parameter Value from SystemVerilog Instantiation

SystemVerilog can provide instantiation override for a boundary CCI parameter when instantiating the SystemC module. The syntax of instantiation-override exactly follows the native SystemVerilog syntax.

For shell-based use model, both positional override and name-based override are supported for boundary parameters. The parameter overrides map to the parameter declarations in the SystemVerilog shell. For example, see lines 12 and 13 in *top.v* in Fig. 1.

For direct-instantiation use model, only name-based override is supported (using SystemC name and not C++ variable name). Since the SystemC name is used for matching, the order of construction in SystemC module is important. If there are pointer CCI parameter declarations inside a module that are interleaved with non-pointer parameter declarations, the order of construction is no longer guaranteed to be the order of declaration. It depends on the order in which the user has dynamically allocated the pointer CCI parameters. Hence positional override is not well-defined or straightforward to establish just by viewing the class declaration of the SystemC module. Hence, we restrict parameter overrides to name-based overrides for direct instantiation.

SystemVerilog also supports modifying a parameter value using the *defparam* construct. Since *defparam* is a deprecated language construct, and may go away in the future [2], *defparam* overrides are not supported for boundary CCI parameters. An implementation may choose to issue an error in this case.

SystemVerilog Override Contributes to Initial Value of CCI Parameter

SystemVerilog overrides contribute to the initial value of a CCI parameter. As explained in Section II, CCI parameters can be assigned initial value(s) before the parameter is constructed, and on parameter construction, CCI provides a well-defined semantics on how to choose the value.

When SystemVerilog instantiates a SystemC module, and specifies a value override for a boundary CCI parameter, it translates to a *set_init_value()* call on the corresponding CCI parameter. The *set_init_value()* call is made just before invoking the constructor of the SystemC module. The normal SystemC semantics of *set_init_value()* applies. If the initial value of the parameter has already been set and locked (say from external configuration file, or from a SystemC module higher up in the hierarchy), this *set_init_value()* call from SystemVerilog fails, and the implementation issues an error message. It is implementation-specific if the error is fatal or not. If the *set_init_value()* call from SystemVerilog succeeds, it is not followed by a *lock_init_value()* call.

If no override has been specified in SystemVerilog, no *set_init_value()* call is made, and the default value in SystemC (or any SystemC override, or external configuration file override) prevails.

To summarize CCI parameter mixed-language initial value semantics:

- For initial values specified in an external configuration file, the implementation ensures these have highest precedence.
- For initial values specified programmatically in SystemC module constructors using *set_init_value()*, it is up to the SystemC user to also issue a corresponding *lock_init_value()* call on parameters of as-yet-unconstructed modules further down the hierarchy. If *lock_init_value()* is issued, the issuing module's initial value always wins, otherwise, *set_init_value()* calls issued lower down the hierarchy win.
- An instantiation override from SystemVerilog translates to a *set_init_value()* call as the mixed-language top-down hierarchy is traversed and constructed. This call should happen before the SystemC module that owns these parameter is constructed. This is very close to mimicking what would have happened if the SystemC module had been instantiated in a SystemC parent, and the SystemC parent had issued a *set_init_value()* call on the child SystemC module's CCI parameter.

The *set_init_value()* API in SystemC uses a JSON string representation for specifying the value. However, the SystemVerilog user is not burdened to specify value overrides using JSON string representation. He continues to use the native SystemVerilog syntax for representing the value override, according to its data type. It is up to the implementation to translate it such that the effect is the same as if an equivalent *set_init_value()* call was issued. An implementation can choose to bypass JSON string altogether and use a dummy value in the *set_init_value()* call from SystemVerilog. Later, during CCI parameter construction, when the data type is known, if the initial value from SystemVerilog wins, an implementation can directly translate the value from SystemVerilog data type to SystemC data type according to Table 1.

Final Value of CCI Parameter is Propagated to SystemVerilog

After a boundary CCI parameter is constructed, and its final value is determined, that final value is propagated to SystemVerilog. For example, in Fig. 1, the final value of boundary parameter *top.sc1.p1* is 17, and this value is visible in the SystemVerilog *$display* statement for *top.sc1.p1*, and is also propagated to the dependent SystemVerilog parameter *top.v1.cp*.

Boundary CCI parameter construction is defined as the final synchronization point for parameter value passing between the two languages. After construction, further value changes to a boundary CCI parameter using the *set()* API is not propagated to SystemVerilog. An implementation can choose to issue an error on any *set()* calls, or continue with a warning that the value change is only visible on the SystemC side of the design.

SystemVerilog Accesses Do Not Trigger CCI Parameter Callbacks

SystemVerilog accesses on a boundary CCI parameter do not trigger any callbacks registered on that CCI parameter in SystemC. Semantically, it is conceivable to trigger *pre-read* callbacks for every SystemVerilog read access, but that is not part of this specification yet. The other callback reasons described in Section II are not applicable to SystemVerilog instantiation overrides.

