Requirements Specification for Configuration Interfaces

Version 1.0

December 18, 2009

Configuration, Control & Inspection (CCI) Working Group
Open SystemC Initiative

Bugs and Suggestions

Please report bugs and suggestions about this document to

http://www.systemc.org

Document History

Version	Authors	Description	Date
1.0	CCI WG	Initial revision for public review.	12/18/2009

Motivation

The value of ESL models is greatly improved when they can readily and easily be configured, controlled and inspected. Delivering this value to users of a system-level model requires that all ingredient models are fully instrumented, that the user employs a tool with configuration, control and inspection capabilities and that the tool fully understands the instrumentation of all ingredient models. If any of these three requirements is not satisfied, the value delivered to the user can be greatly diminished.

Today, each tool has its own preferred way for models to be instrumented. This requires that model developers provide different instrumentation for their customers using different tools; their only way to offset the costs of supporting multiple instrumentation approaches is to provide less instrumentation. Tool companies must bridge their tools and/or the models to ensure value for customers that have developed or obtained models with incompatible instrumentation. All of this results in delays, additional costs and/or diminished value for the end user.

Establishing a standard way to instrument models will improve the efficiency of the entire SystemC modeling ecosystem while ensuring optimal value for model users – and that is precisely what the OSCI CCI WG has set out to do.

Introduction

The CCI WG will define standards to improve interoperability between models and tools, streamlining the exchange of information between them. These standards will be developed incrementally with initial work on configuration, which is the focus of this requirements specification.

What CCI brings is the ability to configure a SystemC simulation model at creation-time and run-time using a flexible and rich system, not just at compile-time using basic C++ mechanisms. This adds flexibility for users, as they do not have to modify source code and recompile to try various scenarios or parameterized configurations.

Models will incorporate configuration parameters following the CCI standard, allowing any tool to connect and perform configuration. Model source code will be portable across simulation environments and tools, while tools are able to add value on top of the basic mechanisms. For example, tools can support the reading of configuration files, interactive configuration of models in a GUI, and structured display and inspection of current parameter values. None of this affects how models are written – as long as they follow the CCI standard, tools will be able to effectively interact with them.

Configuration can be used to specify a system's initial setup, orchestrate run-time operation, control system analyses or many other purposes. Some examples are:

- A system model might have a configurable number of processing engines, DMA controllers, processors, or other blocks. In response to configuration parameter values, the system model will instantiate suitable sets of hardware blocks.

- Configuration parameters can affect the local setup of models, such as the size of memories and buffers, number of channels of an interconnect, or the command set supported by flash memories.
- Configuration could also be used to set up the memory map of a system, and report the memory map resulting from configuring dynamic buses such as PCIe.
- Configuration parameters can affect system behavior, changing iteration counts, identify tests, or setting different operation modes.

These requirements are being provided for public review to collect industry feedback which is highly valued and will certainly influence development of the standard.

The next section provides a Glossary of Terms for convenient reference. A section is provided to describe the conceptual model used by the CCI WG to explore and understand the configuration requirements. A description of the use cases used to drive requirements is provided. A section is then dedicated to the subject of data types. The requirements section follows and the document concludes with future plans and a few notes.

Glossary of Terms

CCI API – The application programming interface (API) used to interact with the configuration system; this is the essence of the configuration standard.

CCI Application – A client of the CCI API. Programs utilizing the CCI configuration standard must be linked to a CCI implementation; SystemC models using CCI will typically be linked with both a SystemC simulation kernel and a CCI implementation.

CCI Implementation – An implementation of the CCI API; this is sometimes referred to as "the tool".

Configurator – A CCI application that performs tool-like operations, such as processing the contents of a configuration file, using only the CCI API.

Name-Value Pair (NVP) – synonymous with "Parameter".

Parameter - A unique, hierarchical name paired with a value.

Parameter Handle (PH) – A reference to an NVP that allows subsequent value access without requiring a name-based lookup, for improved efficiency. The Parameter Handle has a simplified interface in comparison to a Parameter Object.

