December 19, 2017

Draft Standard SystemC® CCI 1.0 Language Reference Manual

Description

This is the SystemC Configuration, Control & Inspection (CCI) Language Reference Manual, draft version 1.0.

Keywords

Accellera Systems Initiative, SystemC, Configuration

NOTICE

This draft standard is a work in progress and subject to change.

Notice

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and comply with them.

Disclaimer

Trademarks

Bugs and Suggestions

Please report bugs and suggestions about this document to:

cci-review@lists.accellera.org

Introduction

This document defines the standard for the SystemC Configuration, Control & Inspection (CCI) draft version 1.0.

Contributors

The development of the SystemC Configuration Language Reference Manual was sponsored by the Accellera Systems Initiative and was created under the leadership of the following people:

Trevor Wieman, Intel (SystemC Configuration, Control and Inspection Working Group Chair)
Bart Vanthournout, Synopsys (SystemC Configuration, Control and Inspection Working Group Vice-Chair)

While developing this standard, the SystemC Configuration, Control and Inspection working group had the following membership:

Daniel Aarno, Intel
Sergei Ananko, Intel
George Andre, Intel
Guillaume Audeon, ARM
John Aynsley, Doulos
Laurent Bernard, ST
Bishnupriya Bhattacharya, Cadence
David Black, Doulos
Bill Bunton, Intel
Mark Burton, GreenSocs
Sheshadri Chakravarthy, TI
Somarka Chakravarti, Intel
Ying-Tsai Chang, Synopsys
Bryan Coish, Intel
Zaitrarrio Collier, Intel
Jerome Cornet, ST
Ola Dahl, Ericsson
Samik Das, Cadence
Guillaume Delbergue, Ericsson
Ajit Dingankar, Intel
Jakob Engblom, Intel
Alan Fitch, Ericsson
Eric Frejd, Ericsson
Enrico Galli, Intel
Vishal Goel, TI
Thomas Goodfellow, OFFIS
Karthick Gururaj, Vayavya
Philipp A Hartmann, Intel
Gino Hauwermeiren, Synopsys
Tor Jeremiassen, TI
Chandra Katuri, Cadence
Suresh Kumar, ST
Erik Landry, Intel
Glenn Leary, Intel
Lei Liang, Ericsson
Hans-Peter Loeb, Cadence
David Long, Doulos
Laurent Maillet-Contoz, ST
Svetlin Manavski, ARM
Scott McMahon, Intel
Victoria Mitchell, Intel
Asif Mondal, TI
Indraneel Mondal, Synopsys
Vincent Motel, Cadence
Rajiv Narayan, Qualcomm
Blake Nicholas, Intel
Ahmed Nizamudheen, TI
Atanas Parashkevov, Australian Semi
P V S Phaneendra, CircuitSutra
Sonal Poddar, Intel
Udaya Ranga, TI
Abhishek Saksena, Intel
Rauf Salimi Khaligh, Intel
Christian Sauer, Cadence
Martin Schnieringer, Bosch
Ravi Singh, Intel
Gary Snyder, Intel
Henrik Svensson, Ericsson
Ramachandran Swaminathan, TI
Yossi Veller, Mentor
CD Venkatesh, Intel
Girish Verma, CircuitSutra
Eric Wallace, Intel
Andy Walton, Intel
Jesus Yuriar, Intel
Hakan Zeffer, Intel

Contents

1. Introduction

This document defines the SystemC Configuration standard as a collection of C++ Application Programming Interfaces (APIs) layered on top of the SystemC language standard; familiarity with the existing ISO C++ and IEEE 1666 SystemC standards is presumed. Configuration represents the first phase of CCI standardization.

SystemC Configuration represents phase one of the Configuration, Control and Inspection (CCI) family of standards for model-to-tool interoperability. The primary use case is configuring variable properties of the structure and behavior of a model. This standard will ensure configurability of SystemC models from different providers and promote a consistent user experience across compliant tools.

Stakeholders in SystemC Configuration include suppliers of electronic components and systems using SystemC to develop configurable models of their intellectual property, and Electronic Design Automation (EDA) companies that implement SystemC Configuration class libraries and supporting tools.

This standard is not intended to serve as a user's guide or to provide an introduction to SystemC Configuration. Readers requiring a SystemC Configuration tutorial or information on its intended use should consult the Accellera Systems Initiative web site (www.accellera.org) to locate available resources.

2. Overview

2.1 Scope

This standard defines SystemC® Configuration as an ANSI standard C++ class library used to make SystemC models configurable. The standard does not specify a file format for specifying configuration parameter values.

2.2 Purpose

The general purpose of SystemC Configuration is to provide a standard for developing configurable SystemC models.

The specific purpose of this standard is to provide precise and complete definitions of (1) the Configuration class library and (2) the interfaces necessary to implement brokers and to integrate existing parameter solutions. This standard is not intended to serve as a user's guide or to provide an introduction to SystemC Configuration.

2.3 Relationship with C++

This standard is closely related to the C++ programming language and adheres to the terminology used in ISO/IEC 14882:2014. This standard does not seek to restrict the usage of the C++ programming language; an application using the SystemC Configuration standard may use any of the facilities provided by C++, which in turn may use any of the facilities provided by C. However, where the facilities provided by this standard are used, they shall be used in accordance with the rules and constraints set out in this standard.

This standard presumes that C++11 is the minimum revision supported and makes use of features of that revision such as move semantics and `=delete` to selectively disable default methods. Implementations may choose to support earlier revisions such as C++03 by hiding or approximating such features, however they are not required to do so.

This standard defines the public interface to the SystemC Configuration class library and the constraints on how those classes may be used. The class library may be implemented in any manner whatsoever, provided only that the obligations imposed by this standard are honored.

A C++ class library may be extended using the mechanisms provided by the C++ language. Implementers and users are free to extend SystemC Configuration in this way, provided that they do not violate this standard.

NOTE - It is possible to create C++ programs that are legal according to the C++ programming language standard but violate this standard. An implementation is not obliged to detect every violation of this standard.

2.4 Relationship with SystemC

This standard is built on the IEEE Std 1666™-2011 (SystemC Language Reference Manual) and extends it using the mechanisms provided by the C++ language, to provide an additional layer of configuration constructs.

2.5 Guidance for readers

Readers who are not familiar with SystemC Configuration should start with Annex A, "Introduction to SystemC Configuration," which provides a brief informal summary intended to aid in the understanding of the normative definitions. Such readers may also find it helpful to scan the examples embedded in the normative definitions and to see 0, "Glossary."

Readers should pay close attention to Clause 3, "Terminology and conventions used in this standard." An understanding of the terminology defined in Clause 3 is necessary for a precise interpretation of this standard.

Clause 4 defines the public interface to the SystemC Configuration class library. The following information is listed for each class:

a) A brief class description.
b) A C++ source code listing of the class definition.
c) A statement of any constraints on the use of the class and its members.
d) A statement of the semantics of the class and its members.
e) For certain classes, a description of functions, typedefs, and macros associated with the class.
f) Informative examples illustrating typical uses of the class.

Annex A is intended to aid the reader in understanding the structure and intent of the SystemC Configuration class library.

0 provides recommended guidelines for effectively using this standard.

C.1 describes how to enable the use of user-defined types with configuration parameters.

0 is a glossary giving informal descriptions of the terms used in this standard.

2.6 Reference documents

The following documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the document (including any amendments or corrigenda) applies.

This standard shall be used in conjunction with the following publications:

- ISO/IEC 14882:2014, Programming Languages – C++
- IEEE Std 1666-2011: IEEE Standard SystemC Language Reference Manual

3. Terminology and conventions used in this standard

3.1 Terminology

3.1.1 Shall, should, may, can

The word *shall* is used to indicate a mandatory requirement.

The word *should* is used to recommend a particular course of action, but it does not impose any obligation.

The word *may* is used to mean shall be permitted (in the sense of being legally allowed).

The word *can* is used to mean shall be able to (in the sense of being technically possible).

In some cases, word usage is qualified to indicate on whom the obligation falls, such as *an application may* or *an implementation shall*.

3.1.2 Application, implementation

The word *application* is used to mean a C++ program, written by an end user, that uses the SystemC Configuration class library, that is, uses classes, functions, or macros defined in this standard.

The word *implementation* is used to mean any specific implementation of the SystemC Configuration class library as defined in this standard, only the public interface of which need be exposed to the application.

3.1.3 Call, called from, derived from

The term *call* is taken to mean call directly or indirectly. Call indirectly means call an intermediate function that in turn calls the function in question, where the chain of function calls may be extended indefinitely.

Similarly, *called from* means called from directly or indirectly.

Except where explicitly qualified, the term *derived from* is taken to mean derived directly or indirectly from. Derived indirectly from means derived from one or more intermediate base classes.

3.1.4 Specific technical terms

The specific technical terms as defined in IEEE Std 1666-2011 (SystemC Language Reference Manual) also apply for the Configuration standard. In addition, the following technical terms are defined:

A *parameter* is a class derived from the class **cci::cci_param_if**.

A *broker* is a class derived from the class **cci::cci_broker_if**.

3.2 Syntactical conventions

3.2.1 Implementation-defined

The italicized term *implementation-defined* is used where part of a C++ definition is omitted from this standard. In such cases, an implementation shall provide an appropriate definition that honors the semantics defined in this standard.

3.2.2 Ellipsis (...)

An ellipsis, which consists of three consecutive dots (...), is used to indicate that irrelevant or repetitive parts of a C++ code listing or example have been omitted for brevity.

3.2.3 Non-public class names

Class names italicized and annotated with a superscript dagger (†) should not be used explicitly within an application. Moreover, an application shall not create an object of such a class. It is strongly recommended that the given class name be used. However, an implementation may substitute an alternative class name in place of every occurrence of a particular daggered class name.

Only the class name is considered here. Whether any part of the definition of the class is implementation-defined is a separate issue.

The non-public class names in this standard are:

- `cci_value_cref`
- `cci_value_ref`
- `cci_value_list_cref`
- `cci_value_list_ref`
- `cci_value_map_cref`
- `cci_value_map_ref`
- `cci_value_string_cref`
- `cci_value_string_ref`

Public typedefs are provided for these classes to avoid the need for clients to refer to them directly.

3.2.4 CCI naming patterns

The CCI interfaces are denoted with the prefix `cci_` for classes, functions, global definitions and variables, and with the prefix `CCI_` for macros and enumeration values. The namespace itself is simply `cci`.

An application shall not make use of these prefixes when defining classes, functions, global definitions, global variables, macros, and enumerations.

An implementation may nest further namespaces within `namespace cci`, but such nested namespaces shall not be used in applications.

3.3 Typographical conventions

The following typographical conventions are used in this standard:

1. The *italic* font is used for:
   - Cross references to terms defined in Subclause 3.1, "Terminology", Subclause 3.2, "Syntactical conventions", and Annex B, "Glossary".
   - Arguments of member functions in class definitions and in the text that are generally substituted with real values by the implementation or application.
2. The **bold** font is used for all reserved keywords of SystemC and the Configuration standard as defined in namespaces, macros, constants, enum literals, classes, member functions, data members and types.
3. The `constant-width` (Courier) font is used:
   - for SystemC Configuration class definitions including member functions, data members and data types.
   - to illustrate SystemC Configuration examples when the exact usage is depicted.
   - for references to the SystemC Configuration language syntax and headers.

The conventions listed previously are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

3.4 Semantic conventions

3.4.1 Class definitions and the inheritance hierarchy

An implementation may differ from this standard in that an implementation may introduce additional base classes, class members, and friends to the classes defined in this standard. An implementation may modify the inheritance hierarchy by moving class members defined by this standard into base classes not defined by this standard. Such additions and modifications may be made as necessary in order to implement the semantics defined by this standard or in order to introduce additional functionality not defined by this standard.

3.4.2 Function definitions and side-effects

This standard explicitly defines the semantics of the C++ functions in the SystemC Configuration class library. Such functions shall not have any side-effects that would contradict the behavior explicitly mandated by this standard. In general, the reader should assume the common-sense rule that if it is explicitly stated that a function shall perform action A, that function shall not perform any action other than A, either directly or by calling another function defined in this standard. However, a function may, and indeed in certain circumstances shall, perform any tasks necessary for resource management, performance optimization, or to support any ancillary features of an implementation. As an example of resource management, it is assumed that a destructor will perform any tasks necessary to release the resources allocated by the corresponding constructor.

3.4.3 Exceptions

Other than destructors and `swap()` or as explicitly noted in documentation, API functions should be presumed to have the potential to throw exceptions, either as the `SC_THROW` action from `sc_report_handler::report()` diagnostic or an explicit `throw`. Callback functions are also permitted to `throw`. Implementations shall ensure that class invariants are preserved in the case of exceptions from all sources. The utility function `cci_handle_exception()` decodes cci framework exceptions using `cci_param_failure` enum values as described in clause 5.8, “Error reporting”.

3.4.4 Functions whose return type is a reference or a pointer

An object returned from a function by pointer or by reference is said to be valid during any period in which the object is not deleted and the value or behavior of the object remains accessible to the application. If an application refers to the returned object after it ceases to be valid, the behavior of the implementation shall be undefined.

3.4.5 Functions that return *this or a pass-by-reference argument

In certain cases, the object returned is either an object (***this**) returned by reference from its own member function (for example, the assignment operators), or it is an object that was passed by reference as an argument to the function being called. In either case, the function call itself places no additional obligations on the implementation concerning the lifetime and validity of the object following return from the function call.

3.4.6 Functions that return const char*

Certain functions have the return type **const char*** indicating they return a pointer to a null-terminated character string. Such strings shall remain valid until returning from `sc_main()`.

3.4.7 Non-compliant applications and errors

In the case where an application fails to meet an obligation imposed by this standard, the behavior of the implementation shall be undefined in general. When this results in the violation of a diagnosable rule of the C++ standard, the C++ implementation will issue a diagnostic message in conformance with the C++ standard.

When this standard explicitly states that the failure of an application to meet a specific obligation is an *error* or a *warning*, the implementation shall generate a diagnostic message by calling an appropriate function in `cci_report_handler`; for common CCI error types the specific diagnostics such as `set_param_failed()`, and for other errors or warnings `report()`. In the case of an *error*, the implementation shall call function `report()` with a severity of SC_ERROR. In the case of a *warning*, the implementation shall call `report()` with a severity of SC_WARNING. See clause 5.8, “Error reporting”, for details of `cci_report_handler`.

An implementation or an application may choose to suppress run-time error checking and diagnostic messages because of considerations of efficiency or practicality. For example, an application may call member function **set_actions** of class **sc_report_handler** to take no action for certain categories of report. An application that fails to meet the obligations imposed by this standard remains in error.

There are cases where this standard states explicitly that a certain behavior or result is *undefined*. This standard places no obligations on the implementation in such a circumstance. In particular, such a circumstance may or may not result in an *error* or a *warning*.

3.5 Notes and examples

Notes appear at the end of certain subclauses, designated by the uppercase word NOTE. Notes often describe the consequences of rules defined elsewhere in this standard. Certain subclauses include examples consisting of fragments of C++ source code. Such notes and examples are informative to help the reader but are not an official part of this standard.

4. CCI architecture overview

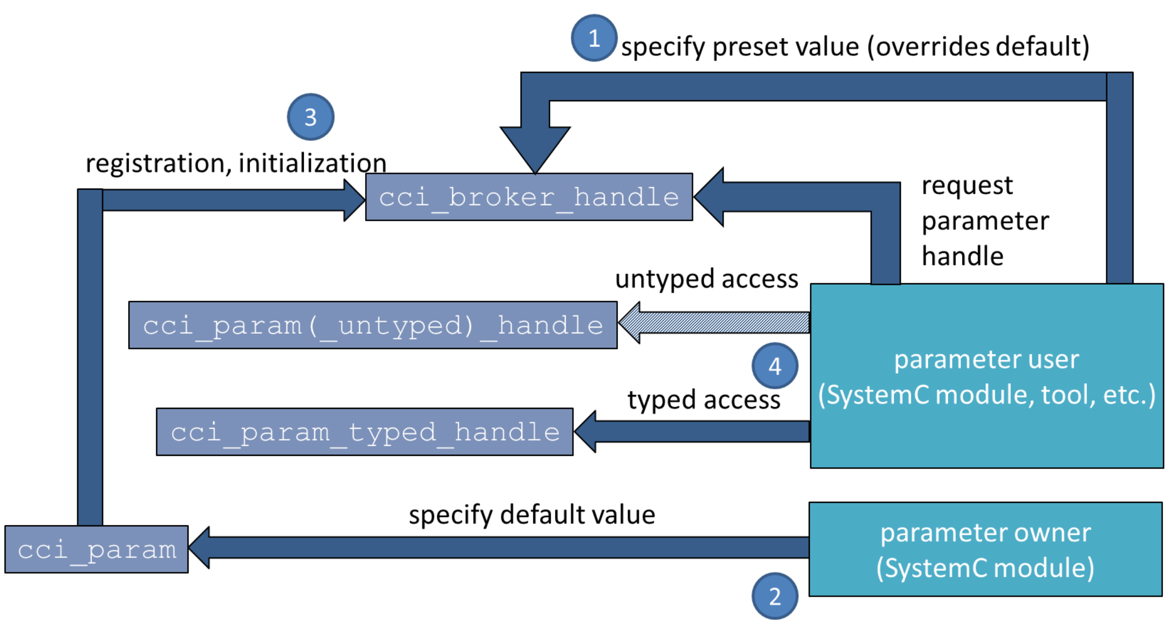
The core of the SystemC Configuration standard is the pairing of parameters and brokers, where a parameter is a named instance of a specific compile-time type and a broker aggregates parameters and provides access to them in the form of handles.

