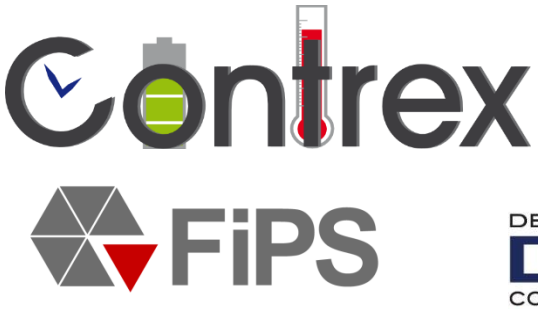
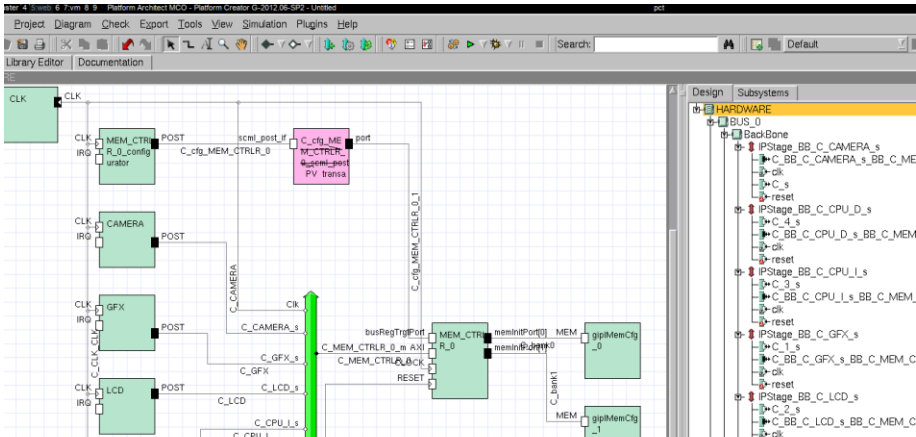
Automated SystemC Model Instantiation with modern C++ Features and sc_vector

Ralph Görgen, OFFIS, Oldenburg, Germany
Philipp A. Hartmann, Intel, Duisburg, Germany
Wolfgang Nebel, CvO University Oldenburg, Germany



Automated Model Instantiation

- VPs are not implemented manually today
- Set up from configurable library components
- Automatic instantiation according to platform description
- Examples:
  - Open Virtual Platform (OVP)
  - Cadence Virtual Platform Catalogue
  - Synopsys Platform Architect
  - FiPS Platform Simulation



Configurable Library Components

- Requires configurable library components
  - Configurable interface
  - Configurable subcomponents
  - Configurable sub-behaviour

- **`sc_vector`** helps to implement configurability
- C++11 and C++14 help to use **`sc_vector`**
- Allows writing more compact code