Parameter Object (PO) - An entity that contains supplemental information about an associated NVP such as its parameter type, data type, default value and documentation. A PO is mapped to exactly one NVP.

Configuration Conceptual Model

In the process of exploring and understanding configuration requirements, the CCI WG has formulated an abstract conceptual model. It is briefly described here as an aid to more quickly understanding the requirements and to allow more concise descriptions for the use cases and requirements that follow.

The essential configuration parameter is the pairing of a name and a value, or Name-Value Pair (NVP). The name is a string that commonly reflects the SystemC module hierarchy but isn't restricted to it; in fact, parameters can be named completely arbitrarily. The value is accessed via JSON, a portable value encoding mechanism (see Requirement 19, *Portable Parameter Value Encoding*, for details). How and where the value is stored is an unspecified implementation detail. Conceptually, the parameter values have no inherent type – which simplifies the mainstream notions of a configuration file processor pushing in string values and models pulling out values of other types (e.g. integer).

Name-based access functions will be provided to set and get parameter values. These functions query the available collection of NVPs to find one with a matching name and then set or get its value, as requested.

For convenience and efficiency, Parameter Objects (POs) are provided as an alternate NVP access mechanism. A PO is "mapped" to an NVP. POs specify a type that the NVP's value will be interpreted as, a default value (of the PO's type), a description, and possibly more information including state. POs provide type-specific value access methods and will generally be used within models.

The partitioning between NVP and PO is key to allowing un-typed initial values to be established prior to the construction of modules which then specify types for the parameters. It also lends itself to employing arbitrary algorithms for mapping POs to NVPs which can be used, as an example, to give precedence to the highest parameter in a given hierarchy when multiple matches exist.

Figure 1 illustrates the conceptual model and key terms from the glossary.

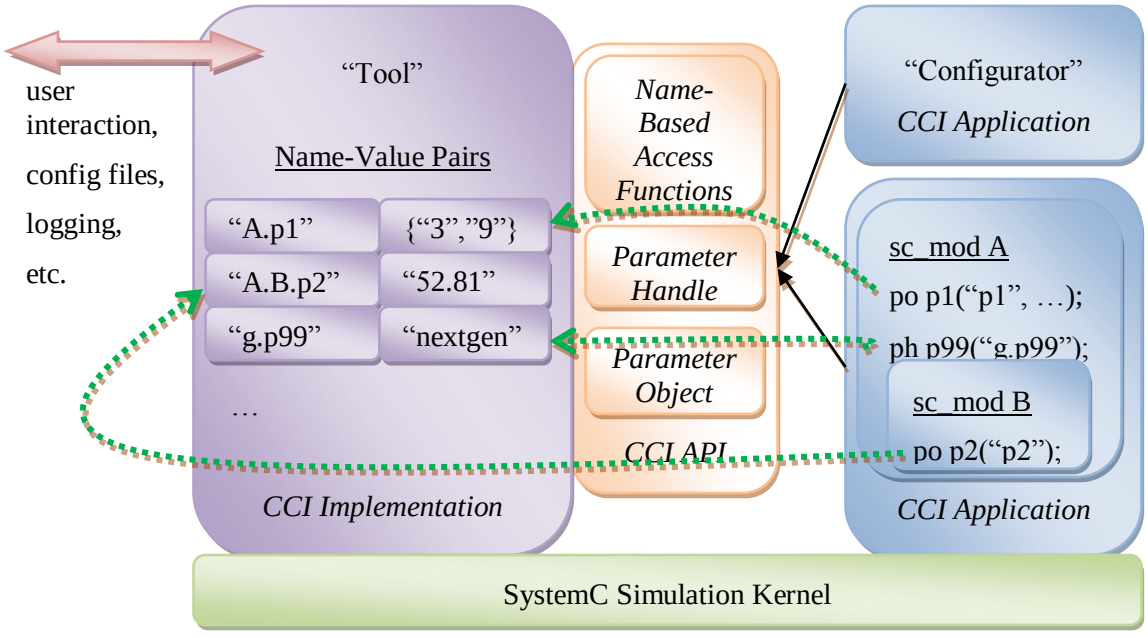

Figure 1: Configuration Concept Model

Use Cases

To explain how configuration can be used in a system, we present some examples of how parameters can be used to configure and setup a system.