Each parameter is registered during construction with a single broker. Parameters are typically constructed and owned by a SystemC module, with other users subsequently obtaining a handle from the broker. The owner constructs a parameter with a default value, however the broker can override this with a preset value, allowing tools to provide runtime configurations.

Typically a global broker will exist, created early in the elaboration phase. Sub-modules may supply their own local brokers, for example to keep their parameters private. In such a case a sparse hierarchy of brokers mirrors the hierarchy of `sc_modules`.

Loose coupling is supported by the separation of owner and user lifetimes: because handles remain minimally valid objects after their parameter is destroyed, undefined behavior is avoided while retaining strong ownership principles. The following diagram shows a sequence of a parameter being constructed and used:

1. A tool obtains a broker handle and specifies a preset value for the named parameter
2. The module owning the parameter instantiates it with a default value
3. The new parameter registers with the broker and acquires the preset value, supplanting the default
4. A user gets a handle for the parameter and through it gets the current (i.e. preset) value.



Central to these interactions is the `cci_broker_handle`, commonly obtained from `cci_get_broker()`. Typically a broker will be constructed early in the elaboration phase and will serve as the default "global" broker. By providing their own "local" brokers sub-modules can keep their parameters effectively private, since there is no public interface for enumerating brokers; a practical configuration tool will provide its own mechanism for presetting parameters on brokers.

5. Configuration interfaces

5.1 Namespaces

All SystemC Configuration classes, functions and enumeration values shall reside inside the namespace `cci`. Generally this document omits the namespace qualification for brevity, so identifiers and code samples have an implicit `using namespace cci`.

5.2 Configuration header file

To use SystemC Configuration class library features, an application shall include the top-level C++ header file at appropriate positions in the source code as required by the scope and linkage rules of C++.

#include “cci_configuration”

The header file **cci_configuration** shall add the name **cci**, as well as the names defined in IEEE Std 1666™-2011 for the header file named **systemc**, to the declarative region in which it is included. The header file **cci_configuration** shall not introduce into the declarative region, in which it is included, any other names from this standard or any names from the standard C or C++ libraries.

Example:

```
#include “cci_configuration”
using cci::cci_param;
...
```

5.3 Enumerations

5.3.1 cci_param_mutable_type

Enumeration for the `cci_param_typed` template specifying the mutability of the parameter:

- `CCI_MUTABLE_PARAM = 0` – parameter is mutable and can be modified, unless locked
- `CCI_IMMUTABLE_PARAM` – parameter is immutable, having either the default value it was constructed with or a preset value configured through the broker
- `CCI_OTHER_MUTABILITY` – Vendor specific mutability control

Mutability forms part of the concrete parameter type as an argument of the `cci_param_typed` template.

5.3.2 cci_param_data_category

Enumeration for the general category of a parameter’s value type; used when details of its specific type are not required.

- `CCI_BOOL_PARAM` – boolean valued parameter
- `CCI_INTEGRAL_PARAM` – integer valued parameter
- `CCI_REAL_PARAM` – real number valued parameter
- `CCI_STRING_PARAM` – string valued parameter
- `CCI_LIST_PARAM` – list valued parameter
- `CCI_OTHER_PARAM` – parameter with values of any other type

5.3.3 cci_name_type

Specifies whether the name used in constructing a parameter is to be emplaced hierarchically:

- `CCI_RELATIVE_NAME` – emplaced by prepending with the name of the enclosing sc_module, e.g. parameter "p" as a member of sub-module "sub" of top-level module "m" will have the full name "m.sub.p"
- `CCI_ABSOLUTE_NAME` – the name isn't modified

In either case the name is required to be unique and if necessary will be modified to make it so as described in clause 5.9, "Name support functions".

5.4 Core interfaces

5.4.1 cci_originator

Originators are used to track the entity responsible for a parameter's current value. An originator can be identified either as an `sc_object` (either explicitly, or implicitly from the current module context), or in the absence of any `sc_object`, by a given name.

```
class cci_originator
{
public:
    inline cci_originator();
    inline cci_originator(const sc_core::sc_object& originator);
    cci_originator(const std::string& originator_name);
    explicit cci_originator(const char *originator_name);

    // Copy constructors
    cci_originator(const cci_originator& originator);
    cci_originator(cci_originator&& originator);

    ~cci_originator();

    const sc_core::sc_object* get_object() const;

    // Returns the parent originator
    cci_originator get_parent_originator() const;

    // Returns the name of the current originator
    const char* name() const;

    // Operator overloads
    cci_originator& operator=( const cci_originator& originator );
    cci_originator& operator=( cci_originator&& originator );
    bool operator==( const cci_originator& originator ) const;
    bool operator<(const cci_originator& originator) const;

    // Swap originator object and string name with the provided originator.
    void swap(cci_originator& that);

    // Returns the validity of the current originator
    bool is_unknown() const;
};
```

5.4.1.1 Construction

```
cci_originator();
```

Use the current `sc_object` as the originator object. If there is no such object, for example if called outside the context of the running simulation or outside the module hierarchy then an error report is issued; to resolve this one of the other constructor forms taking an argument must be used.


```
cci_originator(const sc_core::sc_object& originator);
```

Use the explicitly-given originator object. Using this interface permits the caller to hide or abstract their architecture, for example using a parent `sc_module` when constructing a cci_originator in a child `sc_module`. In this case both parent and child modules have constructed `cci_originator`s with the same identity.

```
cci_originator(const std::string& originator_name);
explicit cci_originator(const char *originator_name);
```

Construct an originator with the explicit name. Note that if a current `sc_object` is available (e.g. during simulation time or construction of an `sc_module`) then the explicit name will be ignored and the current `sc_object` name will be reported as the `cci_originator::name()`. This reflects the preference for object identify over simple names.

Example:

```
SC_CTOR(test_module) {
    cci_originator orig("foo");
    sc_assert( orig.get_object() );
    sc_assert( orig.name() != "foo" );
```

```
cci_originator(const cci_originator& originator);
cci_originator(cci_originator&& originator);
```

Copy and move constructors, initializing the object and name from the source. After a move the source `cci_originator` has a diagnostic "unknown" name and `is_unknown()` returns `true`.

5.4.1.2 Copy and swap

```
cci_originator& operator=( const cci_originator& originator );
cci_originator& operator=( cci_originator&& originator );
```

Copy and move assignments, initializing the object and name from the source. After a move the source `cci_originator` has a diagnostic "unknown" name and `is_unknown()` returns `true`.

```
void swap(cci_originator& that);
```

Swaps the current `cci_originator` object and name with those of the provided "that" `cci_originator`, with guaranteed exception safety.

5.4.1.3 Identity

```
const sc_core::sc_object* get_object() const;
```

Returns the originator object pointer.

```
const char* name() const;
```

Returns the name of the originator. When an originator object has been supplied or located then its `sc_object::name()` is returned, otherwise the explicit name with which the originator was constructed. The returned pointer is non-owning and may be invalidated by the originator's destruction.

```
bool is_unknown() const;
```

Returns `true` if no object or name is defined. Such a state is only likely where the object was the source of a move operation because `cci_originator` reports an error if neither an originator object nor any name is given.

Example:

```
cci_originator o1;
sc_assert( !o1.is_unknown() );
cci_originator o2( std::move(o1) );
sc_assert( o1.is_unknown() );
```

5.4.1.4 Comparisons

```
bool operator==( const cci_originator& originator ) const;
bool operator<(const cci_originator& originator) const;
```

Comparisons devolve to string comparisons of the `name()`s. This means originators constructed using the same sc_object context will match, reflecting again the precedence that object identity has over simple names.

Example:

```
SC_CTOR(test_module) {
cci_originator o1("foo");
cci_originator o2("bar");
sc_assert( o1 == o2 );
}
```

5.4.2 cci_param_if

The basic parameter interface class, providing metadata and variant value access. Concrete descendant classes such as `cci_param_typed` provide implementations. In particular the `cci_param_typed` class provides both the definition of the underlying data type and the instantiable object.

```
class cci_param_if : public cci_param_callback_if
{
public:
    // Get and set cci_value
    cci_value get_cci_value() const;
    virtual cci_value get_cci_value(const cci_originator& originator) const = 0;
    void set_cci_value(const cci_value& val);
    void set_cci_value(const cci_value &val, const cci_originator &originator);
    virtual void set_cci_value(
        const cci_value &val, const void *pwd, const cci_originator &originator) = 0;
    virtual void reset(const cci_originatr& originator) = 0;
    virtual cci_value get_default_cci_value() const = 0;

    // Value type
    virtual cci_param_data_category get_data_category() const = 0;
    virtual const std::type_info& get_type_info() const = 0;

    // Value origin
    virtual bool is_default_value() const = 0;
    virtual bool is_preset_value() const = 0;
    virtual cci_originator get_originator() const = 0;
    virtual cci_originator get_latest_write_originator() const = 0;

    // Name and description
    virtual const std::string &get_name() const = 0;
    virtual std::string get_description() const = 0;
    virtual void set_description(const std::string &desc) = 0;

    // Metadata
    virtual void add_metadata(const std::string &name, const cci_value &value,
                              const std::string &desc = "") = 0;
    virtual cci_value_map get_metadata() const = 0;

    // Value protection
    virtual cci_param_mutable_type get_mutable_type() const = 0;
    virtual bool lock(const void *pwd = NULL) = 0;
    virtual bool unlock(const void *pwd = NULL) = 0;
    virtual bool is_locked() const = 0;

    // Equality
    virtual bool equals(const cci_param_if &rhs) const = 0;

    // Handle creation
    virtual cci_param_untyped_handle
            create_param_handle(const cci_originator &originator) const = 0;

protected:
    virtual ~cci_param_if();
    void init(cci_broker_handle broker);
```



```
    void destroy(cci_broker_handle broker);

private:
    virtual void preset_cci_value(const cci_value& value, const cci_originator& originator);
    virtual void invalidate_all_param_handles();

    // Get and set raw value
    virtual void set_raw_value(const void *vp, const void *pwd,
                               const cci_originator &originator) = 0;
    virtual const void *get_raw_value(const cci_originator &originator) const = 0;
    virtual const void *get_raw_default_value() const = 0;

    virtual void add_param_handle(cci_param_untyped_handle* param_handle) = 0;
    virtual void remove_param_handle(cci_param_untyped_handle* param_handle) = 0;
};
```

5.4.2.1 Value and data type

The parameter value is handled via the variant type `cci_value`. Statically-typed access is provided by the descendant `cci_param_typed` and matching `cci_param_typed_handle` classes.

```
cci_value get_cci_value() const;
cci_value get_cci_value(const cci_originator& originator) const
```

Returns a copy of the current value. The `originator` value identifies the context for pre- and post-read callbacks. If none provided, the parameter's own originator (typically the owning module) is used.

```
void set_cci_value(const cci_value& val);
void set_cci_value(const cci_value &val, const cci_originator &originator);
void set_cci_value(const cci_value &, const void* pwd, const cci_originator& orig);
```

Sets the parameter to a copy of the given value, applying the given password. A `NULL` password is used if none is provided. If no originator is provided, the parameter's own originator is used. If the variant value cannot be unpacked to the parameter's underlying data type then a `CCI_VALUE_FAILURE` error is reported.

```
void reset(const cci_originator &orig);
```

Sets the value back to the initial value the parameter took, i.e. the preset value if one was defined or the default value with which it was constructed. Any pre-write callbacks are run before the value is reset, followed by any post-write callbacks, and finally the latest write originator is set to the given originator.

```
cci_value get_default_cci_value();
```

Returns a copy of the default value the parameter was constructed with.

```
cci_param_data_category get_data_category();
```

Returns the parameter's underlying data category.

```
const std::type_info& get_type_info();
```

Returns the C++ `typeid()` of the parameter's underlying data type.

5.4.2.2 Raw value access

These private methods are accessible only by parameter implementations. They facilitate the exchange of parameter values between arbitrary parameter implementations from levels in the parameter inheritance hierarchy where the specific value type is not known. They provide no type safety.

```
void set_raw_value(const void *vp, const void *password, const cci_originator &originator);
```

Overwrite the stored value with the given value `vp`, where that must point to a valid object of the parameter's underlying data type. In detail:

- `vp` must not be `NULL`
- testing the write-lock state and the `password` validity if locked
- invoking pre-write callbacks with the given originator, aborting the write if callbacks reject it
- copying the value from `vp`
- invoking post-write callbacks with the given originator
- setting the latest write originator

```
const void *get_raw_value(const cci_originator &originator) const;
```

Return a type-punned pointer to the parameter's current value after first invoking the pre-read and then post-read callbacks, both with the given originator.

```
const void *get_raw_default_value() const;
```

Return a type-punned pointer to the parameter's default value.

5.4.2.3 Value origin

Methods to determine the origin of the parameter's current value:

```
bool is_default_value() const;
```

Returns `true` if the current value matches the default value with which the parameter was constructed, using the equality operator of the underlying data type.

NOTE: this is a statement about the current value rather than its provenance; it does not mean that the parameter value is untouched since its construction, simply that the current value matches the default value.

```
bool is_preset_value() const;
```

Returns `true` if the current value matches the preset value set via the parameter's broker using `set_preset_cci_value()`.

- returns `false` if there is no preset value
- the comparison is performed by the equality operator of the underlying data type against the unpacked preset `cci_value`

NOTE: this is a statement about the current value rather than its provenance; it does not mean that the parameter value is untouched since its construction, simply that the current value matches the preset value.

```
cci_originator get_originator() const;
```

Returns a copy of the originator supplied when the parameter was constructed.

```
cci_originator get_latest_write_originator() const;
```

Returns a copy of the originator of the most recent write to the parameter value:

1. the originator supplied as a (possibly default) constructor argument when the parameter was constructed; semantically this is the point where the default value was set
2. the originator supplied if the preset value was set by `cci_broker_if::set_preset_cci_value()`
3. the originator supplied to explicit overloads of `set_cci_value()` and `set_raw_value()`
4. for all indirect writes via methods of `cci_param_if`, the constructor originator (described in case 1)
5. for all writes via methods of `cci_param_untyped_handle` and `cci_param_typed_handle`, the originator given when creating/getting the handle.

5.4.2.4 Name and description

```
const std::string& get_name() const;
std::string get_description() const;
void set_description(const std::string &desc);
```

`get_name()` returns the guaranteed-unique form of the name given when constructing the (typed) parameter. Refer to clause 5.9, “Name support functions”, for name handling details.

A parameter can carry a textual description, given as a `std::string`. Users are encouraged to use this to ensure that parameters are adequately documented, e.g. when enumerated in log files. The description is initialized during construction of the concrete `cci_param_typed` object but can be subsequently updated by `set_description()` and retrieved with `get_description()`.

5.4.2.5 Metadata

```
void add_metadata(const std::string &name, const cci_value &value, const std::string &desc = "");
cci_value_map get_metadata() const;
```

A parameter can carry arbitrary metadata, presented as a `cci_value_map` of `cci_value_list` pairs (`cci_value` value, `std::string` description). Metadata items are added piecewise using `add_metadata()` and cannot be modified or removed since there is no direct access to the underlying map. The metadata is accessed through the return value of `get_metadata()`, which is a deep copy of the metadata (in contrast to the reference returned by `cci_value::get_map()`). This may be a performance consideration if using metadata extensively.

Example:

```
p.add_metadata("alpha", cci_value(2.0));   // description defaulted
p.add_metadata("beta", cci_value("faint"), "Beta description");
cci_value_map meta = p.get_metadata();
cci_value::const_list_reference val = meta["beta"].get_list();
sc_assert(val[0].get_string() == "faint");
sc_assert(val[1].get_string() == "Beta description");
```

5.4.2.6 Protecting parameters

Although parameters are commonly both visible and modifiable this may be undesirable:

- Discoverable parameters may become an inadvertent API. Adding the parameters to a local broker prevents discovery.
- Model structure is generally fixed after the elaboration phase, so being able to modify structural parameters during the simulation can mislead. Restricting the parameter’s mutability to `CCI_IMMUTABLE_PARAM` will reject such misuse with a `CCI_SET_PARAM_FAILURE` error.
- Parameters may be modifiable during simulation but locked as read-only for clients, for example used to publish status. The publisher can `unlock()` the parameter prior to updating it, then `lock()` it again, or more concisely use a setter that accepts a password (but note the special-case behavior of `NULL` passwords below). Writes to a locked parameter are rejected with a `CCI_SET_PARAM_FAILURE` error.

