2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Synthesizable Subset Update

Peter Frey, HLS Technologist

Mentor Graphics®

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Synthesizable Subset Goal

- The standard is intended for use by hardware designers and HLS tool developers in a manner that allows hardware designers to create HLS models in SystemC that will be portable among all conforming HLS tools.

- Approved Version 1.4.7 (March 2016)
  - Covers C++ and SystemC supported and unsupported language constructs
  - Focus on supported language syntax

Some SystemC Synthesizable Subset Rules

```
#include <systemc.h>
SC_MODULE (RndGen) {
  sc_in<bool>           clk;
  sc_in<bool>           rst;
  sc_in<bool>           rdy;
  sc_out<sc_uint<48> > data;
  sc_out<bool>          vld;
  sc_uint<48> a;
  sc_uint<48> c;
  sc_uint<48> val;

  SC_CTOR(RndGen) :
    clk("clk"), rst("rst"),
    rdy("rdy"), data("data"), vld("vld")
  {
    SC_CTHREAD(process, clk.pos());
    reset_signal_is(rst, true);
  }
```

Required signal-based communication channels and ports

Support for most SystemC and C++ datatypes (excluding float/double)

Local variables can only be accessed by a single process.
No support for non-constant globals

Full support for SC_(C)THREAD and SC_METHODS

Setting of clock and reset including polarity

Behavior-Specific Modeling Rules

Reset behavior:
Modeled in process before first wait

Process behavior cannot terminate

Explicit modeling of behavior:
- **Practically full C++ / SystemC operator support**
- **No system library support (for example: math.h)**

Explicit modeling of protocols required based on signals

Only wait() and wait(in) on clock supported

```
void process() {
   a = 25214903917;
   c = 11;
   val = 17;
   vld.write(0);
   data.write(0);
   wait();
   while(1) {
     val = a*val + c;

     data.write(val);
     vld.write(true);
     wait();
     while(!rdy.read()) {
      wait();
     }
     vld.write(false);
   } } };
```

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Ongoing Activities

NOT COMMITTED TO ANY SPECIFIC VERSION

- Newer language constructs C++-11
- Enhanced bit-accurate datatype modeling
- Protocol libraries
- Latency constraints
- Common synthesis control structures

C++-11

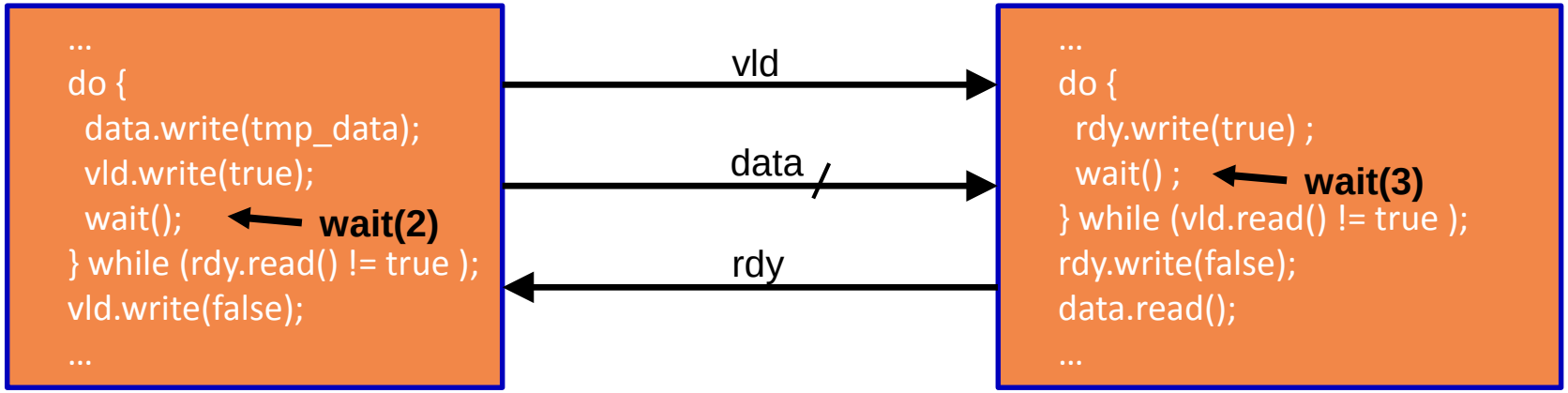
- Extended C++ Standard (C++-11/C++-14) is used more commonly in model descriptions:
  - auto, decltype
    ```
    auto a = 1 + 2;
    ```
  - constexpr
    ```
    constexpr int Fahrenheit(int c) { return c *9/5 +32;}
    const temp = Fahrenheit(20);
    ```
  - template aliasing
    ```
    template<int W> intArray = myArray<W, int>;
    intArray<10> a;
    ```
  - variadic templates
    ```
    template<class … Types>
    struct Tuples {};
    ```
  - Initializer list
    ```
    int a{0};
    ```
  - lambda functions:
    ```
    int func = [](int i)
        { return i+8;};
    ```

Active discussion in SystemC Synthesis Working Group (SWG)

Bit-accurate Datatypes

- Mentor released Algorithmic Datatypes
  - Library of bit-accurate datatypes
  - Licensed under Apache Version 2.0
- Used even in SystemC HLS models due to:
  - Arbitrary-Length
  - Precise and Consistent Definition of Semantics
  - Simulation Speed
- Accellera SystemC Datatype Working Group (SDTWG) has been formed to address SystemC datatype issues and update SystemC standard

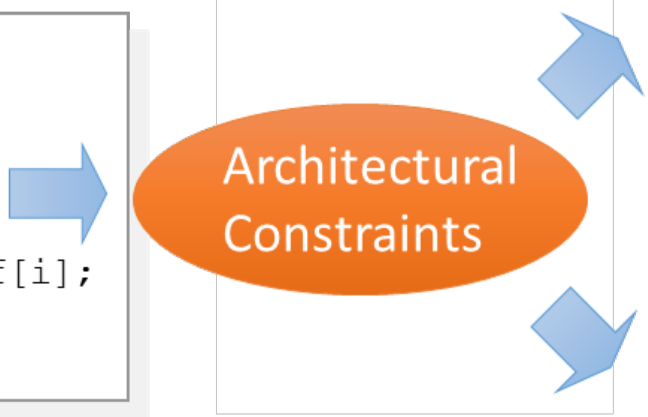
SystemC Interface / Latency Modeling

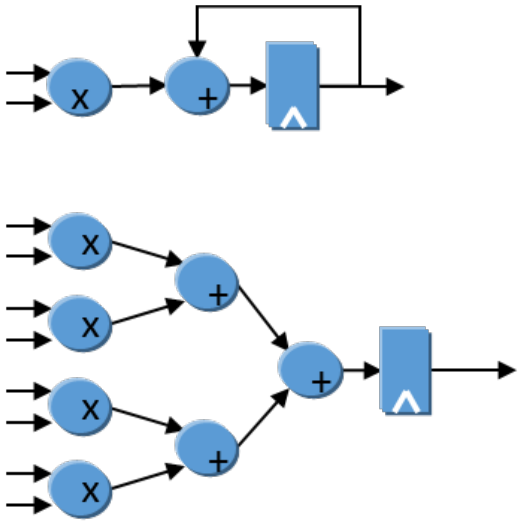


- Synthesis adds latency
- wait statements in SystemC are at best minimum latency boundaries for synthesis
- Acknowledged issue (timeline TBD)