1 Structural Parameters

Parameters may be used to configure the structure of a module or channel, for instance, the number of child modules. To support this use case, it needs to be possible to access parameters at elaboration time, from within a module's constructor.

2 Parameters and Constructor Arguments

Currently, constructor arguments are frequently used to implement structural parameters. To realize the benefit of configurability, explicit parameters should be preferred and coding guidelines/recommendations should be provided to explain this.

3 Default Values and Initial Values

To support structural parameters, it is necessary that parameter values are available when a model is starting to be constructed. This includes using default values for parameters, as well as initial values provided by a tool, and possibly overrides of defaults or initial values by a user. In any case, such value changes have to be completed before model construction begins.

4 Read-only Parameters

Since a structural parameter takes effect as a system is being constructed, once the system construction is complete they have to be considered to be read-only. Changing a structural parameter too late in the process will have no effect on the system, and their values are available only for reference and system inspection, not for change.

5 Elaboration Time Parameter Propagation/Override

To supersede the use of constructor arguments implementing structural parameters, a hierarchical model needs the ability for parent modules to provide values for the parameters used during the construction of its child modules.

6 Configuration Values vs. C++ Variables

Similar to constructor arguments, C++ variables in a model that reflect some configurable aspect should replaced by configuration parameters. This would also allow a tool to reconfigure model aspects that can change during a simulation, and inspect the current state of a model.

7 Changing Parameters During Simulation

Users need a means to alter certain parameter values after elaboration, during simulation run-time. This is often the case when the data field of a SystemC module or channel is replaced by a parameter to achieve run-time configurability. When the value of a parameter is altered by a tool, component-specific code may need to run to make the

model behave in a new way and create a new consistent model state. One way that this can be implemented is for a model to get a notification in the form of a callback when a parameter is changed.

8 Shared parameters

Multiple SystemC module and channel instances may depend on a common parameter. For example, a clock speed parameter shared among all processors in a homogenous cluster, or verbosity level for logging.

9 Different Names, Same Intended Meaning

When models come from various sources, it is possible that parameter names and semantics are different for what is logically the same value. For instance, one module may have a parameter "clk_frequency" expressed in Hertz while another module uses the parameter name "clock_speed" expressed in Megahertz.

Such differences are to be expected, and tools will have to support users to help them create easy-to-use configurations in the case that the same logical value should go to multiple inconsistent parameters. For example, users integrating models could create code that sets the value of one parameter based on the value of the other, doing unit conversion and other semantic adjustments in the process.

10 Authoring

There are tools that support users in the creation, maintenance, debugging, and integration of models. Such tools need to know about CCI configuration parameters and their meaning, so that they can work with the parameters associated with a model configuring it. The CCI API will be designed to support tools as well as users in the process of creating models.

11 Error Checking

There is always the possibility of errors when using parameters. The CCI working group expects many tools to offer a way to specify a set of parameter values prior to actually creating models, and in this case, it has to be possible to detect that certain parameter values were never used (either due to mistakes or intentionally). It is also necessary to check the values assigned to parameters so that they make sense (like any input to a program), and the configuration system will report errors to users. Finally, once a system has been setup, there might be global consistency checks between parameters in different models and different subsystems, ensuring that the overall system configuration is meaningful.

12 Model-to-Model Configuration

It is not just tools and users that will configure models. Models can also provide initial configuration data, or update the configuration of other models. This could apply within the model hierarchy, as a higher-level component creates its children and provides them with configuration data. It could also be used to reroute transactions in a memory map, or change the logging verbosity of a model from another related model.