NOTE: parameter locking is orthogonal to parameter mutability: a CCI_IMMUTABLE_PARAM can be locked and unlocked again but will remain always read-only.

```
cci_param_mutable_type get_mutable_type() const;
```

Returns the parameter's mutability type as described in clause 5.3.1, “cci_param_mutable_type”.

```
bool lock(const void *password = NULL);
```

`lock()` makes the parameter’s value password protected:

- if the parameter is unlocked then it becomes locked with a "password" address (ideally some pointer specific to the locking entity, such as its own `this)`:
    - the given password, if it is non-`NULL`
    - otherwise, with an implementation-defined private password unique to the parameter; a parameter locked in this way must be explicitly unlocked for its value to be set; setters that delegate to `set_cci_value()` and `set_raw_value()` with a `NULL` password will not override the lock (as would happen with an explicit non-`NULL` password)
- if the parameter is locked then:
    - if it already has the given password then it remains locked with it
    - if it has the default `NULL` password then this is upgraded to the given password
    - otherwise it remains locked by the previous password

`lock()` returns `true` if the parameter is now locked with the given password; returning `false` means the parameter is also locked but previously by some other password.

```
bool unlock(const void *password = NULL);
```

To unlock a locked parameter call `unlock()` with the same password used for the latest successful call to `lock()`. If locked without a password, it can also be unlocked (by anyone) without a password. It returns `true` if the parameter became unlocked from this call, `false` otherwise (i.e. either the parameter remains locked or it was already unlocked)

NOTE: locking does not nest; a parameter locked twice with the same password will be unlocked by a single `unlock()` with that password.

```
bool is_locked() const;
```

Returns `true` if the parameter is currently locked.

5.4.2.7 Equality test

```
bool equals(const cci_param_if &rhs) const;
```

Returns `true` if both the type and value of the parameter argument match this parameter as determined by `get_type_info()` and `get_raw_value()`. The value comparison is delegated to the parameter's underlying data type.

Example:

```
cci_param<short> iS("iS", 3);
cci_param<long> iL("iL", 3);
sc_assert( !iS.equals(iL) );                            // short and long are distinct types
sc_assert( iS.get_cci_value() == iL.get_cci_value() );   // but all integer types do fit "3"
```

5.4.2.8 Callbacks

Deferred callback functions can be registered for access to the parameter value. The complete callback interface is extensive since it is the product of functions supporting different phases of invocation, different parameter data types, both global and member functions, and is distributed across both typed and untyped parameter classes and both object and handle interfaces. Therefore the treatment here is not a monolithic exploration of the functions but decomposes it structurally. Although parameter types are a property of the derived typed classes they are discussed here so as to have a single coherent description of callbacks.

Callbacks can be registered against four stages of value access:

1. `register_pre_read_callback()`:
    - callback is invoked before the value is read
    - signature: `void callback(const cci::cci_param_read_event<T>& ev)`

2. `register_post_read_callback()`:
   - callback is invoked after the value is read (i.e. just before the value read is returned to the caller)
   - signature: `void callback(const cci::cci_param_read_event<T>& ev)`
3. `register_pre_write_callback()`:
   - callback is invoked before the new value is written
   - callback is explicitly a validator for the new value; by returning `false` it signals that the write should not proceed, in which case a `CCI_SET_PARAM_FAILURE` error report is immediately issued
   - signature: `bool callback(const cci::cci_param_write_event<T>& ev)`
4. `register_post_write_callback()`:
   - callback is invoked after the new value is written
   - signature: `void callback(const cci::cci_param_write_event<T>& ev)`

Callbacks are invoked in order of registration. If a callback throws an exception (including as part of error reporting) then this immediately propagates through the cci framework code without further callbacks being invoked and leaving all existing state modifications intact. For example a throw from a post-write callback will leave the parameter with the new value, which may surprise a user expecting assignment to have the commonly-supported copy-and-swap semantics. If callbacks are used to update complex state then consideration should be given to at least providing a basic exception guarantee (that system invariants are not violated).

The event object passed to the callback function carries the current parameter value, and also the new value for pre/post-write callbacks. Event objects passed to callbacks registered through the typed parameter interface `cci_param_typed<T>`/`cci_param_typed_handle<T>` convey the values as references to the actual type `T`. Event objects passed to callbacks registered through the untyped parameter interface `cci_untyped_param`/ `cci_param_untyped_handle` convey the values as references to `cci_value`.

For each access stage a pair of overloads exists for registering callbacks: one which creates a functor from the given global/class-static method and another which creates a functor for the given member function:

Example:

```
cci_callback_untyped_handle h1 =
    param.register_pre_read_callback(&global_callback);
cci_callback_untyped_handle h2 =
    param.register_pre_read_callback(&myclass::member_callback, &myclass_object);
```

Note that registration functions of this form are not present in the basic `cci_param_if`, but are introduced in `cci_param_untyped` and `cci_param_untyped_handle` for callbacks with untyped event objects (clause 5.6.2.6, “Callbacks”), and `cci_param_typed` and `cci_param_typed_handle` for callbacks with typed event objects (clause 5.6.4.4, “Callbacks”).

Although the handle object returned from callback registration encapsulates the function to be called and its arguments, from the users perspective it's an opaque token to be used if the callback is to be explicitly unregistered:

Example:

```
bool success = param.unregister_pre_read_callback(h1);
```

returning true if that callback handle was successfully removed from the callbacks for that phase. A specific callback can only be unregistered by providing the callback handle returned when it was registered and unregistering against the correct access stage (i.e. the handle returned from `register_pre_write_callback()` must be passed to `unregister_pre_write_callback()`).

Unregistration is only necessary if the callback is to be suppressed during the lifetime of the parameter, since it is not an error to destroy a parameter that has callbacks remaining registered. `true` is returned if the unregistration was successful. The callback handle is only useful for later unregistration; if the callback is to remain for the lifetime of the parameter then the handle need not be stored.

Lambda functions may also be conveniently used, either simply in place of an explicit function:

Example:

```
// Running count of times that parameter is set to zero
param.register_post_write_callback( [this](auto ev){ this->num_zeroes += ev.new_value == 0; });
```

or to adapt a generic member function with instance-specific parameters:

Example:

```
void audit::updated(const cci::cci_param_write_event<int>& ev, string category);

// Updates to wheels register as mileage, those to axles register as maintenance, C++11
wheel1.register_post_write( [this](auto ev){ this->updated(ev, "mileage");} );
wheel2.register_post_write( [this](auto ev){ this->updated(ev, "mileage");} );
shaft.register_post_write( [this](auto ev){ this->updated(ev, "maintenance");} );
```

Achieving similar results in a C++03 environment (given a C++03-supporting implementation of CCI):

Example:

```
// Running count of times that parameter is set to zero
void count_zero_writes(const cci::cci_param_write_event<int> & ev) {
  num_zeroes += ev.new_value == 0;
}

param.register_post_write_callback(audit::count_zero_writes, this);
```

and to adapt a function, `sc_bind()` can be used:

Example:

```
wheel1.register_post_write(sc_bind(&audit::updated, this, sc_unnamed::_1, "mileage"));
wheel2.register_post_write(sc_bind(&audit::updated, this, sc_unnamed::_1, "mileage"));
shaft.register_post_write(sc_bind(&audit::updated, this, sc_unnamed::_1, "maintenance"));
```

Basic registration interface

The interface provided through `cci_param_if` is intended for use by derived parameters and parameter handles; users will find it more convenient to use the registration overloads exposed by those classes. Only the pre-read phase is detailed here; the behavior of the other three phases is essentially the same:

```
typedef cci_param_pre_read_callback<>::type
        cci_param_pre_read_callback_untyped;
cci_callback_untyped_handle register_pre_read_callback(
    const cci_callback_untyped_handle& cb, const cci_originator& orig );
```

The callback handle is paired with the given originator and appended to the list of pre-read callbacks, and a copy of the callback handle is returned. The originator is presented to the callback through `cci_param_[read|write]_event::originator`.

Unregistering all callbacks:

In addition to unregistering a specific callback handle, all callbacks for all four phases registered by a specific originator can be removed:

```
bool unregister_all_callbacks (const cci_originator& orig);
```


returning `true` if any callback was unregistered. The originator might be retrieved from `get_originator()` on the parameter object or parameter handles; for handles a possible shortcut is `cci_broker_handle::get_originator()` since all parameter handles created from a broker handle share its originator.

Testing for callbacks

```
bool has_callbacks() const;
```

Returns `true` if any callbacks are registered against the parameter, regardless of the originator or phase.

5.4.2.9 Parameter handle management

```
cci_param_untyped_handle create_param_handle(const cci_originator &originator) const;
```

Creates and returns a handle, as described in clause 5.6.3, for the parameter. The handle's originator is set to the given originator. The returned handle is certain to be valid and remains so until such time as the parameter is destroyed.

```
private:
 void add_param_handle(cci_param_untyped_handle* param_handle) = 0;
 void remove_param_handle(cci_param_untyped_handle* param_handle) = 0;
```

The explicit decoupling of parameter object and handle lifetimes requires that a list of (parameter, handle) pairs is maintained, such that destroying a parameter can invalidate all handles to it. The CCI design places this responsibility upon the parameter at the API level (the implementation may delegate it beyond this), which requires these methods to add and remove handles. They are private and provided solely for the `cci_param_untyped_handle` implementation's use.

5.4.2.10 Destructor

```
~cci_param_if();
```

This empty destructor must be overridden by subclass to address:

- Discarding of all registered callbacks
- Invalidation of any `cci_param_[un]typed_handle` pointing to this parameter, after which their `is_valid()` method returns `false` and most operations on the handle will fail with an error report
- Unregistration of the parameter name, meaning that a parameter with the same hierarchical name can be created without having a unique suffix appended
- Removal from the broker, with the preset value (if any) being marked as unconsumed

5.4.3 cci_broker_if

The broker interface class provides metadata and variant value access. A default implementation is provided by `cci_utils::consuming_broker` described in clause 5.7.3. Brokers are typically access through a `cci_broker_handle` obtained from `cci_get_broker()`.

```
class cci_broker_if : public cci_broker_callback_if
{
public:
    // Broker properties
    virtual const std::string &name() const = 0;
    virtual bool is_global_broker() const = 0;

    // Parameter access
    virtual cci_param_untyped_handle get_param_handle(
        const std::string &parname, const cci_originator& originator) const = 0;
    virtual cci_originator get_latest_write_originator(
        const std::string &parname) const = 0;
    virtual cci_value get_cci_value(const std::string &parname) const = 0;

    // Bulk parameter access
    virtual std::vector <cci_param_untyped_handle> get_param_handles(
        const cci_originator& originator = cci_originator()) const = 0;
    virtual cci_param_range get_param_handles(
```



```
        cci_param_predicate& pred, const cci_originator& originator) const = 0;

    // Parameter initialization
    virtual bool has_preset_value(const std::string &parname) const = 0;
    virtual void set_preset_cci_value(
        const std::string &parname, const cci_value &cci_value,
        const cci_originator& originator) = 0;
    virtual cci_value get_preset_cci_value(const std::string &parname) const = 0;
    virtual void lock_preset_value(const std::string &parname) = 0;
    virtual std::vector<cci_name_value_pair> get_unconsumed_preset_values() const = 0;
    virtual cci_preset_value_range get_unconsumed_preset_values(
        const cci_preset_value_predicate &pred) const = 0;
    virtual void ignore_unconsumed_preset_values(
        const cci_preset_value_predicate &pred) = 0;

    // Handle creation
    virtual cci_broker_handle create_broker_handle(
            const cci_originator &originator = cci_originator()) = 0;

    // Callbacks
    virtual cci_param_create_callback_handle register_create_callback(
        const cci_param_create_callback &, const cci_originator &) = 0;
    virtual bool unregister_create_callback(
        const cci_param_create_callback_handle &, const cci_originator &) = 0;
    virtual cci_param_destroy_callback_handle register_destroy_callback(
        const cci_param_destroy_callback &, const cci_originator &) = 0;
    virtual bool unregister_destroy_callback(
        const cci_param_destroy_callback_handle &, const cci_originator &) = 0;
    virtual bool unregister_all_callbacks(const cci_originator &) = 0;
    virtual bool has_callbacks() const = 0;

    // Parameter un/registration
    virtual void add_param(cci_param_if *par) = 0;
    virtual void remove_param(cci_param_if *par) = 0;
protected:
    virtual ~cci_broker_if();
};
```

5.4.3.1 Broker properties

A broker is constructed with a name, which is made unique if necessary by `cci_gen_unique_name()`. Broker names distinguish between the global and local brokers, and can document them in logging.

```
const std::string &name();
```

Returns the broker's name.

```
bool is_global_broker() const;
```

Returns `true` for the global broker, `false` otherwise.

5.4.3.2 Individual parameter access

A broker provides handles to access the parameters it manages.

```
cci_param_untyped_handle get_param_handle(
    const std::string &parname, const cci_originator& originator) const = 0;
```

Given the full hierarchical name of a parameter registered on this broker and the originator to record as the source of writes through the handle, it returns a newly-created handle for the parameter. If the name doesn't match the current parameters then the handle is explicitly invalid.

Example:

```
cci_param<int> p("p1", 42);                       // CCI_RELATIVE_NAME
cci_param_handle ph = broker.get_param_handle("p1");  // get uses CCI_ABSOLUTE_NAME
```

```
sc_assert( !ph.is_valid() );
ph = broker.get_param_handle("testmod.p1");
sc_assert( ph.is_valid() );
```

For convenience and potential efficiency a small subset of the parameter functionality is made directly available:

```
cci_originator get_latest_write_originator(const std::string &parname) const = 0;
cci_value get_cci_value(const std::string &parname) const = 0;
```

`get_latest_write_originator()` returns a copy of the originator that most recently set the parameter's value, or if the parameter is not currently registered then an originator for which is_unknown() is `true`.

5.4.3.3 Bulk parameter access

Retrieves a vector of all parameter handles registered on the broker (and in the case of local brokers those registered on the parental hierarchy), optionally interposing a filtering predicate such that iterating through the vector skips past the handles that the predicate rejects:

```
std::vector <cci_param_untyped_handle> get_param_handles(
    const cci_originator& originator = cci_originator()) const;
cci_param_range get_param_handles(
    cci_param_predicate& pred, const cci_originator& originator) const;
```

Note that generating a handle for every parameter (and subsequently removing them when the vector is destroyed) may be expensive. Note also that the predicate form doesn't avoid this expense – in the following example handles for parameters "b" and "c" are still generated, merely hidden by the range iterator.

Example:

```
cci_param<int> pa("a", 1);
cci_param<string> pb("b", "foo");
cci_param<double> pc("c", 2.0);
cci_param<short> pd("d", 3);

// Simple predicate accepting only numeric params
cci_param_predicate pred([](const cci_param_handle&p)
    {
    return p.get_data_category() == CCI_NUMBER_PARAM;
    } );

cci_param_range r = broker.get_param_handles(pred);
for (auto p : r)
    cout << p.get_name() << endl;  // lists "a" and "d"
```

5.4.3.4 Parameter initialization

A newly-created parameter has the default value, with which it was constructed. This can be supplanted by a preset value, supplied by the broker to which the parameter is added.

```
virtual bool has_preset_value(const std::string &parname) const = 0;
```

Indicates whether the broker has a preset value for the specified parameter.

```
void set_preset_cci_value(
         const std::string &parname, const cci_value &cci_value,
         const cci_originator& originator);
```

Sets the preset value for the parameter with the given full hierarchical name. Whenever a parameter of that name is added to the broker its value will be set to the given preset value and the last_write_originator to the given originator. Note that the `cci_value` added must support `template<typename T> get()` for the `cci_param<T>` being added or a `CCI_VALUE_FAILURE` error will be reported. For example here the value of `qNum` will be displayed as "17.0" (small int successfully coerced as double) but the construction of `qStr` will report `CCI_VALUE_FAILURE` and depending upon `sc_report_handler` configuration either throw the error report or proceed without applying the configuration.