Outline

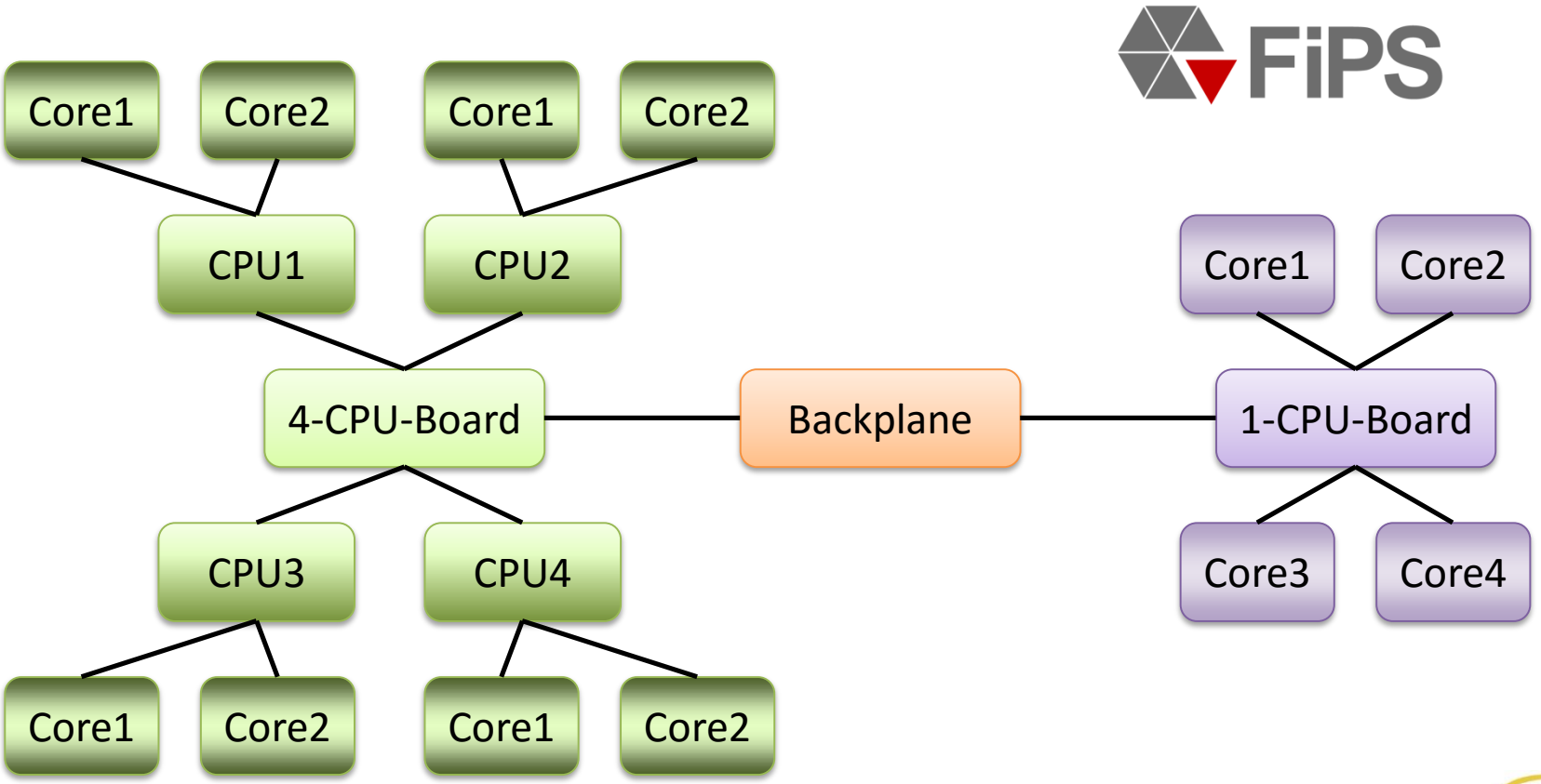
- What is **`sc_vector`**? Why do we need it?
- Use case: FiPS Platform Simulation
- Creating vector elements with anonymous functions
- Bind vectors with **`sc_assemble_vector`** and **`auto`**
- Function calls in template arguments
- Outlook: Extend **`sc_vector`** for
  - **`initializer_list`**
  - **`push_back`**
- Conclusion

sc_vector

- std::vector
  - Container for arbitrary numbers of C++ objects
  - Requires elements to be ***CopyAssignable*** and ***CopyConstructible***
  - Not fulfilled by SystemC objects
- sc_vector
  - Container for arbitrary numbers of **SystemC objects**
  - Custom element creator
  - Vector-based port binding
  - sc_assemble_vector