Common Synthesis Control Structures

```
int mac( char data[N],
         char coef[N]  )
{
  int accum=0;
  for (int i=0; i<N; i++)
    accum += data[i] * coef[i];
  return accum;
}
```





- In many cases designers know the desired implementation and would like to fix it directly in the synthesis source
- Acknowledged issue (timeline TBD)

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Conclusion

- SystemC Synthesizable subset clearly defines the SystemC/C++ syntax supported by compliant synthesis tools

- SystemC standardization needs to go further to capture synthesis specific constructs / libraries

Active community driving the standardization effort with the goal to ease full HLS SystemC adoption

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank You!

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC Configuration Tutorial

A preview of the draft standard

Trevor Wieman, SystemC CCI WG Chair

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC 2.3.2 Public Review

- SystemC 2.3.2 public review release available
  - Maintenance release with some new features
  - Licensed under Apache 2.0
  - www.accellera.org/downloads/drafts-review
  - Public review period until **May 31st 2017**
  - Call for testing and feedback
  - Send feedback to review-systemc@lists.accellera.org
  - forums.accellera.org/forum/10-systemc-language/

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemC 2.3.2 – Highlights

- Foundation for C++11/14 enablement
  - Based on DVCon Europe 2016 proposal *Moving SystemC to a New C++ Standard*
- Centralized global name registry
  - Enables CCI naming requirements
- New TLM socket and sc_signal base classes
- Updated compiler and platform support
  - Windows DLL support
  - Experimental CMake build system
- Tons of bug fixes and cleanups

Acknowledgements

ERICSSON

GreenSocs®

DOULOS

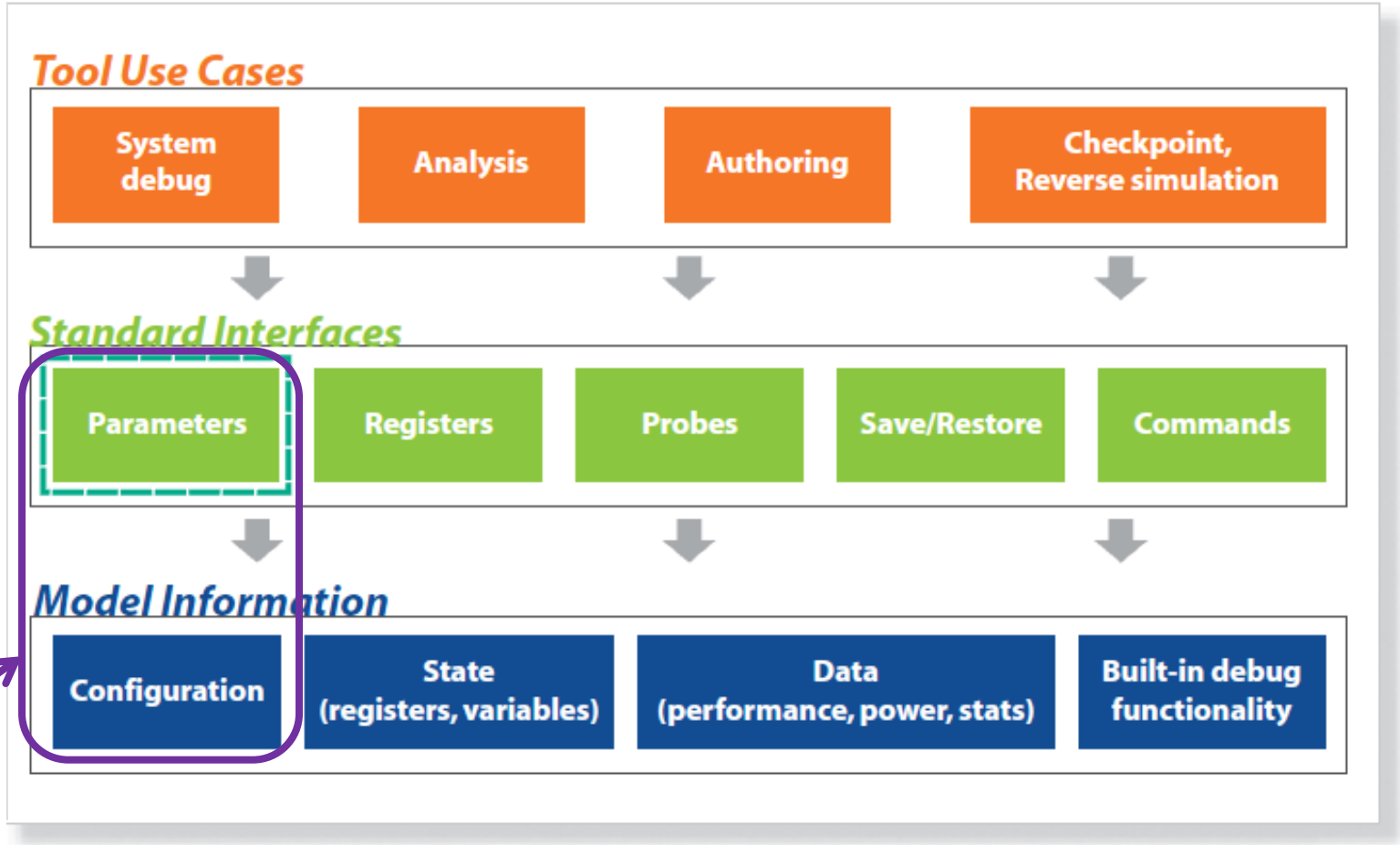
This tutorial builds on content contributed to the Accellera SystemC CCI WG by Ericsson, GreenSocs and Doulos.