Example:

```
cci_broker_manager::get_broker(origI).set_preset_cci_value("m.q", cci_value(17));
{
    cci_param<double> qNum("q", 2.0, "desc", CCI_RELATIVE_NAME, origD);
    cout << "q val=" << qNum.get_cci_value() << endl;
}
{
    cci_param<string> qStr("q", "fish", "desc", CCI_RELATIVE_NAME, origD);
    cout << "q val=" << qStr.get_cci_value() << endl;
}
```

The parameter name is used after it has been made unique, meaning that if two parameters with the same hierarchical name are added only the first will receive the preset value as the second will have been suffixed with a sequence number. The preset value can be changed by further calls to `set_preset_cci_value()` but cannot be removed.

```
cci_value get_preset_cci_value(const std::string &parname) const;
```

Returns the current preset value for the parameter with the given full hierarchical name, or a null `cci_value` if no preset value is defined. Note that a null `cci_value` could in fact be the configured preset value for a parameter.

```
void lock_preset_value(const std::string &parname);
```

If the preset value for the parameter with the given full hierarchical name is locked then attempts to `set_preset_cci_value()` for it will be rejected with a `set_param_failed()` error. It can be locked before any `set_preset_cci_value()` call, meaning that no preset value can be defined and the default value will be in effect. A locked preset value cannot be unlocked.

Enumerating unconsumed preset values

A preset value that is configured but not "consumed" by being assigned to a created parameter may indicate a configuration error such as incorrect hierarchical names or an expected module not being instantiated. A tool or log file might provide such information to the user.

```
std::vector<cci_name_value_pair> get_unconsumed_preset_values() const;
```

Returns a list of all preset values not used for the current set of parameters, as pairs of `(parameter name, preset cci_value)`. A preset value is marked as used when a parameter of that name is constructed and is marked again as unused when that parameter is destroyed. The most useful time to report unconsumed preset values is typically after the end of elaboration.

The list of unconsumed preset values can be filtered by a predicate, for example to remove expected entries:

```
cci_preset_value_range get_unconsumed_preset_values(
    const cci_preset_value_predicate &pred) const;
```

Returns a range iterator for the list of unconsumed preset values, which filters the iteration functions by the given predicate callback. The predicate is presented with `std::pair<parameter_name, parameter preset cci_value>` and returns `false` to skip (suppress) the preset. In the following example, presets for a test module are ignored by checking for a hierarchy level named "testmod".

Example:

```
auto uncon = cci_broker_manager::get_broker().get_unconsumed_preset_values(
    [](const std::pair<string,cci_value>& iv)
     { return iv.first.find("testmod.") == string::npos; }
  );
for(auto v : uncon)
  {
  SC_REPORT_INFO("Unconsumed preset: ", v.first);
  }
```

The provision of the filtering predicate and the retrieval of the list of unconsumed presets can be performed as separate operations:

```
void ignore_unconsumed_preset_values(
    const cci_preset_value_predicate &pred);
```

Applies the given filtering predicate to the current set of unconsumed preset values and accumulates the matches from all such calls in a list of presets to be filtered (omitted) from the results of subsequent calls to `get_unconsumed_preset_values()`. Because the predicate is applied immediately it is advisable that the complete set of preset values is configured before modules and parameters are initialized, i.e. a suitable workflow is:

1. Create (possibly local) broker
2. Initialize presets through `cci_broker_[if|handle]::set_preset_cci_value()`
3. As part of defining parameters modules, use `cci_broker_handle::ignore_unconsumed_preset_values()` to add matching (currently unconsumed) presets to the suppression list.
4. Later (or at end) of simulation fetch the list of interesting preset values that remain unconsumed through `cci_broker_handle::get_unconsumed_preset_values()`

5.4.3.5 Create handle

```
cci_broker_handle create_broker_handle(const cci_originator &originator = cci_originator());
```

Return a newly-created and initialized handle for the broker. The given originator is used for operations that ultimately result in attributable changes, for example setting a preset value or creating a parameter handle.

5.4.3.6 Broker callbacks

Deferred callback functions can be registered on a broker for the creation and destruction of parameters (strictly, this is the addition and removal of the parameters from the broker, however this occurs solely in the context of creating and destroying parameters). The distinction is only important because it means that there is no mechanism for being notified of all parameter creations, so local brokers remain truly local.

Callbacks are invoked in order of registration. If a callback throws an exception (including as part of error reporting) then this immediately propagates through the cci framework code without further callbacks being invoked and leaving all existing state modifications intact. If callbacks are used to update complex state then consideration should be given to at least providing a basic exception guarantee (that system invariants are not violated)

As a structural detail, broker callbacks are actually declared through `cci_broker_callback_if`, the base class of `cci_broker_if`.

Creation callbacks

```
cci_param_create_callback_handle register_create_callback(
    const cci_param_create_callback &, const cci_originator &);
```

Registers a callback function of the signature: `void callback(const cci_param_untyped_handle& ph)`, paired with the given originator. The returned `cci_param_create_callback_handle` is used to unregister the callback.

Creation callbacks are invoked from within the `cci_param_typed` constructor as almost the final action. This means that the parameter handle is functional, but that any further-derived class has not been constructed (this will only be problematic if the `cci_param_typed` is sub-classed, then from the callback `dynamic_cast<sub-class>` will fail). If the callback throws an exception, either directly or through `sc_report_handler::report()`, then the parameter construction is unwound without running destruction callbacks.

```
bool unregister_create_callback(
    const cci_param_create_callback_handle &, const cci_originator& orig);
```

Given both the handle returned by registering a callback through `register_create_callback()` and the same originator with which the registration was made, it unregisters the callback and returns `true`.

Destruction callbacks

```
cci_param_destroy_callback_handle register_destroy_callback(
    const cci_param_destroy_callback &, const cci_originator& orig) = 0;
```

Registers a callback function of the signature: `void callback(const cci_param_untyped_handle& ph)`. The returned cci_param_destroy_callback_handle is used to unregister the callback.

Destruction callbacks are invoked with the parameter still fully constructed and registered with the broker.

Since destruction callbacks are invoked in the context of parameter destruction, exceptions should be avoided but are not prohibited. The behavior in such a case will be defined by the `cci` implementation and may result in `std::terminate().`

```
bool unregister_destroy_callback(
    const cci_param_destroy_callback_handle &, const cci_originator &) = 0;
```

Given the handle returned by registering a callback through `register_destroy_callback()` it unregisters the callback and returns `true`.

Utilities

```
bool unregister_all_callbacks(const cci_originator& orig) = 0;
```

Unregisters all creation and destruction callbacks registered with the given `cci_originator`. Returns `true` if any callbacks were unregistered.

```
bool has_callbacks() const = 0;
```

Returns `true` if any creation or destruction callbacks are currently registered with this broker.

5.4.3.7 Parameter registration

```
    virtual void add_param(cci_param_if *par) = 0;
    virtual void remove_param(cci_param_if *par) = 0;
```

These should only be called by parameter implementations and facilitate registering and unregistering with the broker.

5.4.3.8 Destructor

```
~cci_broker_if();
```

The destructor is protected to reserve destruction for the owner of the broker. Broker owners must be aware that destroying a broker which still has registered parameters is a grievous error (the current reference implementation does not explicitly error this case, however implementations should invoke `cci_abort()`). This is by design as there is no provision for gracefully handling dependent objects such as `cci_broker_handle` (unlike the relationship between `cci_param_if` and `cci_param_[un]typed_handle` where the lifetimes are explicitly decoupled).

In practice employing a common scoping mechanism for both local brokers and their parameters should avoid problems with mismatched lifetimes; for example making both the broker and the parameters member data of a module.

5.5 Variant type parameter values

It shall be possible to examine and modify configuration parameter values of unknown and arbitrarily complex types.

5.5.1 cci_value_category (enum)

The enumeration **cci_data_type** shall define the basic data types that can be used as building blocks to compose variant type parameter values.

```
enum cci_value_category {
  CCI_NULL_VALUE = 0,
```



```
  CCI_BOOL_VALUE,
  CCI_INTEGRAL_VALUE,
  CCI_REAL_VALUE,
  CCI_STRING_VALUE,
  CCI_LIST_VALUE,
  CCI_OTHER_VALUE
};
```

- CCI_NULL_VALUE – no data type, e.g. a variant object with no explicit initialization.
- CCI_BOOL_VALUE – C++ `bool` type
- CCI_INTEGRAL_VALUE – integer of up to 64 bits, i.e. representable as `int64_t` or `uint64_t`
- CCI_REAL_VALUE – floating point value, represented as C++ `double`
- CCI_STRING_VALUE – C++ null-terminated string
- CCI_LIST_VALUE – a list of values, each of which can be of any cci_value_category
- CCI_OTHER_VALUE – a type not matching any other category, including value-maps

5.5.2 cci_value

The `cci_value` class shall provide a variant type for exchanging configuration parameter values. The following types are supported:

- The familiar C++ data types referred to by cci_value_category are supported, as are restricted types that can be coerced into them, such as int32_t, int16_t, int8_t.
- Common SystemC data types: sc_core::sc_time, from sc_dt: sc_logic, sc_int_base, sc_uint_base, sc_signed, sc_unsigned, sc_bv_base, sc_lv_base.
- User-specific data types, supported by implementing the helper template class `cci_value_converter<T>` (which is also the mechanism by which the C++ and SystemC data types are supported).
- C++ arrays and `std::vector<>` of any supported data type, converting to a `cci_value_list`.
- Lists (vectors) of `cci_value`, represented as `cci_value_list`.
- String-keyed maps of `cci_value`, represented as `cci_value_map`.

Because lists and maps contain cci_value objects they are explicitly heterogeneous and can arbitrarily mix data types, including nesting `cci_value_list` and `cci_value_map` to arbitrary depths.

Objects of this class have strict value semantics, i.e. each value represents a distinct object. Due to hierarchical nature of the data structure, values embedded somewhere in the list or map are referenced by dedicated reference objects (cci_value_cref, cci_value_ref, and their specialized variants for strings, lists and maps), with or without constness.

The `cci_value::reference` and `cci_value::const_reference` classes are defined as modifier and accessor interface classes, such that a `cci_value` instance can be transparently used where those interface classes are expected. Having them form base classes for `cci_value` is a suggested approach.

5.5.2.1 Class definition

```
  class cci_value : public implementation-defined
  {
   public:
    /// reference to a constant value
    typedef implementation-defined const_reference;
    /// reference to a mutable value
    typedef implementation-defined reference;
    /// reference to a constant string value
    typedef implementation-defined const_string_reference;
    /// reference to a mutable string value
    typedef implementation-defined string_reference;
    /// reference to a constant list value
    typedef implementation-defined const_list_reference;
    /// reference to a mutable list value
    typedef implementation-defined list_reference;
    /// reference to a constant map value
    typedef implementation-defined const_map_reference;
```



```
    /// reference to a mutable map value
    typedef implementation-defined map_reference;

    // Constructors and destructor
    cci_value();
    template<typename T>
    explicit cci_value( T const & src, typename cci_value_converter<T>::type* = 0);

    cci_value( this_type const & that );
    cci_value( const_reference that );
    cci_value( this_type&& that );
    cci_value( cci_value_list&& that );
    cci_value( cci_value_map&& that );

    this_type& operator=( this_type const & );
    this_type& operator=( const_reference );

    this_type& operator=( this_type&& );
    this_type& operator=( cci_value_list&& );
    this_type& operator=( cci_value_map&& );

    friend void swap( this_type& a, this_type& b );
    void swap( reference that );
    void swap( cci_value & that );


    // Type queries - possibly inherited from "const_reference"
    cci_value_category category() const;
    bool is_null() const;
    bool is_bool() const;
    bool is_false() const;
    bool is_true() const;
    bool is_number() const;
    bool is_int() const;
    bool is_uint() const;
    bool is_int64() const;
    bool is_uint64() const;
    bool is_double() const;
    bool is_string() const;
    bool is_map() const;
    bool is_list() const;
    bool is_same(const_reference that) const;

    // Set basic value - possibly inherited from "reference"
    reference set_null();
    reference set_bool( bool v );
    reference set_int( int v );
    reference set_uint( unsigned v );
    reference set_int64( int64 v );
    reference set_uint64( uint64 v );
    reference set_double( double v );
    string_reference set_string( const char* s );
    string_reference set_string( const_string_reference s );
    string_reference set_string( const std::string& s );
    list_reference set_list();
    map_reference set_map();

    // Set arbitrarily typed value - possibly inherited from "reference"
    template< typename T >
      bool try_set( T const& dst, CCI_VALUE_ENABLE_IF_TRAITS_(T) );
    template< typename T >
      reference set( T const& v, CCI_VALUE_ENABLE_IF_TRAITS_(T) );

    // Get basic value - possibly inherited from "const_reference"
    bool get_bool() const;
    int get_int() const;
    unsigned get_uint() const;
    int64 get_int64() const;
    uint64 get_uint64() const;
    double get_double() const;
    double get_number() const;

    // Get arbitrarily typed value
    template<typename T>
      bool try_get( T& dst ) const;
```



```
    template<typename T>
      (T) get() const;

    // Access as complex value - possibly inherited
    const_string_reference get_string() const;
    string_reference get_string();
    const_list_reference get_list() const;
    list_reference get_list();
    const_map_reference get_map() const;
    map_reference get_map();

    // JSON (de)serialization - possibly inherited
    static cci_value from_json( std::string const & json );
    std::string to_json() const;

    // Friend functions
    friend std::istream& operator>>( std::istream& is, cci_value& v );
  };
```

5.5.2.2 Constructors and destructor

```
cci_value();
```

A default-constructed value has the `cci_value_category` of `CCI_NULL_VALUE`.

```
template<typename T>
  explicit cci_value( T const& src );
```

Construction from a source data type internalizes the value through `cci_value_converter<T>::pack()`. For the conventional data types these delegate to the appropriate explicit setter functions.

```
cci_value( cci_value const& that );
cci_value( const_reference that );
```

Copy-construction, overloaded both for a sibling instance and the `const_reference` accessor interface.

```
cci_value( cci_value&& that );
cci_value( cci_value_list&& that );
cci_value( cci_value_map&& that );
```

Move-construction, acquiring the value of `that` and leaving `that` freshly initialized. The list and map overloads correctly acquire the container types to ensure that the source is left initialized empty and of the correct type.

An implementation may provide similar semantics when compiled for C++ versions prior to C++11, for example through additional methods.

```
~cci_value();
```

Frees the associated value storage. Because reference objects obtained from a `cci_value` are constructed as copies and subsequent assignment to them updates their own storage rather than aliasing the source's storage, they do not pose a dangling-reference hazard. The following example shows that `m2` going out of scope does not invalidate the `map_reference p1` assigned from it, and that `p1` continues to refer to the `cci_value m1` that it was constructed from.

Example:

```
cci_value m1;
cci_value::map_reference p1 = m1.set_map();
p1.push_entry("1", "a");      // m1 == { ["1", "a"] }
{
    cci_value m2;
    p1 = m2.set_map();         // m1 == { }, m2 = { }
    p1.push_entry("2", "b");  // m1 == { ["2", "b"] }, m2 == { }
}
p1.push_entry("3", "c");      // m1 == { ["2", "b"], ["3", "c"] }
```


5.5.2.3 Swap functions

```
void swap( cci_value& that );
void swap( reference that );
cci_value move();
```

The `swap()` functions exchange the value and type of "this" object with that of the supplied **cci_value** argument in an exception- and error-report-safe manner. The `move()` function returns a cci_value which has taken ownership of "this" object's value, with "this" object being reinitialized without an explicit value, i.e. equivalent to the state created by `set_null()`.

NOTE: These functions are intended to support efficient operations with C++ standard container classes and algorithms.

5.5.2.4 Type queries

```
cci_value_category category() const;
```

Returns the basic data type.

```
bool is_null() const;
bool is_bool() const;
bool is_number() const;
bool is_int() const;
bool is_uint() const;
bool is_int64() const;
bool is_uint64() const;
bool is_double() const;
bool is_string() const;
bool is_map() const;
bool is_list() const;
```

Return `true` if the current value can be retrieved as the specified type, or can't be retrieved in the case of `is_null()`. This depends on the data type and in the case of integers also whether the current value can be contained by such an integer type.

Example:

```
cci_value v(7);
sc_assert( v.is_int() && v.is_uint() && v.is_int64() && v.is_uint64() );
v = cci_value(1UL << 34);
sc_assert( !v.is_int() && !v.is_uint() && v.is_int64() && v.is_uint64() );
v = cci_value(1UL << 63);
sc_assert( !v.is_int() && !v.is_uint() && !v.is_int64() && v.is_uint64() );
```

In contrast, coersion between string, integer, and double types is not supported, even where no loss of precision would occur.

Example:

```
cci_value v(1);
sc_assert( v.is_int() && !v.is_double() && !v.is_string() );
v = cci_value(1.0);
sc_assert( !v.is_int() && v.is_double() && !v.is_string() );
v = cci_value("1");
sc_assert( !v.is_int() && !v.is_double() && v.is_string() );
```

Convenience functions combining `is_bool()` and testing the result of `get_bool()`:

```
bool is_false() const;
bool is_true() const;
```


5.5.2.5 Get value

Core types

Explicitly named functions get the core types by value:

```
bool get_bool() const;
int get_int() const;
unsigned get_uint() const;
int64 get_int64() const;
uint64 get_uint64() const;
double get_double() const;
double get_number() const;  // synonym for get_double()
```

In general an error is reported unless the type would be identified by an `is_TYPE()` query, i.e. a safe idiom is:

```
if( cv.is_TYPE() )
    value = cv.get_TYPE();
```

however getting a small integer as a larger one is supported:

```
if( cv.is_int() )
    value = cv.get_int64();
```

The reference implementation supports getting an integer as a double, however this may result in loss of precision.