13 Parameter hiding

In a hierarchical model, there can be cases where a parent module wants to hide certain parameters of its child modules within the subsystem that it represents. Configuration is then forced to go through the parent module, rather being pushed straight into the child modules. The parent module will expose an appropriate set of parameters at the level of abstraction it decides. For example, it might be that a parent module takes care of complex interdependencies between clock settings in its children, and all it exposes to the outside world is a single master clock setting.

14 Functional Coverage Tools

Functional coverage tools need a means of accessing and tracking the values of parameters, since parameters obviously affect the setup and behavior of a system.

Awareness of Parameter Data Types

Configuration interfaces are provided for both the case when a parameter's data type is known as well as when it is not.

1 Data Type Aware Access

This type of access has a priori knowledge about the configuration parameter's underlying data type. This will be used for example when a module accesses one of "its own" parameters. A well informed higher-level model could also use this type of access. POs provide this type of access.

2 Data Type Independent Access

This type of access generally knows little more than the configuration parameter's name. Value information is encoded/decoded using a convenient type which incurs some runtime overhead (refer to R19, *Portable Parameter Value* Encoding, for more details). This method will be used by configurators and by models that don't know the parameter's data type. Name-based access functions and parameter handles provide this type of access.

An example to illustrate the importance of this data access type is a utility that processes the contents of a proprietary configuration file format. Since the format is proprietary, support cannot be assumed across multiple tools so a configurator must accompany the model(s). Suppose that this format consists of name-value pairs, both represented as strings. The utility would extract the name string and attempt to set the corresponding parameter's value with the value string. This will require that a data type translation be performed prior to consumption by a (type-aware) PO.

Requirements

This section describes the detailed requirements for configuration interfaces. There is an implied requirement for these interfaces to be simple, efficient and extensible.

1 Access to Parameters

Parameter accessibility is provided in the following ways which are ordered by efficiency of access (fastest to slowest).

1.1 Declared Parameter Object (PO)

These objects are used to declare parameters within models; they have data type information and may subsequently be used to access parameter values and all other parameter information.

1.2 Parameter Handle (PH)

A parameter handle is initially retrieved using the parameter's name and from that point forward can be used for non-value accesses, such as obtaining value-change callbacks, with the same efficiency as a parameter object. Value accesses carry the overhead of parameter value encoding but still avoid the overhead of parameter lookups.

1.3 Name-Based Parameter Access

Name-based parameter access first performs a parameter lookup then performs the requested access. This will primarily be used to obtain parameter handles but is also convenient for other one-time accesses and therefore additionally supports value and notification accesses. All value accesses require parameter value encoding.

2 Parameter Naming and Mapping

A parameter name identifies a NVP. Parameter names reside in hierarchical namespaces and there are no constraints on the parameter hierarchy; specifically, it need not align with the SystemC module hierarchy.

A recommendation will be provided for aligning parameter hierarchy with SystemC module hierarchy when desired.

Parameter names provide the basis for mapping POs and PHs to NVPs.

2.1 Direct Mapping

Direct mapping refers to the case that the fully qualified name of a PO matches the name of the NVP it is mapped to. For example, a PO "foo" of module instance "a.b.c" maps to the NVP with the name "a.b.c.foo".

Direct mapping is a very common use case and, hence, should be very easy to use.

2.2 Custom Mapping

Custom mappings from PO/PH to NVP need to be supported, where names are translated from one namespace to another.

2.3 Many-to-one Mappings (PO/PH to NVP)

Many-to-one mappings from PO/PH to NVP are supported. Since NVP values are interpreted in a type-specific manner by individual POs, it is possible to map POs of different data types to the same NVP.

It is possible for a PO to update a NVP in a manner that violates the policy of another PO mapped to that NVP, in which case the latter can issue a warning or error. For instance, one of the POs can be locked while another PO, mapped to the same NVP, updates the value.