Progress on the Configuration draft standard has been significantly accelerated through funding from Ericsson

Config, Control, Inspection WG

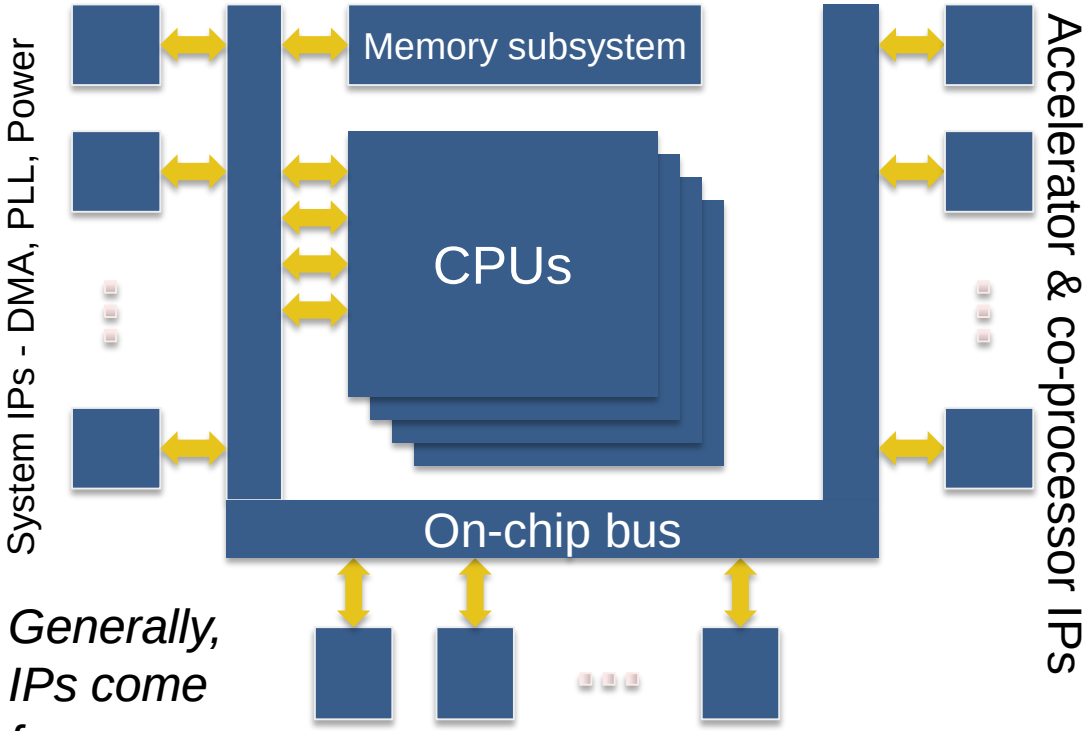
WG is defining these

Public review draft



Goal: Standardizing interfaces between models and tools

Parameterizing a System



Generally, IPs come from many vendors

Parameter Examples

- system clock speed
- # processor cores
- memory size
- address, data widths
- disabled IP(s)
- address maps
- SW image filename
- IP granularity debug control*:
  - logging
  - tracing

** runtime parameters provide initial CCI "control" capability*

Need uniform way to configure simulation without recompilation

CCI Environment

- CCI requires SystemC 2.3.1 and works better with the forthcoming 2.3.2 release
- In order to use CCI classes, a header must be included

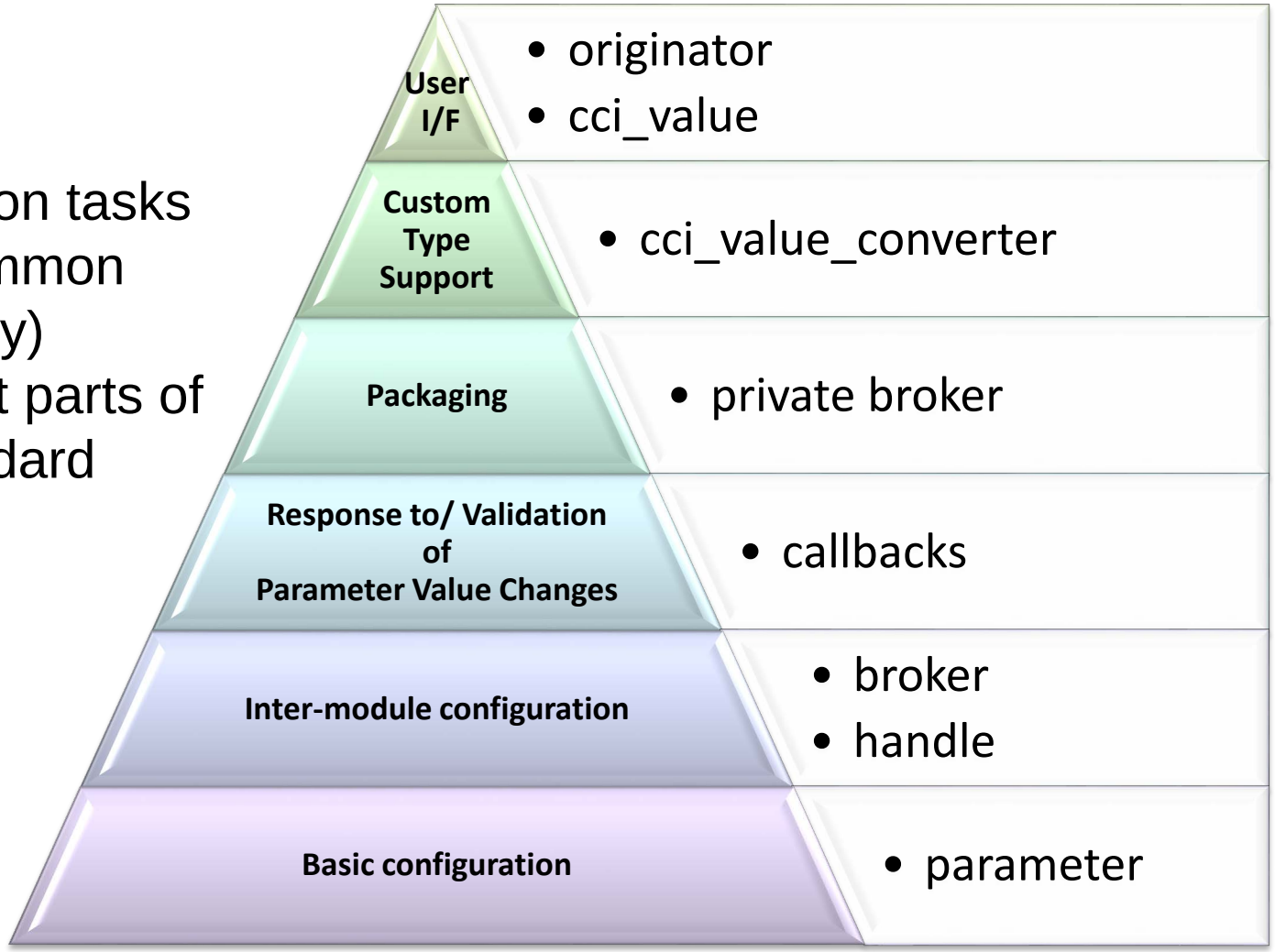
```
#include <cci_configuration>
```

- As with SystemC, CCI code is defined in a specific namespace

```
namespace cci;
```