Example:

```
cv.set_uint64( (1UL << 63) | 0 );
sc_assert( cv.get_uint64() == uint64_t(cv.get_double()) );
cv.set_uint64( (1UL << 63) | 1 );
sc_assert( cv.get_uint64() != uint64_t(cv.get_double()) );
```

Extended and user-defined types

Other value types are retrieved with the type-templated `get()` function:

```
template<typename T>
   typename cci_value_converter<Type>::type get() const;
```

This uses the `cci_value_converter<T>` to extract the stored value and convert it to an object of type `T`, which is returned by value. If the value cannot be converted, for example because it is of a different type, then a `cci_value_failure()` error is reported. The validation and conversion of each type `T` is defined by the `cci_value_converter<T>` implementation. Converters are provided by the CCI library for the supported data types listed in clause 5.5.2. If `get()` is used with a user-defined type that lacks a `cci_value_converter<T>` definition then linker errors will occur.

```
template<typename T>
  bool try_get( T& dst ) const;  // omitting additional type argument for C++ selection logic
```

A conditional form of `get()`, which upon success updates the typed reference argument and returns `true`.

Example:

```
sc_core::sc_time end;
if( !endVal.try_get(end) )
    return ENotFinished;
// calculate total running time; if end was defined then start must be defined
// so can use unconditional get.
sc_core::sc_time start = startVal.get<sc_core::sc_time>();
```

Reference types

The getters for the structured data types (string, list, and map) return by reference:

```
const_string_reference get_string() const;
string_reference get_string();
const_list_reference get_list() const;
list_reference get_list();
const_map_reference get_map() const;
map_reference get_map();
```

As would be expected of reference types, they share the common value.

Example:

```
cci_value val;
val.set_list();
cci_value::list_reference lr1 = val.get_list();
lr1.push_back(1);
sc_assert(lr1.size() == 1);
cci_value::list_reference lr2 = val.get_list();
lr2.push_back(2);
sc_assert(lr1.size() == 2);
```

A natural consequence of this is that changing the underlying value type invalidates the references.

Example:

```
cci_value val;
val.set_list();
cci_value::list_reference lr1 = val.get_list();
val.set_null();
sc_assert(lr1.size() == 0);  // throws a RAPIDJSON_ASSERT exception
```

5.5.2.6 Set value

Value setters:

1. Set the value type, including releasing any existing storage
2. For simple types initialize to the passed value
3. Returning a suitable reference; for simple types a `cci_value_cref` for the value object, for structured types (string, list, map) the matching type reference class (`string_reference`, `list_reference`, `map_reference` respectively)

```
reference set_null();
reference set_bool( bool v );
reference set_int( int v );
reference set_uint( unsigned v );
reference set_int64( int64 v );
reference set_uint64( uint64 v );
reference set_double( double v );
string_reference set_string( const char* s );
string_reference set_string( const_string_reference s );
string_reference set_string( const std::string& s );
list_reference set_list();
map_reference set_map();

template< typename T >
    bool try_set( T const& dst ); // omitting additional type argument for C++ selection logic
template< typename T >
    reference set( T const& v );  // omitting additional type argument for C++ selection logic
```


5.5.2.7 Identity query

```
bool is_same(const_reference that) const;
```

Returns `true` if both this value and the given reference are for the same underlying value object, as opposed to merely having values that evaluate according to `operator==()`.

5.5.2.8 JSON (de) serialization

```
std::string to_json() const;
```

Returns a JSON description of the value. For custom types this will typically be a list or a map (as specified by the `cci_value_converter<T>` implementation).

```
static cci_value from_json( std::string const & json );
```

Given a JSON description of the value, returns a new `cci_value` initialized with the value. Reports a value error if the JSON is invalid.

5.5.3 cci_value_list

A `cci_value_list` is conceptually a vector of `cci_value` objects, where each element remains a variant type, i.e. the value types placed in the vector can be heterogeneous.

Example:

```
cci_value_list val;
val.push_back(7).push_back("fish");
```

The `cci_value_list` type offers `const` and modifiable reference classes (as base classes in the reference implementation) along with the instantiable class. The reference classes provide container interfaces modeled on the C++ standard library such as iterators, while the instantiable class provides the expected construction and assignment methods.

```
class cci_value_list : public implementation-defined
{
public:
  typedef implementation-defined const_reference;
  typedef implementation-defined reference;

  typedef cci_value_iterator<reference>         iterator;
  typedef cci_value_iterator<const_reference>   const_iterator;
  typedef std::reverse_iterator<iterator>       reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

  // "const_reference" members
  bool      empty() const;
  size_type size()  const;
  size_type capacity() const;

  const_reference operator[]( size_type index ) const;
  const_reference at( size_type index ) const;

  const_reference front() const;
  const_reference back() const;

  const_iterator cbegin() const;
  const_iterator cend() const;

  const_iterator begin() const;
  const_iterator end() const;

  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;

  const_reverse_iterator crbegin() const;
  const_reverse_iterator crend() const;
```

```
  proxy_ptr operator&() const { return proxy_ptr(*this); }

  // "reference" (modifable) members
  this_type operator=( this_type const& );
  this_type operator=( base_type const& );

  cci_value move();

  void swap( this_type& );
  friend void swap(this_type a, this_type b);

  cci_value_list_ref reserve( size_type );
  cci_value_list_ref clear();

  reference operator[]( size_type index );
  reference at( size_type index );

  reference front()
  reference back()
  iterator begin();
  iterator end();
  reverse_iterator rbegin()
  reverse_iterator rend()

  cci_value_list_ref push_back( const_reference v );
  cci_value_list_ref push_back( cci_value && v );
  template<typename T>
    cci_value_list_ref_type push_back( const T & v

  iterator insert( const_iterator pos, const_reference value );
  iterator insert( const_iterator pos, size_type count, const_reference value );
  template< class InputIt >
  iterator insert( const_iterator pos, InputIt first, InputIt last );

  iterator erase( const_iterator pos );
  iterator erase( const_iterator first, const_iterator last );

  void pop_back();

  proxy_ptr operator&() const { return proxy_ptr(*this); }

  // Concrete class
  cci_value_list();
  cci_value_list( this_type const & );
  cci_value_list( const_reference );
  cci_value_list( this_type&& );

  this_type& operator=( this_type const & );
  this_type& operator=( const_reference );
  this_type& operator=( this_type && );

  friend void swap(this_type& a, this_type& b) { a.swap(b); }
  void swap( reference that ) { reference::swap( that ); }
  void swap( this_type & );

  ~cci_value_list();

  const cci_value_list * operator&() const { return this; }
  cci_value_list * operator&() { return this; }
};
```

5.5.4 cci_value_map

A `cci_value_map` is conceptually a map of string keys to `cci_value` objects, where each element remains a variant type, i.e. the value types placed in the vector can be heterogeneous.

Example:

```
cci_value_map vmap;
vmap["foo"] = cci_value(7);
vmap["bar"] = cci_value(sc_core::sc_time_stamp());
```


The `cci_value_map` type offers `const` and modifiable reference classes (as base classes in the reference implementation) along with the instantiable class. The reference classes provide container interfaces modelled on the C++ standard library such as iterators, while the instantiable class provides the expected construction and assignment methods.

```
class cci_value_map_cref : public implementation-defined
{
public:
  typedef implementation-defined const_reference;
  typedef implementation-defined reference;
  typedef size_t size_type;
  typedef cci_value_iterator<cci_value_map_elem_ref>  iterator;
  typedef cci_value_iterator<cci_value_map_elem_cref> const_iterator;
  typedef std::reverse_iterator<iterator>             reverse_iterator;
  typedef std::reverse_iterator<const_iterator>       const_reverse_iterator;

  // "const_reference" members
  bool empty() const;
  size_type size() const;
  bool has_entry( const char * key ) const;
  bool has_entry( std::string const & key ) const;
  bool has_entry( cci_value_string_cref key ) const;

  const_reference at( const char* key ) const;
  const_reference at( std::string const& key ) const;

  const_iterator cbegin() const;
  const_iterator cend() const;

  const_iterator begin() const;
  const_iterator end() const;

  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;

  const_reverse_iterator crbegin() const;
  const_reverse_iterator crend() const;

  const_iterator find( const char* key ) const;
  const_iterator find( const std::string& key ) const;

  proxy_ptr operator&() const { return proxy_ptr(*this); }

  // "reference" (modifable) members
  this_type operator=( base_type const& );
  this_type operator=( this_type const& );

  cci_value move();

  /// exchange contents with another map
  void swap( this_type& );
  friend void swap(this_type a, this_type b);

  this_type clear();

  reference at( const char* key );
  reference at( std::string const& key );
  reference operator[]( const char* key );
  reference operator[]( std::string const& key );

  iterator begin();
  iterator end();

  reverse_iterator rbegin();
  reverse_iterator rend();

  iterator find( const char* key );
  iterator find( const std::string& key );

  this_type push_entry( const char* key, const_reference value );
  this_type push_entry( std::string const& key, const_reference value );

  this_type push_entry( const char* key, cci_value&& value );
  this_type push_entry( std::string const& key, cci_value&& value );
```

```
  /// add an arbitrary cci_value_converter enabled value
  template<typename T>
    this_type push_entry( const char* key, const T & value );
  template<typename T>

  size_type erase( const char* key );
  size_type erase( const std::string& key );

  iterator erase( const_iterator pos );
  iterator erase( const_iterator first, const_iterator last );

  proxy_ptr operator&() const { return proxy_ptr(*this); }

  // Concrete class
  cci_value_map();
  cci_value_map( this_type const & );
  cci_value_map( const_reference );
  cci_value_map( this_type && );

  this_type& operator=( this_type const& );
  this_type& operator=( const_reference );
  this_type& operator=( this_type && );

  friend void swap(this_type& a, this_type& b);
  void swap( reference that );
  void swap( this_type & );

  ~cci_value_map();

  const cci_value_map * operator&() const { return this; }
  /// @copydoc cci_value_cref::operator&
  cci_value_map * operator&() { return this; }
};
```

5.5.4.1 Element access

The `const_map_reference` interface provides the checked `at()` function:

```
const_reference at( const char* key ) const;
const_reference at( std::string const& key ) const;
```

This returns a reference to the cci_value object at the given index, or reports a value error if the index is invalid. The map_reference interface retains the validity checking but returns a modifiable element reference:

```
reference at( const char* key );
reference at( std::string const& key );
```

and adds array-styled access which inserts new index values:

```
reference operator[]( const char* key );
reference operator[]( std::string const& key );
```

Example:

```
cci_value_map vmap;
cci_value::map_reference mr(vmap);
mr["foo"] = cci_value(1);
mr.at("foo") = cci_value(2);
mr.at("bar") = cci_value(3);  // reports CCI_VALUE error
```

5.6 Parameters

Actual parameters are created as instances of `cci_param_typed`, which in concert with its base class `cci_param_untyped` implements the `cci_param_if` interface. As the names suggest the functionality is divided between that common to all parameter types and that which depends upon the concrete value type.

5.6.1 cci_param_untyped

Implements much of the parent `cci_param_if` interface class and extends it with convenient registration of untyped callbacks. The inherited methods are described in the `cci_param_if` interface class and not further detailed here.

```
class cci_param_untyped : public cci_param_if
{
public:
    // The pre-read callback phase detailed here; equivalent methods exist for all three phases
    cci_callback_untyped_handle register_pre_read_callback(
            const cci_param_pre_read_callback_untyped& cb,
            cci_untyped_tag = cci_untyped_tag());
    template<typename C>
    cci_callback_untyped_handle register_pre_read_callback(
            cci_param_pre_read_callback_untyped::signature (C::*cb), C* obj,
            cci_untyped_tag = cci_untyped_tag());
    bool unregister_pre_read_callback(const cci_callback_untyped_handle &cb);
    bool unregister_all_callbacks();
};
```

These additional callback registration and unregistration methods provide a convenient veneer; the actual callback semantics remain as described in `cci_param_if`.

```
cci_callback_untyped_handle register_pre_read_callback(
    const cci_param_pre_read_callback_untyped& cb,
    cci_untyped_tag = cci_untyped_tag() );
```

Register a global function as a pre-read callback, using the parameter's originator as the callback originator (as passed to the callback through the `cci_param_read_event` object). The following example uses a static member function.

Example:

```
auto cbh = paramUT.register_pre_read_callback(&Logger::static_pre_read_callback);
```

Note that as above the packaging `cci_param_pre_read_callback_untyped` object will typically be implicitly constructed simply by passing the pointer to the static/global function .

```
template<typename C>
cci_callback_untyped_handle register_pre_read_callback(
        cci_param_pre_read_callback_untyped::signature (C::*cb), C* obj,
        cci_untyped_tag = cci_untyped_tag());
```

Register a member function as a pre-read callback, using the parameter's originator as the callback originator (as passed to the callback through the `cci_param_read_event` object).

Example:

```
auto cbh = paramUT.register_pre_read_callback(&Logger::member_pre_read_callback, &loggerObject);
```

Once again the packaging `cci_param_pre_read_callback_untyped` object will typically be implicitly constructed simply by passing the pointer to the member function along with a pointer to the instance.

```
bool unregister_pre_read_callback(const cci_callback_untyped_handle &cb);
```

Unregister a pre-read callback, given its registration handle. Returns `true` if successful. A `false` return may diagnose that unregistration was already performed or that the registration was made from a `cci_param_untyped_handle` (although all callback handles have the static type of `cci_callback_untyped_handle` it is required that unregistration is made through the same object as the registration).

```
bool unregister_all_callbacks();
```

Unregisters all callbacks for all four phases (i.e. pre-read, post-read, pre-write, and post-write) that were registered directly through this parameter object. Returns `true` if any callback was unregistered.

5.6.2 cci_param_typed

The concrete instantiable type for all parameters, extending `cci_param_untyped` with direct access to the parameter value. An instance is templated by:

- the data type. The data type must have the following set of features (note that this set is more extensive than is required for compatibility with `cci_value`, i.e. it is possible to construct a `cci_value` object with a value type that would not permit construction of a `cci_param_typed` object). Given the value type "`VT`":
    - default constructor: `VT()` (`DefaultConstructible` in C++ concept terminology)
    - copy constructor: `VT(const VT&)` (`CopyConstructible`)
    - value type assignment operator: `operator=(const VT&)` (`CopyAssignable`)
    - value type equality operator: `operator==(const VT&)` (`EqualityComparable`)
    - `cci::cci_value_converter<value type>` defined
- value mutability expressed as `cci_param_mutable_type`.

A concise alias of `cci_param` is provided for the common case of mutable parameters, as seen in these two equivalent definitions:

```
cci_param_typed<int, CCI_MUTABLE_PARAM> p1("p1", 0);
cci_param<int> p2("p2", 0);
```

The inherited methods are described in the `cci_param_if` interface class and not further detailed here.

```
template<typename T, cci_param_mutable_type TM = CCI_MUTABLE_PARAM>
class cci_param_typed : public cci_param_untyped
{
public:
    typedef T value_type;

    // Construction
    cci_param_typed(const std::string& name, const value_type& default_value,
                    const std::string& desc = "",
                    cci_name_type name_type = CCI_RELATIVE_NAME,
                    const cci_originator& originator = cci_originator());
    cci_param_typed(const std::string& name, const cci_value& default_value,
                    const std::string& desc = "",
                    cci_name_type name_type = CCI_RELATIVE_NAME,
                    const cci_originator& originator = cci_originator());
    cci_param_typed(const std::string& name, const value_type& default_value,
                    cci_broker_handle private_broker,
                    const std::string& desc = "",
                    cci_name_type name_type = CCI_RELATIVE_NAME,
                    const cci_originator& originator = cci_originator());
    cci_param_typed(const std::string& name, const cci_value& default_value,
                    cci_broker_handle private_broker,
                    const std::string& desc = "",
                    cci_name_type name_type = CCI_RELATIVE_NAME,
                    const cci_originator& originator = cci_originator());

    // Typed value access
    const value_type& get_value() const;
    const value_type& get_value(const cci_originator& originator) const;
    operator const value_type& () const;
    const value_type & get_default_value();

    void set_value(const value_type& value);
    void set_value(const value_type& value, const void * pwd);
    cci_param_typed& operator= (const cci_param_typed & rhs);
    cci_param_typed& operator= (const value_type & rhs);
    void reset();

    // For brevity, only the pre-read callbacks are detailed here
    cci_callback_untyped_handle register_pre_read_callback(
            const cci_param_pre_read_callback_untyped &cb,
            cci_untyped_tag);
```



```
    template<typename C>
    cci_callback_untyped_handle register_pre_read_callback(
            cci_param_pre_read_callback_untyped::signature (C::*cb), C* obj,
            cci_untyped_tag);

    typedef typename cci_param_pre_read_callback<value_type>::type
            cci_param_pre_read_callback_typed;

    cci_callback_untyped_handle register_pre_read_callback(
            const cci_param_pre_read_callback_typed &cb,
            cci_typed_tag<value_type> = cci_typed_tag<value_type>());

    template<typename C>
    cci_callback_untyped_handle register_pre_read_callback(
            typename cci_param_pre_read_callback_typed::signature (C::*cb),
            C* obj, cci_typed_tag<value_type> = cci_typed_tag<value_type>());

    cci_param_untyped_handle create_param_handle(
            const cci_originator& originator) const;


private:
    const void* get_raw_value(const cci_originator &originator) const;
    const void* get_raw_default_value() const;
    void set_raw_value(const void* vp, const void* pwd,
                       const cci_originator& originator);
private:
    void preset_cci_value(const cci_value& value, const cci_originator& originator) override;
};
```

5.6.2.1 value_type

The underlying data type that the `cci_param_typed` instance was instantiated with is aliased as `value_type`.