3 Parameter Searches

It will be possible to search for either (a) all visible parameters or (b) just those at a particular scope. The results can be subsequently processed to provide more sophisticated queries, such as those based on regular expressions, but this will not be a part of the standard.

Note: the preliminary intent is to return an STL-like iterator in response to parameter searches for ease of use and independence from underlying container representations.

3.1 Detecting Unconsumed Parameters

In order to support the identification of unconsumed (not yet mapped and yet unread) NVPs, queries will be provided to (a) determine if the NVP's value has been read and (b) obtain an iterator over the POs mapped to it.

4 Parameters Types

There are three types of configuration parameters: immutable, elaboration-time, and mutable.

4.1 Immutable Parameters

Immutable parameters have a fixed value this is governed by the modeler (i.e. they cannot be updated using a tool); these parameters can be created at any time. This parameter type exists so constant values affecting system configuration can be presented to the user in a manner that is consistent with other types of parameters.

4.2 Elaboration-Time Parameters

Elaboration-time parameters are mutable until end of elaboration, and then they become immutable. An elaboration-time parameter created after end of elaboration is effectively immutable. Elaboration-time parameters typically influence the design structure.

Examples of things that might be represented by elaboration-time parameters are the number of ports in a USB controller or the number of channels in a memory controller.

4.3 Mutable Parameters

Mutable parameters can be created at any time and their values can be updated throughout the simulation.

Examples of things that might be represented by mutable parameters are the name of an output file, model modes, hardware latency parameters, or logging control.

5 Parameter Lifetime

NVPs are never deleted, and no default modifications result from the last PO mapping being deleted. The application is free however to orchestrate other policies, such as restoring an NVP's end_of_elaboration value.

6 Superset Parameter Value Specification

It shall be possible to specify parameter values, on a system level, for parameters that are never used in a particular simulation. This makes it possible to use a single parameter set (for example, fixed in a configuration file) for different subsets of a system.

7 Parameter Information

For the benefit of users, POs shall carry information that a user and tool can access; e.g. documentation, default value, etc. The contents of the documentation are up to the modeler to determine, but it shall at minimum be human-readable and guide in the setting of parameter values.

8 Parameter Default Values

It shall be possible to specify a parameter's default value to be assumed until otherwise set. The default value has lowest priority when evaluating overrides.

Querying an unset parameter that has no default value will yield a response that explicitly indicates the value is uninitialized.

9 Parameter Initial Value Availability

It needs to be well-defined when the initial set of NVPs provided by the tool will be available. Initial values need to be available no later than the start of elaboration (i.e. start of sc_main).

10 CCI API Availability

It needs to be well-defined when the CCI API will be available during the execution of a SystemC application. The CCI API needs to be available no later than the start of elaboration (i.e. start of sc_main).

11 Parameter Value's Origin

It shall be possible to determine if a parameter's value is uninitialized, the default, or a value that has been explicitly set (by a model or user).

It shall be possible to determine what module or process set the value of each parameter and what module or process last read the value of each parameter.

12 Precedence for Overriding Parameter Values

It shall be possible to override the value of a parameter. The precedence for competing parameter value overrides will be well-defined. This may take into account the originator's hierarchical level (i.e. testbench -> platform -> subsystem -> component), the order of execution and/or additional factors.

13 Notification on Parameter Value Change

It shall be possible for arbitrary code to track the changes to a parameter, and perform some processing in response to it. Such changes can come both from model-internal activity (for example, a write to a configuration register from target software) or model-external activity (for example, a tool changing a parameter to force an error condition).

14 Locking Parameter Values

The standard shall facilitate the locking and unlocking of parameters such that a locked PO's value(s) cannot be changed. That is, a mutable PO can be changed to temporarily behave like an immutable PO. This does not restrict updates to the mapped NVP that are made by other (presumably unlocked) POs.

15 Rejection of Access to a Parameter's Value(s)

Parameters must have the ability to indicate an attempt to read or write their values has failed. Failure determination may be made by the parameter itself or indirectly in response to a value change notification.