Configuration Overview

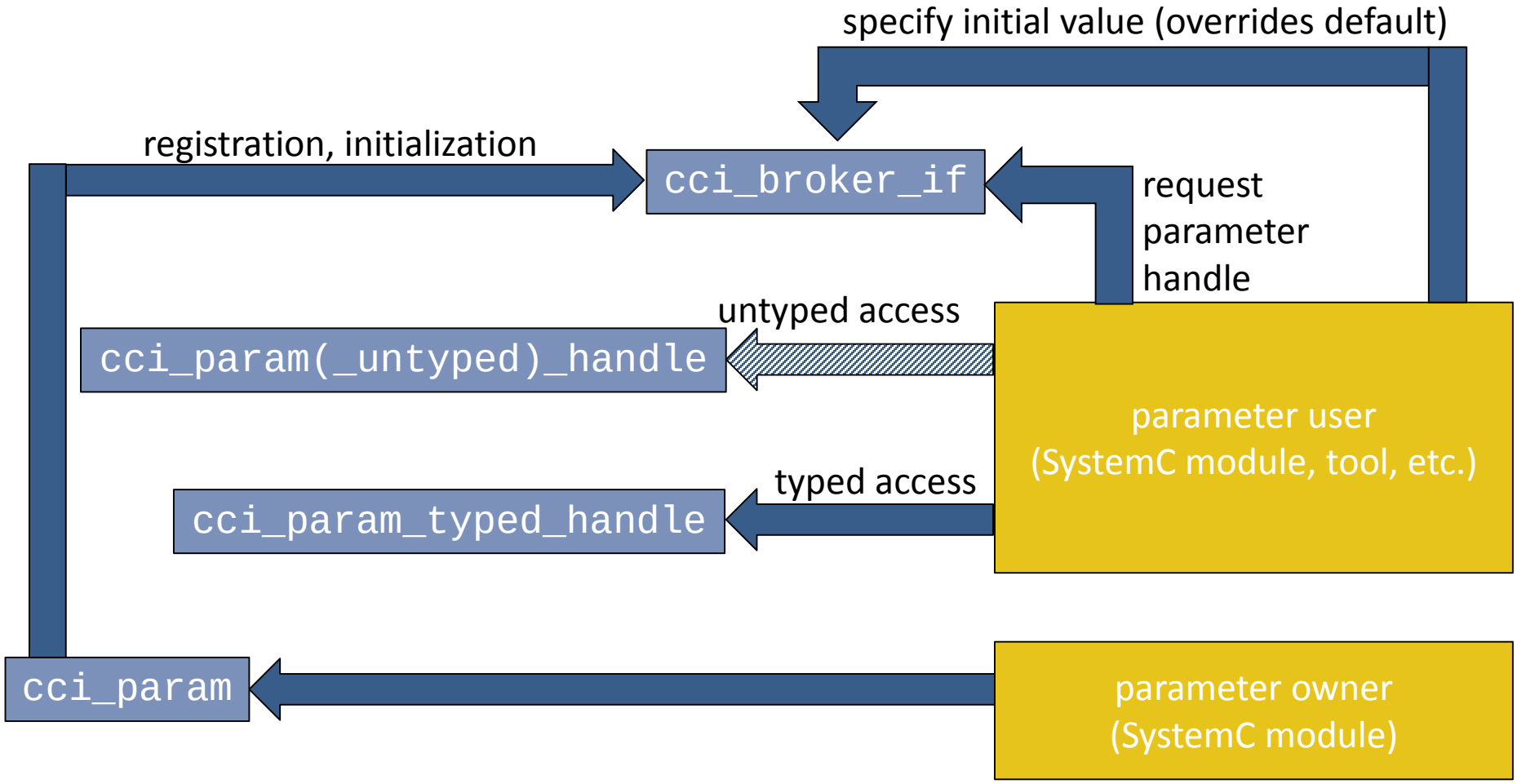
Configuration tasks
- How common (relatively)
- Relevant parts of the standard



Key Configuration Components

- Parameter
  - consists primarily of a name (string) and a value
  - is an instance of `cci_param<T>` (T is the value type)
  - registers with a broker at construction
  - provides 2 interfaces to set/get values
    - "untyped": uses variant type; interoperable with JSON
    - "typed": a templated interface using instantiated type T
- Broker
  - Manages access to parameters registered with it
    - Used by both models and infrastructure/tools
  - Two kinds of brokers:
    - There is one global (public) broker
    - Any number of private brokers may also exist

Key Classes & Interactions



Default and Initial Values

- The broker allows an INITIAL parameter value to be specified prior to the parameter's construction:

```
cci::cci_get_broker().
     set_initial_cci_value("param_name", cci::cci_value(10));
```

- When you create a parameter, you must specify a DEFAULT value:

```
cci::cci_param<int> my_param("param_name", 42);
```

- The parameter will use the INITIAL value if there is one, otherwise it will use the DEFAULT value.

```
std::cout << my_param.get_value(); // Output = 10
```

Actual vs. Variant Value Types

There are two ways to access parameter values:

- **Untyped**: using a variant type, `cci_value`
  - Complex types emulated as collection of primitive types
    - Built-in support for basic and SystemC type conversions
    - Extensible to support user-defined types
  - Important for tool enabling, for example:
    - Applying initial values from an ASCII configuration file
    - Presenting/validating values of complex parameter types
- **Typed**: using the template instantiated value type
  - More direct (and efficient) when value type is known

Note: `cci_value` may be promoted to core language as `sc_variant`.

Originator

The origin of a parameter's current value is always known; the `cci_originator` class is used for this

- Within the SystemC module hierarchy, originator modules are determined automatically
  - e.g. "top.platformX.subsystemA.dma1"
- Originators can, and outside the SystemC module hierarchy must, be set explicitly (unrestricted text)
  - e.g. "platformX_basic_configuration.cfg"
    Indicating the value came from a configuration file
  - e.g. "sim_user"
    Indicating the value was set interactively