5.6.2.2 Construction

Four constructors are provided, combining the pairs of (automatic broker, explicit broker) and the default value expressed as (literal value_type, `cci_value`). The constructor parameters are:

- parameter **name** – parameters are indexed by name, which is required to be unique (duplicates are suffixed with a number to ensure this and a warning report issued)
- **default_value** – must be explicitly given rather than taken from value_type's implicit construction, either as the literal value_type or a `cci_value`
- **description** –a description of the parameter is encouraged, for example to annotate configuration logs; defaults to an empty string
- **name_type** – defaults to `CCI_RELATIVE_NAME`, in which case the parameter name is made absolute (or hierarchical) by prepending with the name of the enclosing `sc_module`
- **originator** – the origin of the default value and of subsequent assignments (unless those are made with an explicit originator); by default, an originator representing the current `sc_module`
- **explicit broker** – a specific broker to hold the parameter; if unspecified, the result of cci_get_broker() is used.

```
// Default as literal value_type, current broker
cci_param_typed(const std::string& name, const value_type& default_value,
                const std::string& desc = "",
                cci_name_type name_type = CCI_RELATIVE_NAME,
                const cci_originator& originator = cci_originator());

// Default as cci_value, current broker
cci_param_typed(const std::string& name, const cci_value& default_value,
                const std::string& desc = "",
                cci_name_type name_type = CCI_RELATIVE_NAME,
                const cci_originator& originator = cci_originator());

// Default as literal value_type, explicit broker
```



```
cci_param_typed(const std::string& name, const value_type& default_value,
                cci_broker_handle private_broker,
                const std::string& desc = "",
                cci_name_type name_type = CCI_RELATIVE_NAME,
                const cci_originator& originator = cci_originator());

// Default as cci_value, explicit broker
cci_param_typed(const std::string& name, const cci_value& default_value,
                cci_broker_handle private_broker,
                const std::string& desc = "",
                cci_name_type name_type = CCI_RELATIVE_NAME,
                const cci_originator& originator = cci_originator());
```

Parameters shall not be instantiated as C++ global variables. Global parameters are prohibited in order to guarantee that the global broker can be instantiated prior to the instantiation of any parameters.

5.6.2.3 Typed value access

The parameter value can be read and written directly as the `value_type`.

```
const value_type& get_value() const;
operator const value_type& () const; // convenience form of get_value()
```

Provides a typed reference to the current value. Note that the pre-read and post-read callbacks are triggered by the creation of the reference and not by actually reading the value, in contrast to `get_cci_value()` which takes a copy of the value. To avoid confusion, especially with callbacks, it is preferable to dereference the reference immediately rather than storing it for later use.

Example:

```
cci_param<int> p("p", 3);
p.register_post_read_callback(&log_reads);
const int& rp = p;  // log shows value 3 was read
p = 4;
int val_p = rp;     // current value of 4 is really "read"
```

```
const value_type & get_default_value();
```

Provide a typed reference to the default value.

```
void set_value(const value_type& value);
void set_value(const value_type& value, const void* pwd);
```

Pre-write callbacks are run, then the parameter value is copied from the argument, then post-write callbacks are run. If a lock password (`pwd`) is given then the parameter value must both be locked and the lock be with that password or a `CCI_SET_PARAM_FAILURE` error report will be issued.

```
void reset();
```

Convenient shortcut for `cci_param_if::reset(const cci_origiantor&)` using the parameter's owner as the originator, i.e. passing the result of `get_originator()`.

5.6.2.4 Raw value access

Direct untyped access to the parameter value storage is provided for the `cci_typed_handle` implementation; consequently these methods shall be private and accessed through `friend`-ship with the handle classes.

```
const void* get_raw_value(const cci_originator &originator) const override;
```

As with `cci_value` and `value_type` value queries, pre-read and post-read callbacks are executed before the pointer is returned.


```
const void* get_raw_default_value() const override;
```

Direct untyped access to the default value.

```
void set_raw_value(const void* vp, const void* pwd, const cci_originator& originator) override;
```

Pre-write callbacks are run, then the parameter value is copied from the `vp` argument, then post-write callbacks are run. The latest write originator is updated from the given originator. If the parameter is locked then the correct password must be supplied; if the parameter is not locked then the password must be set to `NULL`, or a `CCI_SET_PARAM_FAILURE` error report will be issued.

5.6.2.5 Assignment operator

```
cci_param_typed& operator= (const value_type & rhs);
```

An instance of the `value_type` can be assigned, as a shorthand for calling `set_value(const value_type&)`.

```
cci_param_typed& operator= (const cci_param_typed & rhs);
```

This parameter value is set to a copy of the given parameter's value. Incompatible `value_type`s may cause a compilation error or be reported as a `CCI_VALUE_FAILURE`.

5.6.2.6 Callbacks

The callback support of `cci_param_untyped` is extended with typed callbacks, which provide direct `value_type` access to the current and new parameter values. The semantics are further described in the `cci_param_if` clause 5.4.2.8.

Untyped callbacks can be registered through the `cci_param_typed` interface by explicitly tagging them as untyped:

```
void untyped_pre_read_callback(const cci::cci_param_read_event<void> & ev) {
  const cci_value& val = ev.value;
}
...
cci_param_typed<int> p("p", 1);
p.register_pre_read_callback(&untyped_pre_read_callback, cci_untyped_tag());
```

Typed callbacks are implicitly tagged:

```
void typed_pre_read_callback (const cci::cci_param_read_event<int> & ev) {
  const int& val = ev.value;
}
...
cci_param_typed<int> p("p", 1);
p.register_pre_read_callback(&typed_pre_read_callback);
```

The sixteen callback registration functions are then composed simply from: four access phases (pre-read, post-read, pre-write, and post-write), two function types (global, member), and two kinds of value access (untyped via `cci_value`, typed as `value_type`).

5.6.3 cci_param_untyped_handle

Parameter handles function as proxies for the parameter instances, providing most of the `cci_param_untyped` functionality (functionality such as setting descriptions and metadata is not present, as that is reserved for the parameter owner). They provide a means of reducing coupling in the model to the parameter name (and potentially value type).

The underlying parameter instance can be destroyed while handles remain, however this immediately invalidates the handles with effect:

- `is_valid()` returns `false`
- Calling any delegating method results in an error report

Once a handle has become invalid it remains forever invalid, even if a parameter of that name is recreated; conceptually the handle was created from a specific parameter instance, not for a parameter name (which may be valid at some times and not at other times).

Example:

```
auto p = new cci_param<int>("p", 5);
auto h1 = cci_broker_manager::get_broker().get_param_handle("testmod.p");
sc_assert( h1.is_valid() );
delete p;
sc_assert( !h1.is_valid() );
p = new cci_param<int>("p", );
auto h2 = cci_broker_manager::get_broker().get_param_handle("testmod.p");
sc_assert( h2.is_valid() );   // newly obtained handle functional
sc_assert( !h1.is_valid() );  // original handle for same name still invalid
```

5.6.3.1 **Class overview**

Handles are created with a specific originator, which is used in cases where the cci_param_untyped interface allows the originator to be specified. For example with the parameter handle created, subsequent setting of the value records that originator as the origin of writes:

```
auto ph = param.create_param_handle(orig);
ph.set_cci_value(val1);
ph.set_cci_value(val2);
```

where through the parameter interface the originator would be specified upon each setting:

```
param.set_cci_value(val1, orig);
param.set_cci_value(val2, orig);
```

Handles have no inherent collation properties and no comparisons are defined.

```
class cci_param_untyped_handle
{
public:
    // Constructors
    cci_param_untyped_handle(cci_param_if& param, const cci_originator& originator);
    explicit cci_param_untyped_handle(const cci_originator& originator);
    cci_param_untyped_handle(const cci_param_untyped_handle& param_handle);
    cci_param_untyped_handle(cci_param_untyped_handle&& that);

    ~cci_param_untyped_handle();

    // Assignment
    cci_param_untyped_handle& operator=(const cci_param_untyped_handle& param_handle);
    cci_param_untyped_handle& operator=(cci_param_untyped_handle&& that);

    // Handle validity
    bool is_valid() const;
    void invalidate();

    cci_originator get_originator() const;

    // Delegated functions
    cci_param_data_category get_data_category() const;
    const std::string& get_name() const;
    cci_param_mutable_type get_mutable_type() const;

    std::string get_description() const;
    cci_value_map get_metadata() const;

    cci_value get_cci_value() const;
    void set_cci_value(const cci_value& val);
    void set_cci_value(const cci_value& val, void* pwd);
```

```
    cci_value get_default_cci_value() const;
    void reset();

    bool lock(const void* pwd = NULL);
    bool unlock(const void* pwd = NULL);
    bool is_locked() const;

    bool is_default_value() const;
    bool is_preset_value() const;

    cci_originator get_latest_write_originator() const;

    // For brevity only pre-read callbacks are shown
    cci_callback_untyped_handle register_pre_read_callback(
        const cci_param_pre_read_callback_untyped &, cci_untyped_tag);
    cci_callback_untyped_handle register_pre_read_callback(
        const cci_callback_untyped_handle &, cci_typed_tag<void>);
    bool unregister_pre_read_callback(const cci_callback_untyped_handle &);

    bool unregister_all_callbacks();
    bool has_callbacks() const;


protected:
    // Raw value access provided for derived typed value accessors; no direct client access
    const void* get_raw_value() const;
    const void* get_raw_default_value() const;
    void set_raw_value(const void* vp);
    void set_raw_value(const void* vp, const void* pwd);
};
```

5.6.3.2 Construction

```
explicit cci_param_untyped_handle(const cci_originator& originator);
```

Create an explicitly uninitialized handle, i.e. where `is_valid() == false`.

```
cci_param_untyped_handle(cci_param_if& param, const cci_originator& originator);
```

Create a handle for the given parameter.

```
cci_param_untyped_handle(const cci_param_untyped_handle& param_handle);
```

Copy constructor; duplicates the given source handle, after which both the original and new handles have the same validity and originators but different identities (i.e. if valid then both are registered with the parameter and would be separately invalidated if the parameter predeceases them).

```
cci_param_untyped_handle(cci_param_untyped_handle&& that);
```

Move constructor; duplicate the original handle, after which the original handle is invalidated.

5.6.3.3 Destruction

```
~cci_param_untyped_handle();
```

Invalidates the handle (if valid), thereby unregistering it from the parameter as detailed for `~cci_param_if()`.

5.6.3.4 Assignment

```
cci_param_untyped_handle& operator=(const cci_param_untyped_handle& param_handle);
cci_param_untyped_handle& operator=(cci_param_untyped_handle&& that);
```

Assignment to a handle consists of:

- if valid, the existing destination handle is invalidated
- duplicating the source handle, after which both the original and new handles have the same validity and originators but different identities (i.e. if valid then both are registered with the parameter and would be separately invalidated if the parameter predeceases them)

- in the case of move-assignment, invalidating the source handle and resetting its originator to be `is_unknown()`

5.6.3.5 Handle validity

A handle constructed against a parameter begins its life as a valid handle for that parameter and remains valid until one of:

- destruction of the parameter
- explicit invalidation of the handle by `invalidate()`
- move construction or assignment from the handle

Once invalidated a handle remains invalid unless used as the destination for assignment from a valid handle.

```
bool is_valid() const;
```

Returns `true` if the handle is valid.

```
void invalidate();
```

Invalidates the handle: `is_valid()` returns `false` and the object is no longer registered with the parameter.

5.6.3.6 Delegated functions

With the conspicuous exception of `get_originator()`, the remainder of the class delegates predictably to the equivalent `cci_param_untyped` functionality with this pattern:

- if the handle is invalid then:
  - report a bad handle error through `cci_report_handler`
  - if the error report is not thrown as an exception (the SystemC default behavior but controllable through `sc_report_handler::set_actions()`) then calls `cci_abort()` to halt the simulation
- calls the matching `cci_param_untyped` member function of the parameter instance the handle represents, using the handle's originator where an explicit originator is catered for: `get_cci_value()`, `set_cci_value()`, `reset()`, callback registration and unregistration

The exception to this pattern is `get_originator()`, which returns the originator for the handle rather than that of the parameter.

Example:

```
sc_assert( !(origD == origI) );
cci_param<int> qp("q", 1, "q description", CCI_RELATIVE_NAME, origD);
cci_param_untyped_handle qh = qp.create_param_handle(origI);
sc_assert( qp.get_originator() == origD );
sc_assert( qh.get_originator() == origI #);
```

5.6.4 cci_param_typed_handle

Typed handles extend `cci_param_untyped_handle` with type-safe assignment and callbacks.

```
template<typename T>
class cci_param_typed_handle : public cci_param_untyped_handle
{
public:
    /// The parameter's value type.
    typedef T value_type;

    // Constructors
    explicit cci_param_typed_handle(cci_param_untyped_handle untyped);
    cci_param_typed_handle(const cci_param_typed_handle&) = default;
    cci_param_typed_handle(cci_param_typed_handle&& that);
```



```
    // Assignment
    cci_param_typed_handle& operator=(const cci_param_typed_handle&) = default;
    cci_param_typed_handle& operator=(cci_param_typed_handle&& that)

    // Typed value access
    const value_type& operator*() const;
    const value_type& get_value() const;

    void set_value(const value_type& value);
    void set_value(const value_type & value, const void * pwd);

    const value_type & get_default_value() const;

    // For brevity only pre-read callbacks are shown
    cci_callback_untyped_handle register_pre_read_callback(
            const cci_param_pre_read_callback_untyped &cb,
            cci_untyped_tag);
    template<typename C>
    cci_callback_untyped_handle register_pre_read_callback(
            cci_param_pre_read_callback_untyped::signature (C::*cb), C* obj,
            cci_untyped_tag);

    typedef typename cci_param_pre_read_callback<value_type>::type
            cci_param_pre_read_callback_typed;

    cci_callback_untyped_handle register_pre_read_callback(
            const cci_param_pre_read_callback_typed& cb,
            cci_typed_tag<value_type> = cci_typed_tag<value_type>());

    template<typename C>
    cci_callback_untyped_handle register_pre_read_callback(
            typename cci_param_pre_read_callback_typed::signature (C::*cb),
            C* obj, cci_typed_tag<value_type> = cci_typed_tag<value_type>())
};
```

5.6.4.1 Construction

```
explicit cci_param_typed_handle(cci_param_untyped_handle untyped);
```

Constructs the typed handle from an untyped handle, immediately invalidating it if the `typeid` of the `value_type` of the typed handle doesn't match the `typeid` of the `value_type` of the actual `cci_param_typed`.

Example:

```
cci_param_typed_handle<int> hTest( cci_get_broker().get_param_handle("global.test") );
if( !hTest.is_valid()) { /* param missing or wrong type */ }
```

```
cci_param_typed_handle(const cci_param_typed_handle&);
```

Copy constructor; duplicates the given source handle, after which both the original and new handles have the same validity and originators but different identities (i.e. if valid then both are registered with the parameter and would be separately invalidated if the parameter predeceases them).

```
cci_param_typed_handle(cci_param_typed_handle&& that);
```

Move constructor; duplicate the original handle, after which the original handle is invalidated.

5.6.4.2 Assignment

```
cci_param_typed_handle& operator=(const cci_param_typed_handle&);
cci_param_typed_handle& operator=(cci_param_typed_handle&& that);
```

Both copy and move assignment replace the handle, with the same semantics as `cci_param_untyped_handle`.

5.6.4.3 Typed value access

The parameter value can be read and written directly as the `value_type`. The semantics described for cci_param_typed typed value access in clause 0 apply here too.

```
const value_type& get_value() const;
const value_type& operator*() const;  // convenience form of get_value()

void set_value(const value_type& value);
void set_value(const value_type& value, const void * pwd);

const value_type & get_default_value() const;
```

5.6.4.4 Callbacks

Registration functions for callbacks providing `value_type` access to the parameter.