The access rejection mechanism will explicitly support common designations such as private, read-only, restricted access and value constrained parameters. Non-specific (i.e. catch-all) rejections will also be supported to allow user-defined reasons.

16 Indicate Violation of Inter-Parameter Value Constraints

It will be possible for a parameter to accept a new value but indicate that it violates established inter-parameter constraints.

Whereas Requirement 15 provides for the outright rejection a new value in violation of the parameter's individual policies/constraints, this requirement allows the cautionary acceptance of new values that violate policies/constraints that span more than one parameter.

This will be useful e.g. for checking the consistency of a group of related parameters. As individual parameters are updated, the collective intermediate state may become invalid in which case a constraint violation shall be reported (informing which parameters are related and the constraints on their values). Once a constraint violation has been reported, it will be recommended that subsequent resolution of the violation also be indicated, to inform the user that a valid system state has been attained.

17 Models Can Configure Other Models

It shall be possible for models to configure the parameters of other arbitrary models. For example, a subsystem could expose high-level semantic parameters (configure me for chip variants "A" or "B") and create the corresponding parameters for its models.

Note that any of the parameter access methods may be used for this, depending on how familiar the configuring model is with the configured model's parameters.

18 Supported Data Types

There will be no restrictions on the data types of parameters; i.e. any legal C++ data type, including user-defined types, is permitted. Un-typed references, such as (void *), may not be supported.

Full support will be provided for std::string, sc_time and common C++ and SystemC numeric types. User-defined types will require that value encoding (see below) be provided.

19 Portable Parameter Value Encoding

A well-defined mechanism will be provided for encoding the type and data of a parameter value and it will be possible to get and set a parameter's value using this encoding without a priori knowledge of its inherent type.

The encoding will be based on JSON which uses the following type model (from http://www.json.org):

- Number (integer, real, or floating point), which can encode 64-bit integers
- String (double-quoted Unicode with backslash escaping)
- Boolean (true and false)
- Array (an ordered sequence of values, comma-separated and enclosed in square brackets) – *used to encode arbitrary lists, including nested lists.*
- Object (collection of key:value pairs) – *not likely to be used in CCI*
- null

The JSON type model will be extended to support the following SystemC simulation-specific data types:

- sc_time
- Binary data blobs, to efficiently represent large bodies of binary data without having to use lists of integers, which both unnatural and inefficient.

Arbitrary nested lists of values are needed in order to support the encoding of arbitrarily complex data. Lists offer a way to describe complex and variable-length configuration parameters such as memory maps and interrupt-pin assignments in an implementation-independent and general way. It also offers a simple way to encode the content of C structs and C++ classes, as they are essentially collections of basic types and thus naturally map to lists.

Using a defined set of types has the advantage that any tool can configure any parameter in a type-aware way. It also ensures interoperability between models from different sources, as they all use the same pre-known set of types to encode their configuration data.

20 Documentation

20.1 LRM

A preliminary LRM strength document should accompany the first official release of a configuration interface standard.

20.2 Overview Presentation

An introductory overview presentation should accompany both community review and official release packages.

20.3 Design (w/UML) & Rationale

A UML-based interface design description should accompany both community review and official release packages.

20.4 Examples walkthroughs

Walkthroughs of key examples should be provided for both community review and official release packages.

Notes

This section covers relevant topics, which – while relevant to the discussion of Configuration Interfaces – neither constitutes a strict requirement nor a use model.

- Custom mapping of POs/PHs to NVPs will not be explicitly supported in the CCI API. Searches necessary to support this will be provided. Iterating over all NVPs will be the catch-all, but specific support will be provided for searches we anticipate to be commonly used, such as bottom-up.
- It is likely that CCI API will not yet be available during static initialization (before main()) and will no longer be available during static destruction (after main()).

Future Plans

Addressing save/restore will have eventual implications on configuration parameters. Once an initial definition of the save/restore standard interface is complete, the configuration parameters will be revisited and enhanced to include save/restore support.

Requirements Summary