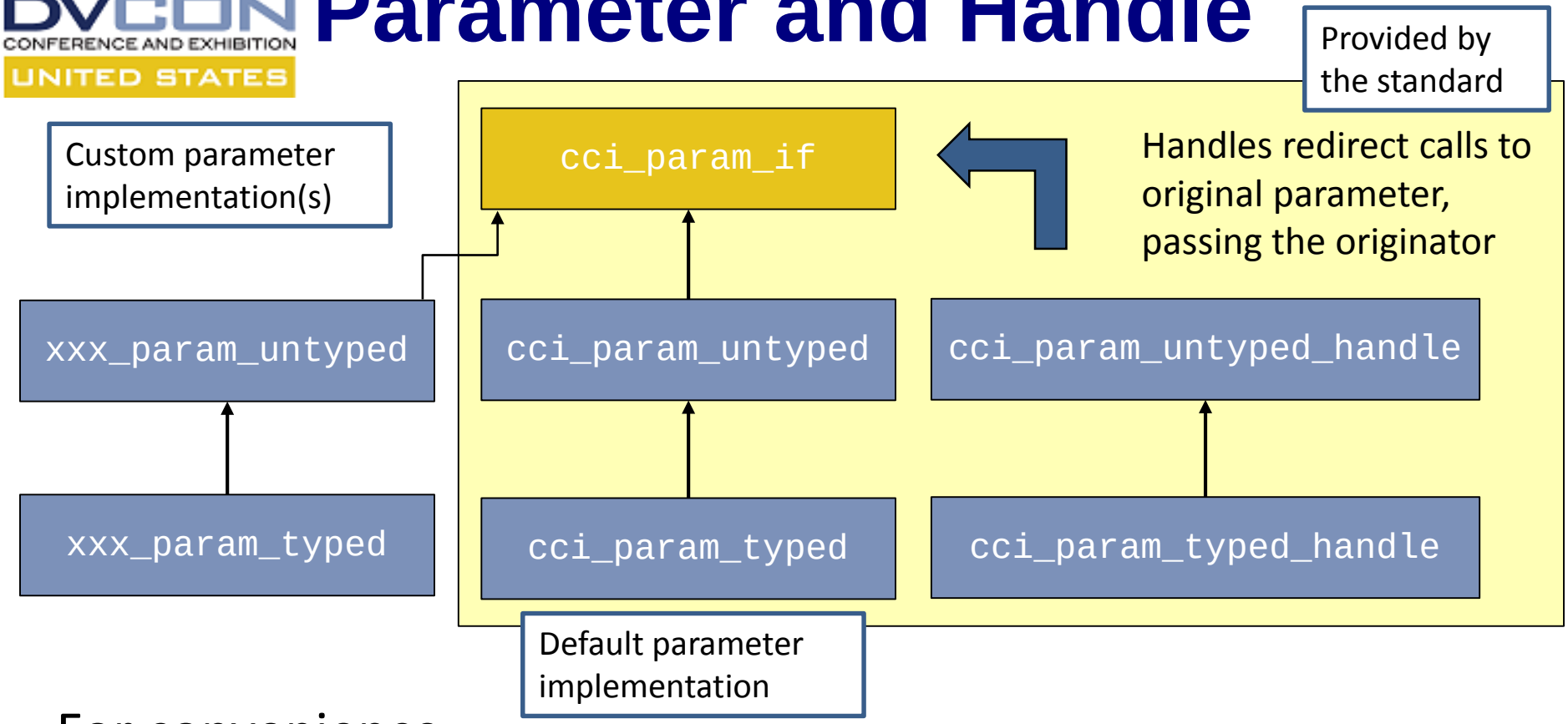
Originators are generally managed behind-the-scenes

Parameter Handles

- Returned by a broker for name-based parameter lookup
- Provides a parameter-like interface
- May impose read-only access restrictions
  - Determined by broker based on handle originator
- Informs parameter of originator when value is updated
- Available in both untyped and typed forms:

```
class cci_param_untyped_handle;
template<typename T> class cci_param_typed_handle;
```

Parameter and Handle



For convenience:

```
typedef cci_param_untyped_handle cci_param_handle
template<typename T> using cci_param = cci_param_typed<T>;
(pre-C++11: #define cci_param cci_param_typed)
```

Standard vs. Implementation

- Standard:
  - Parameter and broker interface
  - Default implementation of `cci_param`
  - Originator and Handle
  - Callback Infrastructure
  - `cci_value`
- Vendor Implementation:
  - Broker
  - Private broker
  - Support for configuration files (xml, conf…)

A Parameter Owner Module

```
SC_MODULE(simple_ip) {

private:
 cci::cci_param<int> int_param;

public:
 SC_CTOR(simple_ip)
 : int_param("int_param", 0)
 {
   int_param.set_description("...");
   SC_THREAD(do_proc);
 }
 void do_proc() {
   for(int i = 0; i < int_param; i++) {
     ...
   }
 }
};
```

Parameters are usually private members forcing brokered access

Default values must be supplied using the constructor argument

Owner may read parameter's value

Accessing Parameter Values

- When value type is known, call parameter's `set_value(T val)` or `get_value()` function
  - Common C++ types
  - SystemC Data types
  - Extensible to user-defined types
- When parameter type is unknown or unsupported:
  - Use `cci_value` representation (variant type)
  - `set_cci_value()`
    - Takes variant typed value; fails if incompatible contents
  - `get_cci_value()`
    - Returns variant typed value

Broker Lookup of Params (1)

```
SC_MODULE(configurator) {

 cci::cci_broker_handle m_brkr;


 SC_CTOR(configurator)
 : m_brkr(cci::cci_get_broker())
 {
   sc_assert(m_brkr.is_valid());

   SC_THREAD(do_proc);
 }
…
```

Declaration of broker handle variable

Get handle to broker associated with this module

Broker Lookup of Params (2)

```
void do_proc() {
 const std::string int_param_name = "top.sim_ip.int_param";

 cci::cci_param_handle int_param_handle =
     m_brkr.get_param_handle(int_param_name);

 if(int_param_handle.is_valid()) {

   cci::cci_value value = int_param_handle.get_cci_value();
   value = value.get_int() + 1;
    ...

   int_param_handle.set_cci_value(value);

    ...
  }
```

Get handle to named parameter from broker

Check handle validity / parameter exists

Get current parameter value

Set new parameter value

Parameter Mutability

- Parameters are mutable by default
- Mutability is set by template parameter

```
cci::cci_param<int, CCI_MUTABLE_PARAM> p1;
```

- Parameters may also be permanently immutable or immutable after elaboration

```
cci::cci_param<int, CCI_IMMUTABLE_PARAM> p2;
cci::cci_param<int, CCI_ELABORATION_TIME_PARAM> p3;
```

Elaboration parameters should be used to configure the simulated system's structure

Description and Metadata

- CCI parameters have a description intended to explain their purpose and usage to a simulation user. It can be supplied either as a constructor argument or with a setter.

```
my_param.set_description("Clock frequency");
```

- CCI parameters also have an array (Array of CCI Value) of meta information. You can use this as you see fit!

```
my_param.add_metadata("units", "V", "Units of the value");
my_param.get_metadata(); // Return a map of CCI Value
```