IV. RESULTS

The specification described here has been implemented in the Cadence® Incisive simulator [8] for the shell-based use model. At this time, a subset of the data type support specified in Table 1 is available.

The example in Fig. 1 shows a mixed-language SystemVerilog-SystemC design in the Incisive simulator. The functional output from the Incisive simulator, with the user-provided external parameter configuration file is shown in Fig. 2.

```
// sc.cpp
1  SC_MODULE (scmod) { // declare SystemC module ‘scmod’
2  public:
3    SC_CTOR(scmod) : // constructor of SystemC module ‘scmod’
4       scp1(“p1”, 0) // construct CCI parameter with SystemC name ‘p1’ and default value 0
5    {
6      …….
7    }
8  protected:
9    cci_param<int> scp1; // declare CCI parameter with C++ variable name ‘scp1’
10  };

// top.v
1  `timescale 1 ns/1 ns
2  module scmod () // declare SystemVerilog shell module ‘scmod’
3    (* integer foreign = "SystemC"; *); // Incisive syntax to indicate this is a shell module for SystemC
4    parameter int p1 = 11; // declare parameter ‘p1’ with default value of 11; this default value is ignored
5  endmodule
6
7  module child #(parameter cp = 1); // declare SystemVerilog module ‘child’ with parameter ‘cp’
8  endmodule
9
10  module top; // declare SystemVerilog module ‘top’
11    parameter vp = 9; // declare parameter ‘vp’
12    scmod #(.p1(vp+1)) sc1(); // instantiate SC module ‘scmod’ using name-based parameter value override
13    scmod #(29) sc2(); // instantiate SC module ‘scmod’ using positional parameter value override
14    child #(top.sc1.p1) v1(); // instantiate SV  module ‘child’, using boundary CCI parameter value
15                                  // ‘top.sc1.p1’ to override the SV module’s parameter value of ‘cp’
16    initial  $display(“top.sc1.p1 = %d", sc1.p1); // print the value of boundary CCI parameter ‘top.sc1.p1’
17    initial  $display(“top.sc2.p1 = %d", sc2.p1); // print the value of boundary CCI parameter ‘top.sc2.p2’
18    initial $display (“top.v1.cp = %d”, cp); // print the value of SystemVerilog parameter ‘top.v1.cp’
20  endmodule
```

Figure 1. Example of parameter passing from SystemVerilog to SystemC

```
// user can override CCI parameter value through external configuration file
$> cat param.conf
  top {
    sc1 {
      !parameters {
        p1 "17" <----- set up initial value of ‘top.sc1.p1’ as 17
      }
    }
  }
$> irun –sysc –top top sc.cpp top.v –scconfig param.conf  <------ invoke Incisive simulator
```

top.sc1.p1 ***17*** <------- initial value specified in external configuration file wins

top.sc2.p1 ***29*** <------- initial value specified in instantiation override from SystemVerilog wins

top.v1.cp ***17*** <------- boundary CCI parameter’s final value is propagated to dependent parameters in SV

Figure 2. User provided external parameter configuration file, and Incisive simulator output for the SystemVerilog-SystemC parameter passing example in Fig. 1.

V. FUTURE WORK

We plan future work in this area to define parameter passing semantics in the reciprocal direction of SystemC instantiating SystemVerilog. Another natural area of work is to define a correspondence between VHDL *generic*s [3] and CCI parameters in SystemC.

Current activity is ongoing in the CCI WG to formally standardize the CCI parameter specification, and as part of the standardization effort, some changes are expected over the current (draft) LRM [5]. Once the standard is finalized, we will analyze and update if any corresponding changes need to be incorporated in the SystemVerilog-SystemC parameter specification described in this work.

REFERENCES

[1] 16666-2011, IEEE SystemC Language Reference Manual, available at www.systemc.org, 2011.
[2] 1800, IEEE SystemVerilog Language Reference Manual, 2005.
[3] 1076, IEEE VHDL Language Reference Manual, 2008.
[4] Accellera Configuration, Control, and Inspection Working Group, http://www.accellera.org/apps/org/workgroup/cciwg/.
[5] CCI LRM Draft 4, http://www.accellera.org/apps/org/workgroup/cciwg/download.php/5432/CCI%20LRM%20draft4.pdf, 2011.
[6] JSON, Java Script Object Notation, www.json.org.
[7] Arnab Saha, Doug Warmke, Hui Yin, "Introducing DPI-C++," DVCon, 2009.
[8] Incisive Enterprise Simulator, http://www.cadence.com/products/sd/enterprise_simulator/pages/default.aspx.
[9] Modelsim, http://www.mentor.com/products/fpga/simulation/modelsim?sfm=auto_suggest.