`cci_param_read_event` objects provide the context for pre-read and post-read callback invocations, carrying a handle to the parameter, its current value, and the originator that the callback function was registered with. The class is templated by the parameter value type, with the specialization for void providing the value as cci_value:

```
template<>
 struct cci_param_read_event<void>
 {
     typedef cci_param_read_event type;
     typedef cci_value value_type;

     const value_type& value;
     const cci_originator& originator;
     const cci_param_untyped_handle& param_handle;
 };

template<typename T>
 struct cci_param_read_event
 {
     typedef cci_param_read_event type;
     typedef T value_type;

     const value_type& value;
     const cci_originator& originator;
     const cci_param_untyped_handle& param_handle;
 };
```

The presence of the parameter's value type in the callback signature mirrors the parameter hierarchy, with callbacks registered through the `cci_param_untyped` class requiring the untyped `cci_param_read_event<void>` and those registered through `cci_param_typed<T>` requiring `cci_param_read_event<T>`. When working with a concrete parameter object it may prove advantageous to use untyped callbacks where the actual value is irrelevant or can be masked through `cci_value` access. For example a generic parameter access logger may have the signature:

```
void log_parameter_read(cci_param_read_event<void>& ev);
```

and so be able to be registered against `cci_param<int>`, `cci_param<std::string>`, etc.

5.6.5 cci_param_write_event

Write event objects provide the context for pre-write and post-write callback invocations, carrying a handle to the parameter, its current value, and the originator that the callback function was registered with.

The class is templated by the parameter value type, with the specialization for void providing the value as cci_value:

```
template<>
 struct cci_param_write_event<void>
 {
     typedef cci_param_read_event type;
     typedef cci_value value_type;
```

```
     const value_type& old_value;
     const value_type& new_value;
     const cci_originator& originator;
     const cci_param_untyped_handle& param_handle;
 };

template<typename T>
 struct cci_param_write_event
 {
     typedef cci_param_read_event type;
     typedef T value_type;

     const value_type& old_value;
     const value_type& new_value;
     const cci_originator& originator;
     const cci_param_untyped_handle& param_handle;
 };
```

The presence of the parameter's value type in the callback signature mirrors the parameter hierarchy, with callbacks registered through the `cci_param_untyped` class requiring the untyped `cci_param_write_event<void>` and those registered through `cci_param_typed<T>` requiring `cci_param_write_event<T>`. When working with a concrete parameter object it may prove advantageous to use untyped callbacks where the actual value is irrelevant or can be masked through `cci_value` access. For example a generic pre-write validator for positive numbers might be written:

```
bool validate_positive_number (cci_param_read_event<void>& ev)
{
    return ev.new_value.is_double() && ev.new_value.get_double() >= 0 ||
           ev.new_value.is_int64() && ev.new_value.get_int64() >= 0 ||
           ev.new_value.is_uint64();
}
```

and so be able to be registered as a pre_write callback against `cci_param<int>`, `cci_param<short>`, etc.

5.7 Brokers

- All brokers implement the basic `cci_broker_if` interface, although almost all users access them via `cci_broker_handle`.
- A broker aggregates parameters defined in the same `sc_object` level and from child objects. For example if a module registers a broker then the module's parameters and those belonging to submodules will by default be added to that broker. Such brokers are referred to as "local brokers" since the parameters they hold are kept local to that module, rather than being generally enumerable.
- Above the `sc_module` hierarchy is the global broker, which aggregates all parameters for which no local broker is located. The global broker must be registered before any parameters or local brokers.
- The closest broker is located by walking up the `sc_object` hierarchy until meeting either a local broker registration for that object or the global broker. Only one broker can be registered for each object; similarly a single global broker exists. Attempting to register additional brokers reports an error.
- Two reference broker implementations are provided: `cci_utils::broker` which supports selectively delegating parameters to a parent broker and `cci_utils::consuming_broker` which lacks this delegation ability. A module can use such delegation to expose some public parameters beyond its local broker.

5.7.1 cci_broker_handle

A broker handle acts as a proxy to a broker implementation, delegating the functionality. Note that where the delegated broker function takes an originator parameter, it is omitted in the handle interface since the handle was constructed with the originator.

Unlike the relationship between parameters and parameter handles, the relationship between broker objects and `cci_broker_handle`s is not managed. When a broker object is destroyed all handles to it are left dangling, without any way for the handle users to test their validity.

```
class cci_broker_handle
{
public:
    // Constructors
    cci_broker_handle(cci_broker_if& broker, const cci_originator& originator);
    cci_broker_handle(const cci_broker_handle&) = default;
    cci_broker_handle(cci_broker_handle&& that);

    ~cci_broker_handle() = default;

    // Assignment & comparison
    cci_broker_handle& operator=(const cci_broker_handle&) = default;
    cci_broker_handle& operator=(cci_broker_handle&& that);
    bool operator==(const cci_broker_if *b) const;
    bool operator!=(const cci_broker_if *b) const;

    // Originator
    cci_originator get_originator() const;

    // Delegated functions
    cci_broker_handle create_broker_handle(const cci_originator &originator = cci_originator());

    const std::string& name() const;

    void set_preset_cci_value(const std::string &parname, const cci_value &cci_value);
    cci_value get_preset_cci_value(const std::string &parname) const;

    std::vector<cci_name_value_pair> get_unconsumed_preset_values() const;
    bool has_preset_value(const std::string &parname) const;
    cci_preset_value_range get_unconsumed_preset_values(
        const cci_preset_value_predicate &pred) const;
    void ignore_unconsumed_preset_values(const cci_preset_value_predicate &pred);

    cci_originator get_latest_write_originator(const std::string &parname) const;

    void lock_preset_value(const std::string &parname);
    cci_value get_cci_value(const std::string &parname) const;

    void add_param(cci_param_if *par);
    void remove_param(cci_param_if *par);

    std::vector <cci_param_untyped_handle> get_param_handles() const;
    cci_param_range get_param_handles(cci_param_predicate& pred) const;
    cci_param_untyped_handle get_param_handle(const std::string &parname) const;

    template<class T>
    cci_param_typed_handle<T> get_param_handle(const std::string &parname);

    cci_param_create_callback_handle register_create_callback(
        const cci_param_create_callback& cb);
    bool unregister_create_callback(const cci_param_create_callback_handle& cb);
    cci_param_destroy_callback_handle register_destroy_callback(
        const cci_param_destroy_callback& cb);
    bool unregister_destroy_callback(const cci_param_destroy_callback_handle& cb);
    bool unregister_all_callbacks();
    bool has_callbacks() const;

    bool is_global_broker() const;
};
```

5.7.1.1 Construction

Construction requires either the pairing of the broker interface and the originator for the handle:

```
cci_broker_handle(cci_broker_if& broker, const cci_originator& originator);
```

or an existing handle to copy or move these attributes from:

```
cci_broker_handle(const cci_broker_handle&) = default;
cci_broker_handle(cci_broker_handle&& that);
```


5.7.1.2 Assignment

```
cci_broker_handle& operator=(const cci_broker_handle&) = default;
cci_broker_handle& operator=(cci_broker_handle&& that);
```

The presence of the = default" demonstrates that simple copying and moving of the attributes suffices.

5.7.1.3 Comparison

```
bool operator==(const cci_broker_if *b) const;
bool operator!=(const cci_broker_if *b) const;
```

Equality and inequality tests of whether this broker handle is for the given broker implementation.

5.7.1.4 Originator

The handle consists of the pairing (`cci_broker_if`, `cci_originator`), where the originator identifies the handle to delegated functions such as `set_preset_cci_value()`. This originator is accessible through:

```
cci_originator get_originator() const;
```

5.7.1.5 Delegated functions

The remainder of the class delegates predictably to the equivalent `cci_broker_if` functionality, supplying the handle's originator where a `cci_originator` is required.

5.7.2 cci_broker_manager

The mapping between `sc_objects` and `cci_broker_if` implementations is maintained by the broker manager, which provides an interface for registering new brokers and retrieving the responsible broker for the current object. The broker manager is implemented as a private class, exposing the functionality through global (non-member) functions.

```
cci_broker_handle cci_get_broker();
```

Finds the broker responsible for the current `sc_object` and returns a handle for it, using the `sc_object` also as the originator object. If there is no current `sc_object`, for example before the simulation starts and outside the construction of modules, then an error is reported. Note that the broker located may in fact be the global one.

```
cci_broker_handle cci_get_global_broker(const cci_originator &originator);
```

Returns a handle for the global broker. An error is reported if no global broker has been registered, or if the function is called with a current sc_object, for example during module construction or after `sc_start()`.

```
cci_broker_handle cci_register_broker(cci_broker_if& broker);
cci_broker_handle cci_register_broker(cci_broker_if* broker);
```

Register the given broker as being responsible for the current `sc_object`, including all sub-objects lacking a specific broker of their own. In the absence of a current `sc_object` the broker is registered as the global broker. If a broker has already been registered for the `sc_object` then that existing registration is left unchanged and an error is reported.

5.7.3 Reference brokers

`cci_utils::broker` provides the ability to selectively delegate parameters to a parent broker, by adding their name to a set of parameter names to be "exposed".

```
class broker : public consuming_broker
{
public:
  std::set<std::string> expose;
// ...
};
```


The following example shows a test module using a local `cci_utils::broker` to keep a one parameter private and make another public, the success of which is demonstrated by testing for their existence through the global broker.

Example:

```
SC_MODULE(testMod)
  {
  private:
    cci_utils::broker locBroker;
    cci_param<int>* p_private;
    cci_param<int>* p_public;
  public:
    SC_CTOR(testMod) :
      locBroker("testBroker")
      {
      cci_register_broker(locBroker);
      locBroker.expose.insert("testMod.p_public");
      p_private = new cci_param<int>("p_private", 1);
      p_public = new cci_param<int>("p_public", 2);

      sc_assert(!locBroker.param_exists("p_glob"));  // can't see into parental broker
      sc_assert( locBroker.param_exists("testMod.p_public"));
      sc_assert( locBroker.param_exists("testMod.p_private"));
      }
  };

int sc_main(int argc, char *argv[])
  {
  cci::cci_register_broker(new cci_utils::consuming_broker("Global Broker"));
  cci_param<int> p_glob("p_glob", 3, "Global param", CCI_RELATIVE_NAME, cci_originator("glob"));
  testMod tm("testMod");
  cci_broker_handle gBrok(cci_get_global_broker(cci_originator("glob")));
  sc_assert( gBrok.param_exists("p_glob"));
  sc_assert( gBrok.param_exists("testMod.p_public"));
  sc_assert(!gBrok.param_exists("testMod.p_private"));  // can only see explicity exposed param
  }
```

Note that a `cci_utils::consuming_broker` was used for the global broker since there is no possibility of delegating the parameter handling beyond it (although in fact a `cci_utils::broker` would function correctly in its place).

5.8 Error reporting

Where an application action is a definitive error, such as attempting to get a value as an incorrect type, an error diagnostic is issued through an extension of the customary SystemC `sc_report_handler::report()` mechanism with severity `SC_ERROR`. The tacit expectation is that the default `SC_THROW` handling for `SC_ERROR` is in effect. If the environment has been configured to not throw error reports then an implementation should remain functional if possible or call `cci_abort()` otherwise. "Functional" means preserving class invariants and not deceiving the client (e.g. as would be the case when returning the integer zero from an attempted `get_int()` upon a string value).

A client that wishes to handle thrown CCI error diagnostics should `catch(sc_core::sc_report&)` exceptions (or simply all exceptions) and use `cci_handle_exception()` to decode the current `sc_report::get_msg_type()` as the `cci_param_failure` enum.

```
enum cci_param_failure
 {
   CCI_NOT_FAILURE = 0,      // i.e. not a CCI-failure; some other diagnostic
   CCI_SET_PARAM_FAILURE,    // "SET_PARAM_FAILED"
   CCI_GET_PARAM_FAILURE,    // "GET_PARAM_FAILED"
   CCI_ADD_PARAM_FAILURE,    // "ADD_PARAM_FAILED"
   CCI_REMOVE_PARAM_FAILURE, // "REMOVE_PARAM_FAILED"
   CCI_VALUE_FAILURE,        // "CCI_VALUE_FAILURE"
   CCI_UNDEFINED_FAILURE,

   CCI_ANY_FAILURE = CCI_UNDEFINED_FAILURE
 };
```


The `cci_report_handler` class provides functions both for emitting CCI-specific `SC_ERROR` diagnostics and decoding an `sc_report` as a `cci_param_failure`.

```
class cci_report_handler : public sc_core::sc_report_handler
{
public:
  static void report( sc_core::sc_severity severity
                    , const char* msg_type, const char* msg
                    , const char* file, int line);

  //functions that throw a report for each cci_param_failure type
  static void set_param_failed(const char* msg="", const char* file=NULL, int line = 0);

  static void get_param_failed(const char* msg="", const char* file=NULL, int line = 0);

  static void add_param_failed(const char* msg="", const char* file=NULL, int line = 0);

  static void remove_param_failed(const char* msg="", const char* file=NULL, int line = 0);

  static void cci_value_failure(const char* msg="", const char* file=NULL, int line = 0);

  // function to return cci_param_failure that matches thrown (or cached) report
  static cci_param_failure decode_param_failure(const sc_core::sc_report& rpt);

};
```

```
cci_param_failure cci_handle_exception(cci_param_failure expect = CCI_ANY_FAILURE);
```

Can only be called with an exception in flight, i.e. from an exception handler. If the exception is both a CCI error diagnostic and once decoded as a `cci_param_failure` matches the given `expect`ed failure type then it is returned, otherwise the exception is re-thrown. Example handling where a pre-write callback may reject an update.

Example:

```
try {
  param = updatedValue;
} catch(…) {
  cci_handle_exception(CCI_SET_PARAM_FAILURE);
  gracefully_handle_update_failure();
}
```

```
cci_abort();
```

If an application determines that for CCI-related reasons (such as unrecoverable misconfiguration) it must immediately halt the simulation it should call `cci_abort()`, which may emit a suitable diagnostic before terminating via `std::terminate()` or `sc_core::sc_sbort()` where available. It may be appropriate to first issue an error report, both to better explain the violation and to allow the problem to be handled at a higher structural level once the exception has provoked suitable cleanup, e.g. abandoning the construction of an optional sub-module.

Example:

```
if(!param.get_cci_value().try_get(limit_depth)) {
  cci_report_handler::get_param_failed("Missing FooModule configuration");
  // Simulation configured with SC_THROW disabled, so object remains alive but unviable
  cci_abort();
}
```

5.9 Name support functions

Both parameters and brokers are required to have unique names relative to each other; this extends to include all named SystemC objects for version 2.3.2 and later by using `sc_core::sc_register_hierarchical_name()`. In the event of a duplicate, the given name is made unique by suffixing with a sequence number and a warning report is issued (important, since the simulation may now malfunction if the name is relied upon to find or distinguish the entity). Although this avoidance of duplicates is internal to the construction of parameters and brokers the underlying tools are exposed for client use.


```
const char* cci_gen_unique_name(const char* name);
```

Ensures that the given name is unique by testing it against the existing name registry and if necessary suffixing it with a sequence number, of format "_*n*" where n is an integer ascending from zero and counting duplicates of that specific name. The return value is a pointer to an internal string buffer from which the name must be immediately copied.

This has the explicit effect of registering the name. A name can be tested for its registration status, and if registered can be unregistered.

```
const char* cci_get_name(const char *name);
```

Verify that a name has been registered. If the given name is registered then returns it unmodified, otherwise returns `NULL`.

```
bool cci_unregister_name(const char *name);
```

If the given name is registered then removes it from the registry and returns `true`, otherwise simply returns `false`. The caller should be the owner of a name; unregistering names belonging to other entities may result in undefined behavior.

5.10 Utility definitions

5.10.1 Software version information

The header file **cci_configuration** shall include a set of macros, constants, and functions that provide information concerning the version number of the CCI software distribution. Applications may use these macros and constants.

#define CCI_SHORT_RELEASE_DATE 20171218

#define CCI_VERSION_ORIGINATOR “Accellera”

#define CCI_VERSION_MAJOR 0

#define CCI_VERSION_MINOR 0

#define CCI_VERSION_PATCH 0

#define CCI_IS_PRERELEASE 1

#define CCI_VERSION …

The macros will be defined using the following rules:

a) Each *implementation-defined_number* shall consist of a sequence of decimal digits from the character set [0–9] not enclosed in quotation marks.
b) The originator and pre-release strings shall each consist of a sequence of characters from the character set [A–Z][a–z][0–9]_ enclosed in quotation marks.
c) The version release date shall consist of an ISO 8601 basic format calendar date of the form YYYYMMDD, where each of the eight characters is a decimal digit, enclosed in quotation marks.
d) The CCI_IS_PRERELEASE flag shall be either 0 or 1, not enclosed in quotation marks.
e) The CCI_VERSION string shall be set to the value "major.minor.patch_prerelease-originator" or "major.minor.patch-originator", where major, minor, patch, prerelease, and originator are the values of the corresponding strings (without enclosing quotation marks), and the presence or absence of the prerelease string shall depend on the value of the CCI_IS_PRERELEASE flag.
f) Each constant shall be initialized with the value defined by the macro of the corresponding name converted to the appropriate data type.