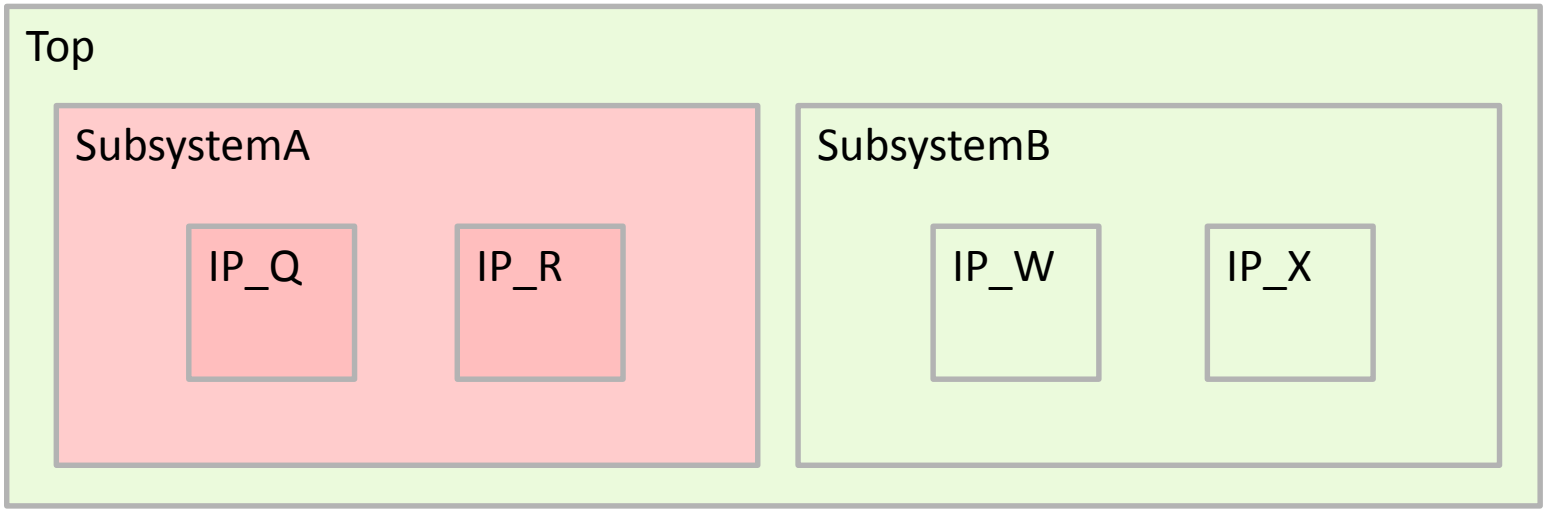
Private Brokers

- Each parameters is registered with a single broker
- Private brokers are assigned to a module hierarchy
  - Contained parameters are registered with that broker by default
  - Parameters can still be explicitly registered with the global (or any other) broker
- Private brokers are undiscoverable outside of their associated module hierarchy
  - Even undiscoverable by tools
- Parameters inaccessible from outside that hierarchy
- Private brokers facilitate encapsulation of IP configuration
  - E.g. a configurator that applies pre-compiled configuration

Private Broker Example

```
SC_CTOR(SubsystemA)
 {
   cci_register_broker(my_priv_broker);
 }
```

Only module itself can specify its broker



- parameters managed by global broker
- parameters managed by my_priv_broker

Callbacks

- Used to track parameter value changes: pre-read, post-read, pre-write and post-write
- Parameter creation and destruction callbacks are also available through the broker
- Callbacks contain a payload and can return a value
- Callback payloads can be untyped or typed
- Compatible with C++ lambdas, function objects
- Internal callback mechanism provided by the standard

Note: The callback mechanism may be provided more widely across SystemC in the future.

Parameter Owner Callback

```
SC_MODULE(simple_ip) {

private:
 cci::cci_param<int> P1;
 cci::cci_callback_untyped_handle P1_cb;

public:
 SC_CTOR(simple_ip): P1("P1", 0) {
  P1_cb = P1.register_post_write_callback(&simple_ip::cb,
  this);
   ...
   }

void cb(const cci::cci_param_write_event<int> & ev);
…
```

Callback handle

Post-write callback registered in constructor

Post-write callback function with the write event as parameter

Callback Events

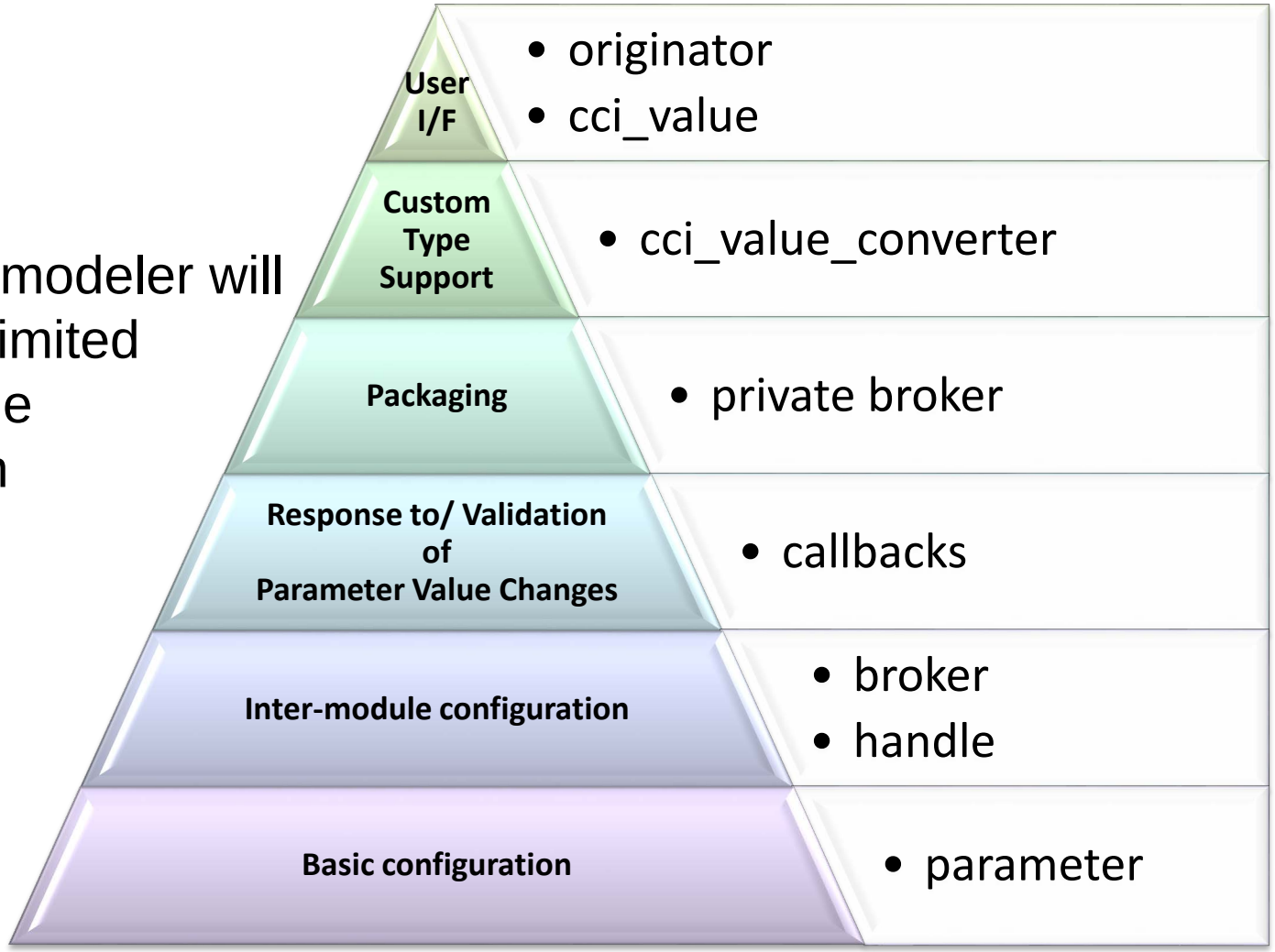
- Callback `pre_write` and `post_write` event:
  - Contains an untyped parameter handle, the old value, new value and the originator.
- Callback `pre_read` and `post_read` event:
  - Contains an untyped parameter handle, the value and the originator.
- Callback `create_param` and `destroy_param` event:
  - Contains the untyped parameter handle
- Support for lambdas/function objects/`sc_bind` allows customization for different signatures and stateful callbacks

Custom Types

- Support for legacy parameter implementation is done via the `cci_param_if` interface
  - Explicit registration with the broker is required
- `cci_value` provides an extensible infrastructure to add packing/unpacking support for custom C++ datatypes
- When a custom C++ data type is extended with `cci_value` support, it can be transparently used with `cci_param`

Summary Usage

The typical modeler will use only a limited subset of the standard on a routine basis.



2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Looking Forward

- Beyond Configuration, the CCI WG will prioritize and begin work on “Control” and “Inspection” standards

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank You!

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

UVM-SystemC Standardization Status and Latest Developments

Trevor Wieman, SystemC CCI WG Chair

Slides by Michael Meredith, Cadence Design Systems

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Outline

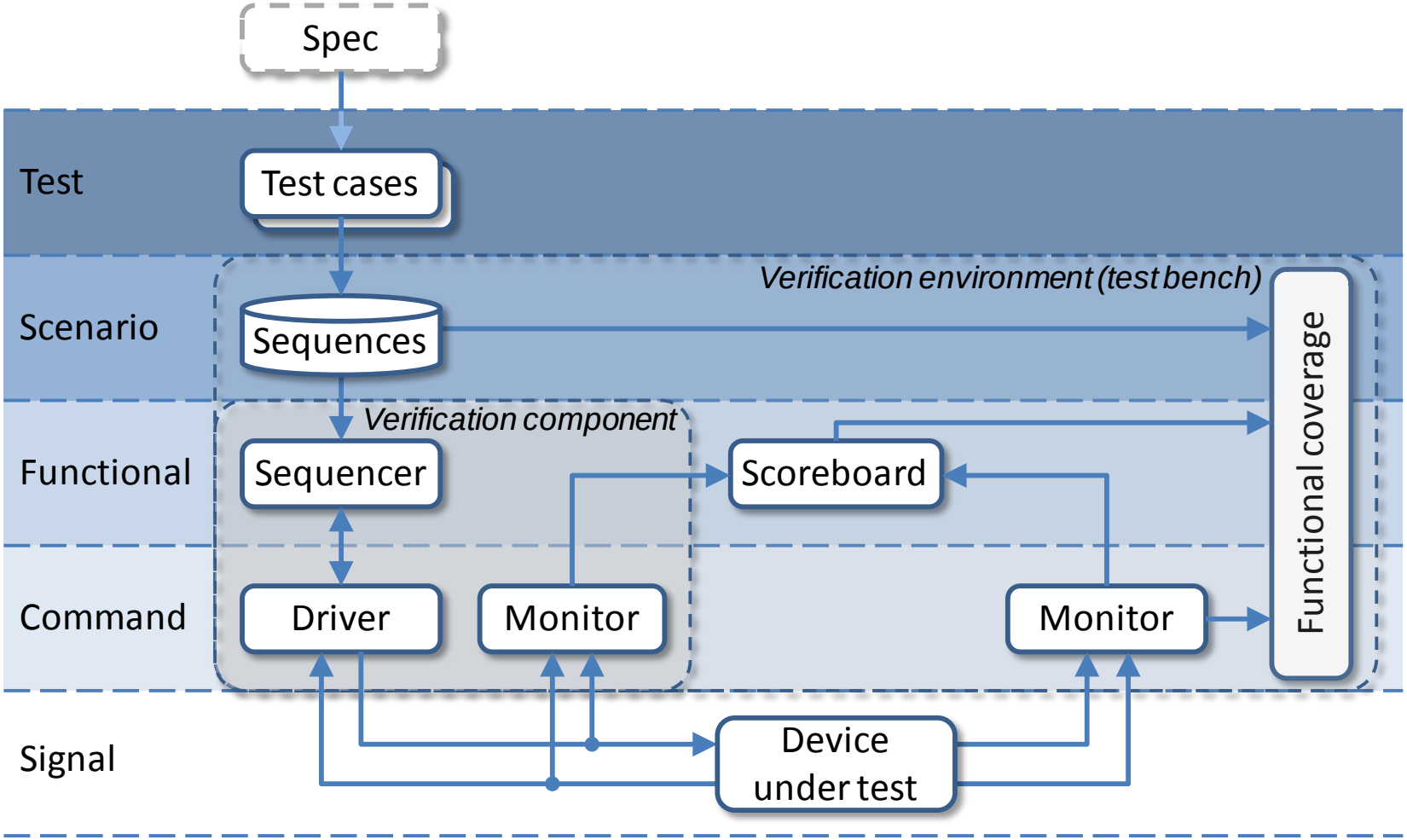
- Why UVM-SystemC?
- UVM layered architecture
- UVM-SystemC LRM + proof-of-concept library
- Status Accellera standardization
- Next steps
- You can help!

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Why UVM-SystemC?

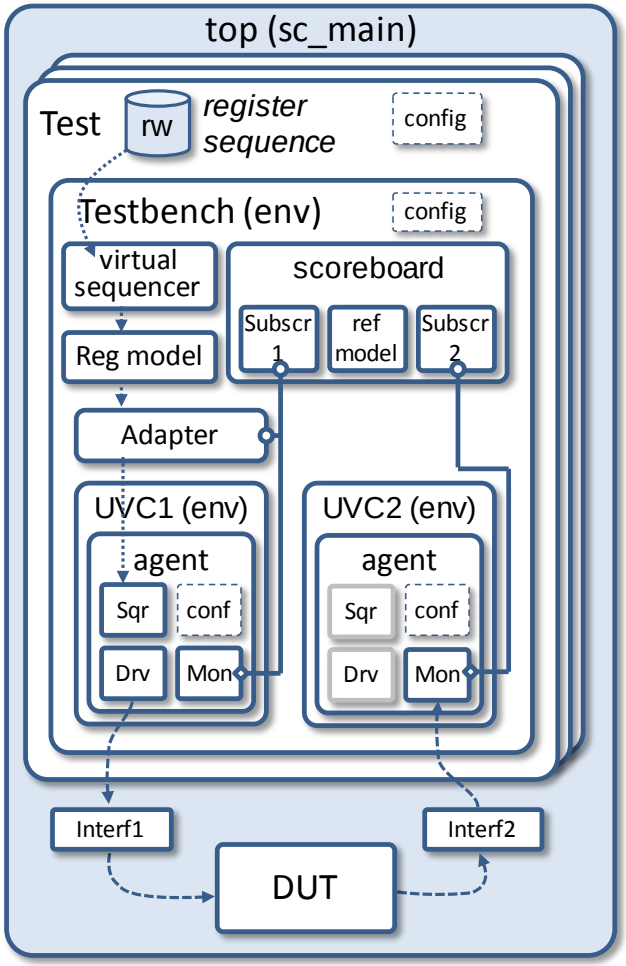
- Current ESL focus is on *system design* not on *system verification* – step-up is required!
- Testbenches and tests for system design are often ad-hoc, unstructured and incomplete
- No reuse of system verification components and environment – missing link to RTL world
- UVM-SystemVerilog lacks native C/C++ integration to support SW-driven verification use cases
- Reuse testbenches and tests in HW prototyping, equipment and Hardware-in-the-loop (HiL) setup