Annex A Introduction to SystemC Configuration

(Informative)

This clause is informative and is intended to aid the reader in the understanding of the structure and intent of the SystemC Configuration standard. The Configuration API is entirely within `namespace cci`. Code fragments illustrating this document have an implicit `using namespace cci` for brevity.

A.1 Sample code

A.1.1 Basic parameter use

Defining a parameter and treating it like a variable:

```
cci_param<int> p("param", 17, "Demonstration parameter");
p = p + 1;
sc_assert( p == 18 );
```

A.1.2 Parameter handles

Retrieving a parameter by name and safely using the handle:

```
cci_broker_handle broker(cci_get_broker());
auto p = new cci_param<int>("p", 17);
string name = p->get_name();
// Getting handle as wrong type fails
cci_param_typed_handle<double> hBad = broker.get_param_handle(name);
sc_assert( !hBad.is_valid() );
// Getting handle as right type succeeds
cci_param_typed_handle<int> hGood = broker.get_param_handle(name);
sc_assert( hGood.is_valid() );
// Operations upon handle affect original parameter
hGood = 9;
sc_assert(*p == 9);
// Destroying parameter invalidates handle
delete p;
sc_assert( !hGood.is_valid() );
```

A.1.3 Enumerating parameters

Listing all parameter names and values for the originator "widget":

```
auto broker(cci_get_broker());
for(auto p : broker.get_param_handles(cci_originator("widget"))) {
    std::cout << p.get_name() << "=" << p.get_cci_value() << std::endl;
}
```

A.1.4 Preset and default parameter values

Setting a preset value through the broker overrides the default value provided as a constructor argument:

```
auto broker(cci_get_broker());
broker.set_preset_cci_value("module.sip", cci::cci_value(7));
auto sip = cci_param<int>("sip", 42);
sc_assert( sip == 7 );
sc_assert( sip.is_preset_value() && !sip.is_default_value() );
```

A.1.5 Linking parameters with callbacks

Uses a callback function to set parameter "triple" to three times the value of some other modified parameter:

```
void set_triple_callback(const cci_param_write_event<int> & ev) {
  auto broker(cci_get_broker());
```

```
  cci_param_typed_handle<double> h = broker.get_param_handle("m.triple");
  h = 3 * cci_param_typed_handle<int>(ev.param_handle);
}

void test() {
  cci_param<int> p("p", 0);
  cci_param<double> triple("triple", 0);
  p.register_post_write_callback(set_triple_callback);
  p = 7;
  sc_assert(triple == 21);
}
```

A.2 Interface classes

The interface classes are described in detail in the main document body; what follows here is a description of the relationships of some major classes, providing a conceptual model for locating functionality.

A.2.1 cci_value

Variant data types are provided by the `cci_value` hierarchy (depicted in Figure **1**). The encapsulated type can be:

- One of the directly supported standard data types: `bool`, `int`, `unsigned int, sc_dt::int64`, `sc_dt::uint64`, `double`, or `std::string`
- a user-defined type such as a struct, where the user provides the definition for the converter `cci_value_converter<` *`type`* `>`
- a list of `cci_value` objects (`cci_value_list`)
- a string-keyed map of `cci_value` objects (`cci_value_map`)

Accessors such as `get_int64()` retrieve the value, verifying that the type matches or trivially coerces to the accessor type. For example:

```
cci_value vi(-7);
auto i32 = vi.get_int();     // succeeds
auto i64 = vi.get_int64();   // succeeds
auto d = vi.get_double();    // succeeds
auto u64 = vi.get_uint64();  // reports CCI_VALUE_FAILURE error
```

Standard and user-defined types are set by initialization (initially through the constructor, subsequently through a setter function). `set_list()` and `set_map()` return adapter objects (`cci_value_list_ref` and `cci_value_map_ref` respectively) providing appropriate container methods:

```
cci_value val;
cci_value_map_ref vm(val.set_map());
vm.push_entry("width", 7.3);
vm.push_entry("label", "Stride");
optionClass defaultOptions;
vm.push_entry("options", defaultOptions);
```

Containers can be nested:

```
cci_value_map options;
cci_value_list enabledBits;
enabledBits.push_back(0).push_back(3);  // b01001
options.push_entry("widget0_flags", enabledBits);
enabledBits.pop_back();                 // b00001
enabledBits.push_back(4);               // b10001
options.push_entry("widget1_flags", enabledBits);
```

To make the interfaces more granular each of the `cci_value` sub-hierarchies has `_cref` classes with accessor methods and `_ref` classes with modifier methods.

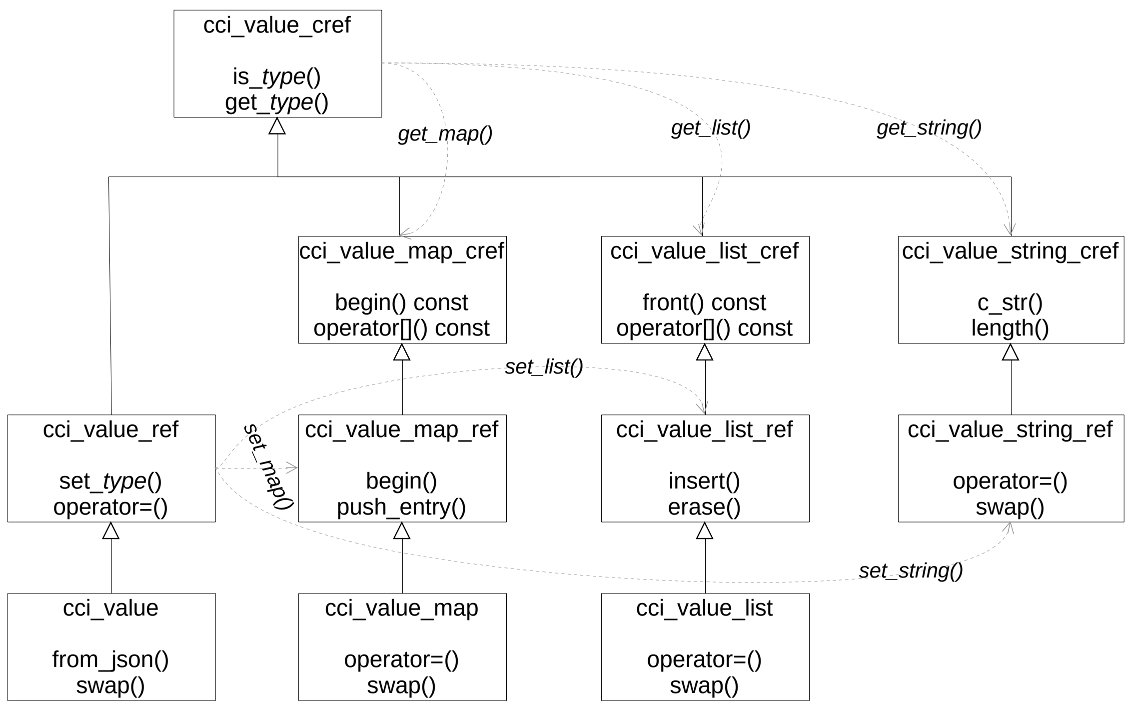


Figure 1 **- cci_value hierarchy**

A.2.2 cci_param

Parameter functionality is implemented by the small hierarchy shown in Figure 2. The final class, `cci_param_typed`, is parameterized by both data type `T` and mutability `TM` (with mutability defaulted to mutable) and is instantiated with both a name and a default value to create the parameter and add it to a broker:

- the final parameter name may include the hosting object name and a suffix to make it unique
- if no broker is specified then the broker associated with the current context is used
- a description and originator may optionally be given

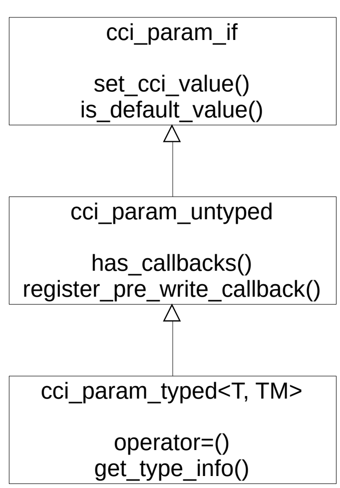


Figure 2 - cci_param hierarchy

The base class `cci_param_untyped` and the interface class `cci_param_if` provide most of the functionality free of concrete type and so are suitable for library interfaces.

For brevity `cci_param<T, TM>` is an alias for `cci_param_typed<T, TM>`, as seen in the above code samples.

A.2.3 cci_param_handle

Parameter handles provide a safe reference to parameters: safety is ensured by asserting the validity of the handle upon all operations and invalidating handles when their parameter is destroyed. Using an invalid handle results in an `SC_ERROR` report. As with parameters both untyped and typed handles exist: untyped handles are returned from parameter lookups and callbacks and typed handles provide direct access to the typed parameter value and are safely constructible from the untyped handle:

```
cci_param_typed_handle<int> val(broker.get_param_handle("mode"));
if(val != DEFAULT_MODE) { ... }
```

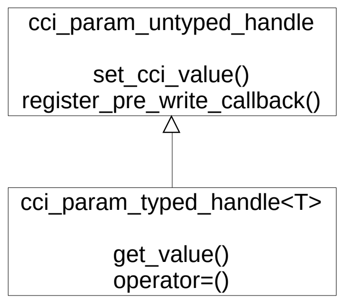


Figure 3 **- cci_param_handle hierarchy**

For brevity `cci_param_handle` is an aliased for `cci_param_untyped handle`.

A.3 Error reporting

Errors are reported through the `sc_report_handler::report()` mechanism with severity `SC_ERROR` and the message type prefixed with `__CCI_SC_REPORT_MSG_TYPE_PREFIX__` (currently "/Accellera/CCI/"). A convenience function `cci_report_handler::get_param_failure()` decodes common CCI error messages as the `cci_param_failure` enum.

Annex B Glossary

(Informative)

This glossary contains brief, informal descriptions for a number of terms and phrases used in this standard. Where appropriate, the complete, formal definition of each term or phrase is given in the main body of the standard. Each glossary entry contains either the clause number of the definition in the main body of the standard.

automatic broker: The broker that has responsibility for the current module hierarchy, obtained by calling `cci_get_broker()`. This will be the broker registered at, or most closely above, the current module hierarchy and will be the global broker in the event that no local brokers have been registered. Parameters are registered with the automatic broker at the time of their creation, unless explicitly overridden. The automatic broker is sometimes referred to as the "responsible" broker. (See 5.6.2.2)

broker: An object that aggregates parameters, providing container behaviors such as finding and enumerating, as well as managing preset values for parameters. A global broker is requisite; additional local brokers can be instantiated, e.g. to confine parameters to a sub-assembly. (See 5.7)

broker handle: An object that acts as a proxy to a broker implementation while relaying an originator representing the handle owner. (See 5.7.1)

broker manager: A private singleton class accessed via global functions to register brokers, using `cci_register_broker()`, and retrieve the currently responsible broker, using `cci_get_broker()`. (See 5.7.2)

callback: A function registered to be invoked when a particular action happens. Both brokers and parameters support callbacks to enable custom processing of actions of interest, such as the creation of a new parameter or accessing a parameter value. (See 5.4.3.6 for broker callbacks and 5.4.2.8 for parameter callbacks)

global broker: This broker must be registered before any parameters are constructed and it has responsibility (1) outside of the module hierarchy and (2) for all module hierarchies that have no registered local broker. A global broker handle is obtained outside the module hierarchy by calling `cci_get_global_broker()`; within the module hierarchy, it is returned by `cci_get_broker()` when appropriate. (See 5.7)

local broker: A broker explicitly registered at a specific level in the module hierarchy, becoming the automatic broker for that module and submodules below it that don't register a local broker themselves. (See 5.7)

originator: An object used to identify the source of parameter value and preset value changes. Originators are embedded within handles allowing source identification to be provided in a largely implicit manner. (See 5.4.1)

parameter: An object representing a named configuration value of a specific compile-time type. Parameters are typically created within modules from which their name is derived, managed by brokers, and accessed externally via parameter handles (See 5.6)

parameter default value: The value provided as an argument to the parameter's constructor. This value is supplanted by the preset value, when present. (See 5.4.2.3)

parameter handle: An object that acts as a proxy to a parameter while relaying an originator representing the handle owner. Parameter handles can be either untyped (See 5.6.3) or typed (See 5.6.4).

parameter value: The current value of the parameter, accessible in either an untyped or typed manner. (See 5.4.2.1)

parameter value originator: The originator that most recently set the parameter's value. (See 5.4.2.3)

parameter preset value: A value used to initialize the parameter, overriding its default value. Preset values are supplied to the appropriate broker prior to parameter construction. (See 4).

parameter underlying data type: The specific compile-time type supplied as a template instantiation argument in the parameter's declaration. Syntactically, this is referenced as the parameter's `value_type`. (See 5.6.2.1)

typed parameter access: Using interfaces based on the parameter's underlying data type to access a parameter value. (See 0)

untyped parameter access: Using interfaces based on `cci_value` to access a parameter value. (See 5.4.2.1)

Annex C SystemC Configuration modeler guidelines

(Informative)

The following guidelines are provided to help ensure proper and most effective use of this standard.

C.1 Declare parameter instances as `protected` or `private` members

Making parameters non-`public` ensures they are accessed via a handle provided by a broker, adhering to any broker access policies and properly tracking originator information.

C.2 Initialize broker handles during module elaboration

Broker handles should be obtained, and stored for later use, during elaboration when the well-defined current module can be used to accurately determine implicit originator information.

C.3 Prefer typed parameter value access over untyped, when possible, for speed

When a parameter's underlying data type is known, access via the typed handle is preferred over the untyped handle since it avoids the overhead associated with cci_value conversions.

C.4 Prefer CCI parameters over module constructor parameters

CCI parameters provide configurability without requiring recompilation. Module instantiation code can still manipulate the parameter either by setting its preset value (pre-construction, subject to broker conflict resolution) or its value (post-construction, ensuring an attempt to set the supplied value).

C.5 Provide parameter descriptions

Providing a description of parameters, which can only be done during parameter construction, is recommended when the parameter's purpose and meaning are not entirely clear from the name. Tools can relay descriptions to users to give insights about parameters.

Annex D Enabling user-defined parameter value types

To be able to instantiate a `cci_param_typed` with some user-defined type "VT", that type must provide these features:

- default constructor: `VT()` (DefaultConstructible in C++ concept terminology)
- copy constructor: `VT(const VT&)` (CopyConstructible)
- value type assignment operator: `operator=(const VT&)` (CopyAssignable)
- value type equality operator: `operator==(const VT&)` (EqualityComparable)
- `cci::cci_value_converter<value type>` defined

The following example takes a small class `custom_type`, the pairing of an integer and string, and enables use such as:

```
custom_type ct1(3, "foo");
cci_param<custom_type> pct("p1", ct1);
custom_type ct2 = pct;
```

Emphasized in italics below is the added support code - note it relies on the default copy constructor and assignment operator – defining these as "=delete" demonstrates their necessity.

```
class custom_type
  {
  private:
    int val_;
    string name_;
    friend class cci_value_converter< custom_type >;
  public:
    custom_type()
      : val_(0) {}
    custom_type(int val, const char* name)
      : val_(val), name_(name) {}
    bool operator==(const custom_type& rhs) const
      {
      return val_ == rhs.val_ && name_ == rhs.name_;
      }
  };

template<>
struct cci_value_converter< custom_type >
  {
  typedef custom_type type;
  static bool pack(cci_value::reference dst, type const & src)
    {
    dst.set_map()
      .push_entry("val", src.val_)
      .push_entry("name", src.name_);
    return true;
    }
  static bool unpack(type & dst, cci_value::const_reference src)
    {
    // Highly defensive unpacker; probably could check less
    assert(src.is_map());
    cci_value::const_map_reference m = src.get_map();
    return m.has_entry("val")
      && m.has_entry("name")
      && m.at("val").try_get(dst.val_)
      && m.at("name").try_get(dst.name_);
    }
  };
```

There is no explicit stability requirement for the packing and unpacking operations; for example it is not required that:

```
T x;
cci_value vX(x);
T y = vX.get<T>();
sc_assert(x == y);
```


and for some data types such as floating point it may not be practicable, nor desirable to encourage thinking of equality as a useful concept when comparing types. However in general such behavior may astonish users, so stability may be a sensible default goal.

Index