UVM Layered Architecture (1)



UVM Layered Architecture (2)

- The top-level contains the test(s), the DUT and interfaces
- DUT interfaces are stored in a configuration database, used by the UVCs to connect to the DUT
- The testbench contains the UVCs, register model, scoreboard and sequencer to execute the stimuli and check the results
- (Almost) everything is configurable



2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Standardization Status

- Draft 1.0 standard released early 2016
  - Language Reference Manual as public review
  - Alpha release of Proof-of-concept implementation
- Current standardization focus
  - Register Abstraction Layer
  - Constrained Randomization (using CRAVE)
  - Reporting infrastructure
  - Test and regression environment

Register Abstraction Layer

Register Abstraction Layer	Status
Register model containing registers, fields, blocks, etc.	testing
Register callbacks	testing
Register adapter, register sequences and transaction items	testing
Register frontdoor access	testing
Build-in register test sequencers	development
Register backdoor access (hdl_path)	testing
Memory and memory allocation manager	development
Virtual registers and fields	development
Randomization of register content	development

Register Model Example

```
class reg_a : public uvm_reg
{
 public:
  uvm_reg_field* F1;
  uvm_reg_field* F2;

  UVM_OBJECT_UTILS(reg_a);

  reg_a( uvm_object_name name = "a" )
  : uvm_reg(name, 32, UVM_NO_COVERAGE) {}

  virtual void build()
  {
    F1 = uvm_reg_field::type_id::create("F1");
    F1->configure(this, 8, 0, "RW", false, 0x0, true, false, true);
    F2 = uvm_reg_field::type_id::create("F2");
    F2->configure(this, 8, 16, "RO", false, 0x0, true, false, true);
  }
};
```

UVM register class

Register A contains two fields, F1 and F2

Use of the UVM factory to instantiate the register fields

Work-in-progress: API might change!

Constrained Randomization

- UVM-SystemC will introduce constrained randomization by using a library called CRAVE:
- **C**onstrained **Ra**ndom **V**erification **E**nvironment
  - Powerful & extensible constrained random stimuli generator
  - Leverage latest constraint solving technologies
  - C++11 syntax for constraint definition
- More information
  - www.systemc-verification.org/crave

Randomization Example

```
class ubus_transfer : public uvm_randomized_sequence_item {
public:
  crv_variable<ubus_rw_enum> read_write;
  crv_variable<sc_bv<16> > addr;
  crv_variable<unsigned> size;
  crv_vector<sc_bv<8> > data;
  crv_vector<sc_bv<4> > wait_state;
  ...

  crv_constraint c_read_write { inside( read_write(),
    std::set<ubus_rw_enum> {
      ubus_rw_enum::READ, ubus_rw_enum::WRITE })};

  crv_constraint c_size { inside( size(),
    std::set<int> { 1, 2, 4, 8 } )};

  crv_constraint c_data_wait_size {
    data().size() == size(),
    wait_state().size() == size() };
...
```

CRAVE variables and vectors

Constraint definitions

Work-in-progress: API might change!

Reporting Infrastructure

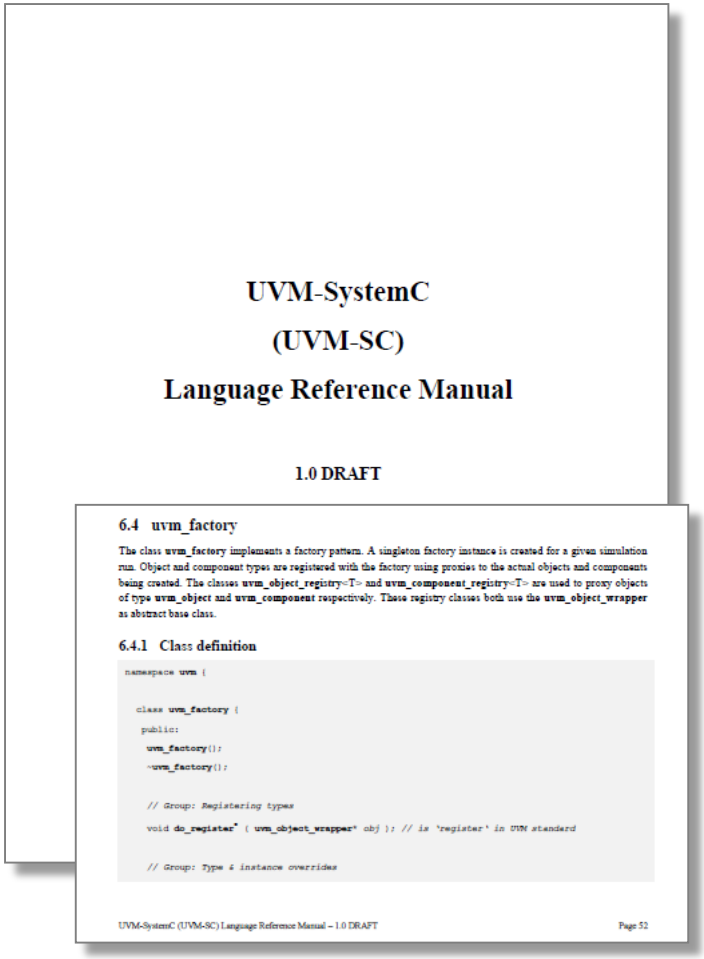
- Reporting infrastructure upgraded to UVM 1.2 capabilities
  - Reporting mechanism is now object oriented using the new UVM report message object
  - All std::cout have been replaced by UVM_INFO messages, enabling visibility (verbosity) control
- For the user, no big difference in usage of the API
- Note: UVM report message object extension macros (e.g. UVM_MESSAGE_ADD_INT) not yet available

Test and Regression Environment

- Simple examples available in PoC to demonstrate basic functionality (component instantiation, factory, configuration, reporting, phasing, sequences, etc.)
- Unit tests developed as part of a regression suite – closely following the UVM-SystemVerilog tests
- More integrated examples under development (e.g. ubus and codec example)
- Adding new unit tests for register model testing

Next Steps

- Update LRM
  - Register abstraction classes
  - Reporting classes
- Constrained randomization API and functional coverage
- Introduction of UVM-SystemC in Accellera Multi-language verification standardization
- Align configuration concept with SystemC CCIWG



2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

You Can Help!

- Lots of things to do to grow UVM-SystemC
- How you can contribute
  - Join Accellera and participate in the SystemC Verification Working Group
  - Development of unit tests, examples and applications
  - Contribute to proof-of-concept development: robustness, quality, maturity

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Thank You!