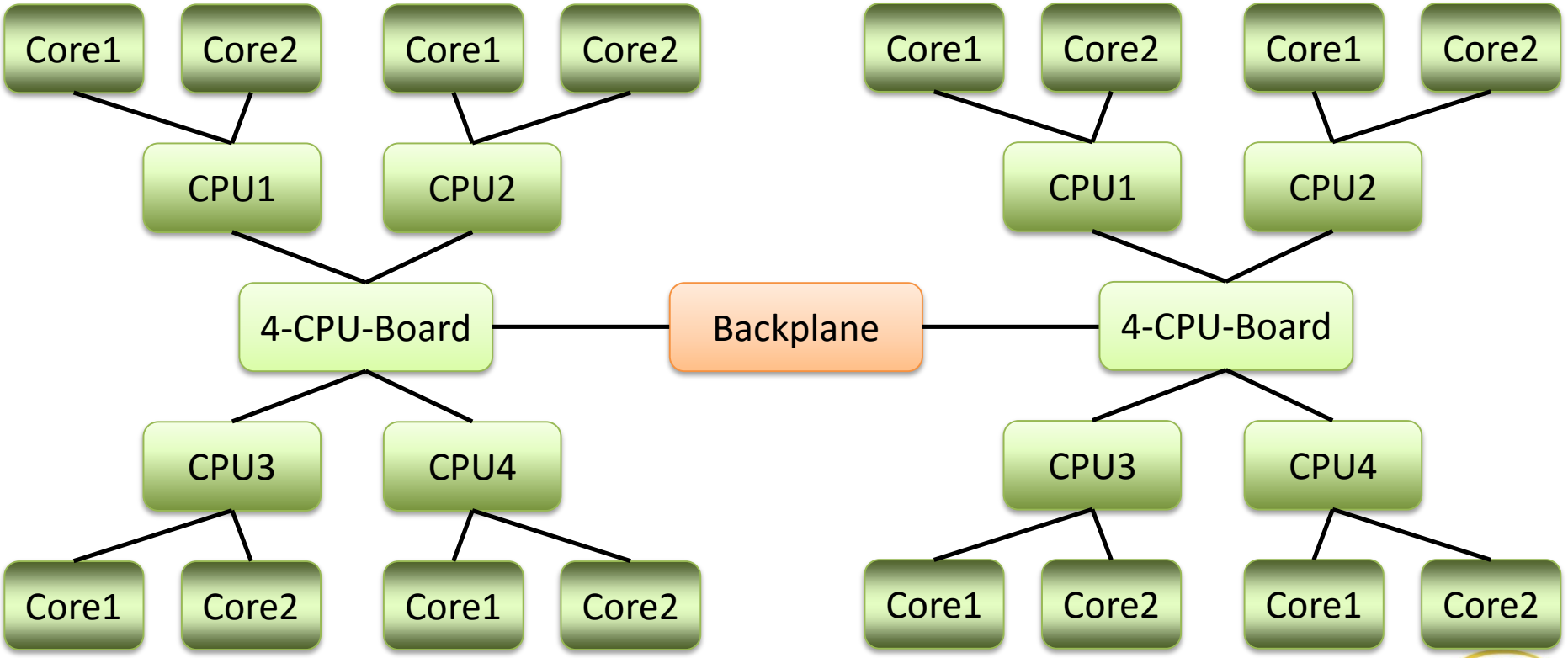
FiPS Platform Simulation

- Configurable Multiprocessor System



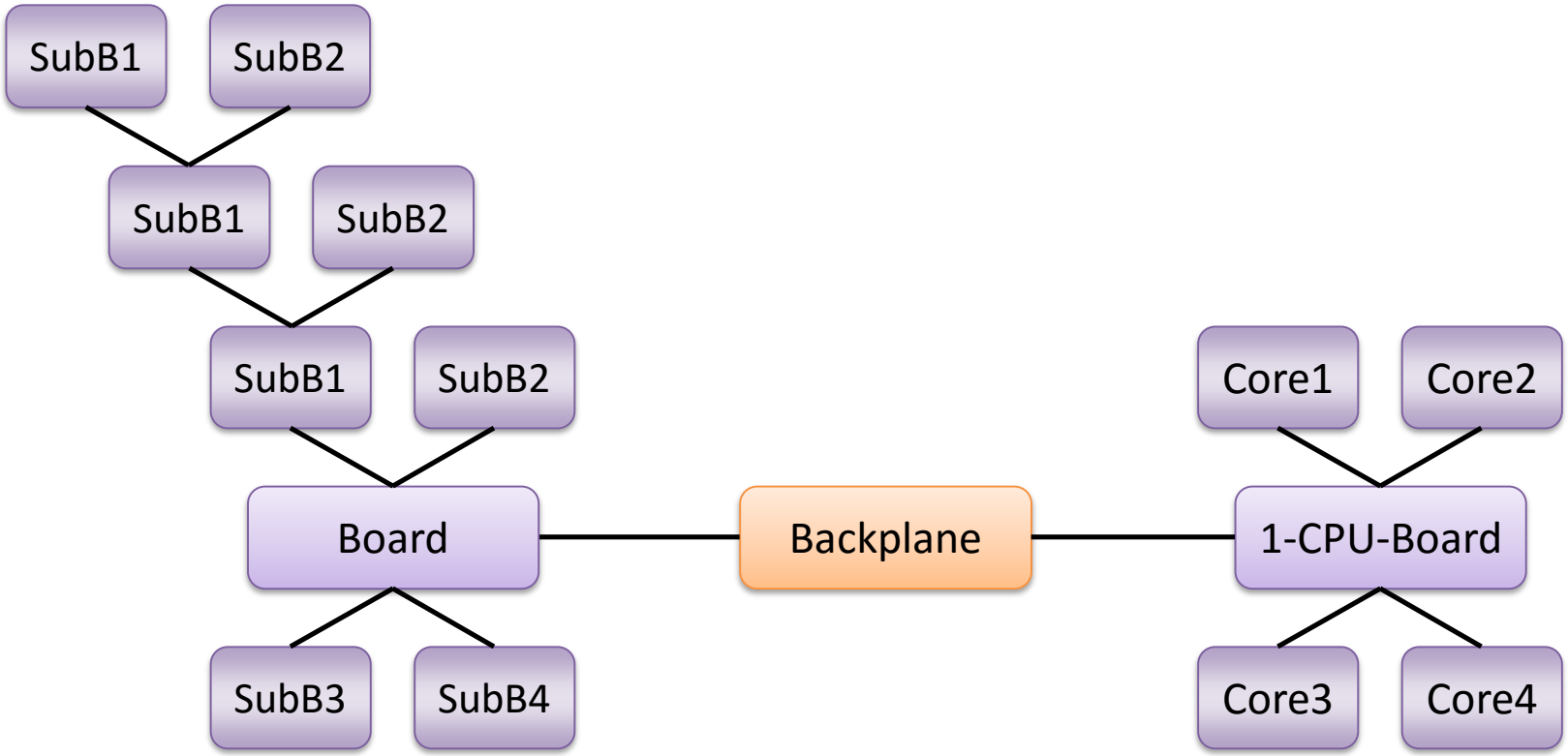
FiPS Platform Simulation

- Configurable Multiprocessor System

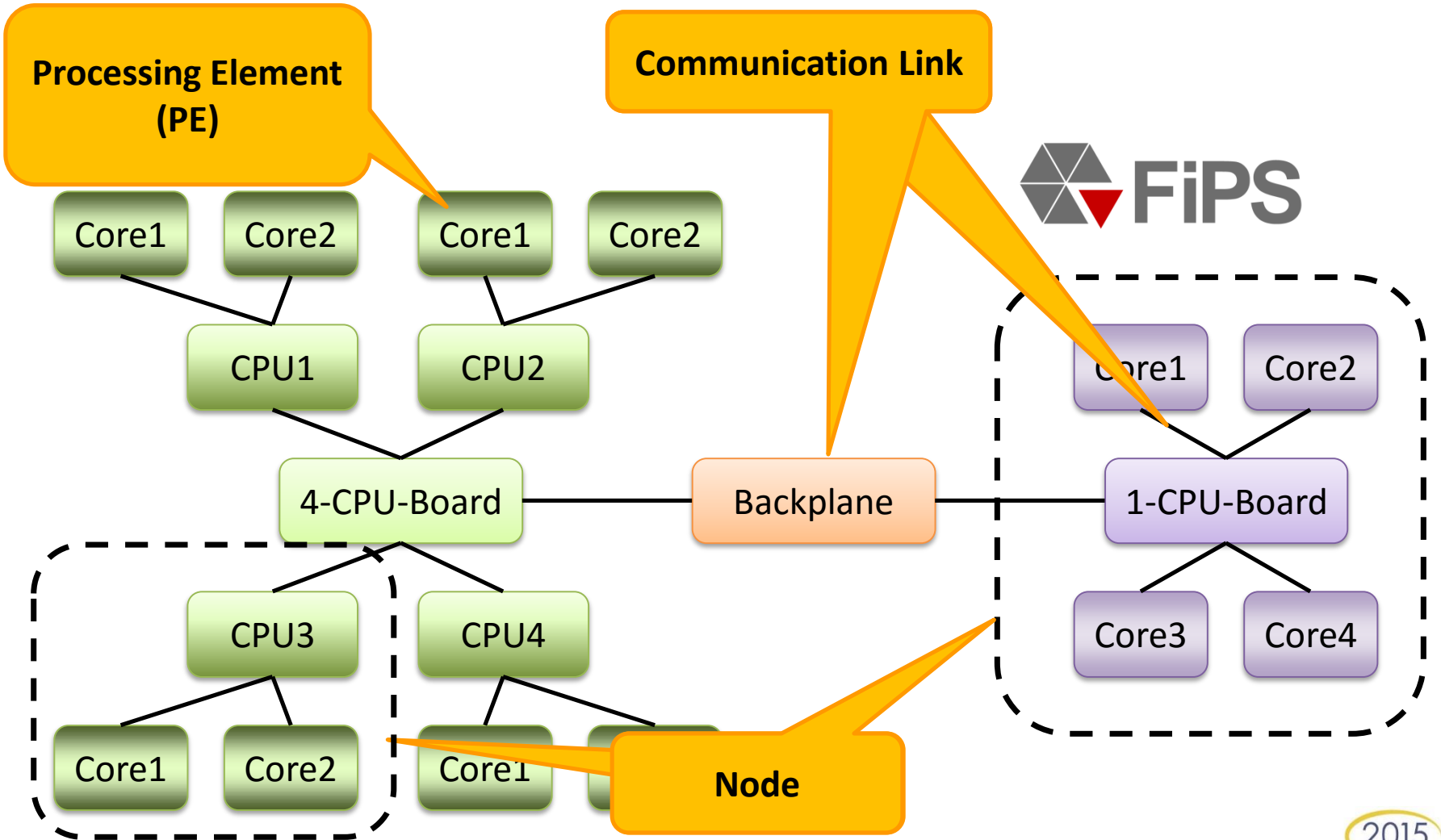


FiPS Platform Simulation

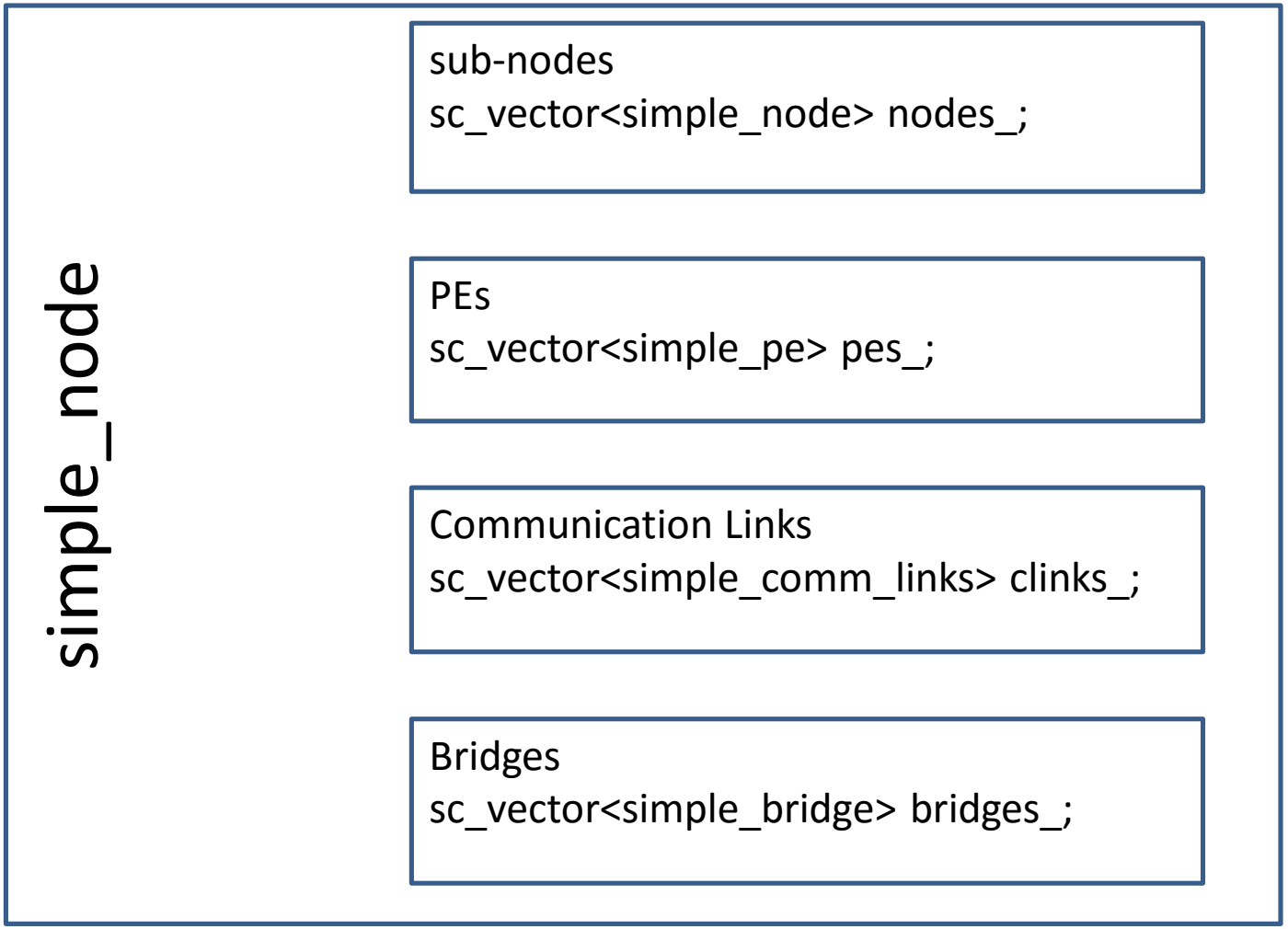
- Arbitrary Hierarchy Levels



FiPS Platform Basic Elements



simple_node


simple_node
sub-nodes
sc_vector<simple_node> nodes_;
PEs
sc_vector<simple_pe> pes_;
Communication Links
sc_vector<simple_comm_links> clinks_;
Bridges
sc_vector<simple_bridge> bridges_;

Instantiating PEs

```
simple_pe(  sc_core::sc_module_name name
          , const size_t address
          , sca_util::sca_trace_file * tf );
```

- More than one module name argument
- Custom creator required for **`sc_vector`**

```
template< typename Creator >
sc_vector( const char* , size_type , Creator );
```

- SystemC LRM:
  - Function with arguments const char * and size_type
  - Returns pointer to element object
  - Pass as function object or member function

C++11: Anonymous Function (Lambda)

- Function definition without identifier
- Used to initialize function objects

List of parameters like usual functions

Return type definition (optional)

```
[capture](parameters) ->return_type {function_body}
```

***Capture*: defines access to environment**

- **[]: nothing accessible in lambda body**
- **[&]: captures variables by reference**
- **[=]: captures variables by value**
- **[a,&b]: captures a by value,b by reference**

Function body like usual functions

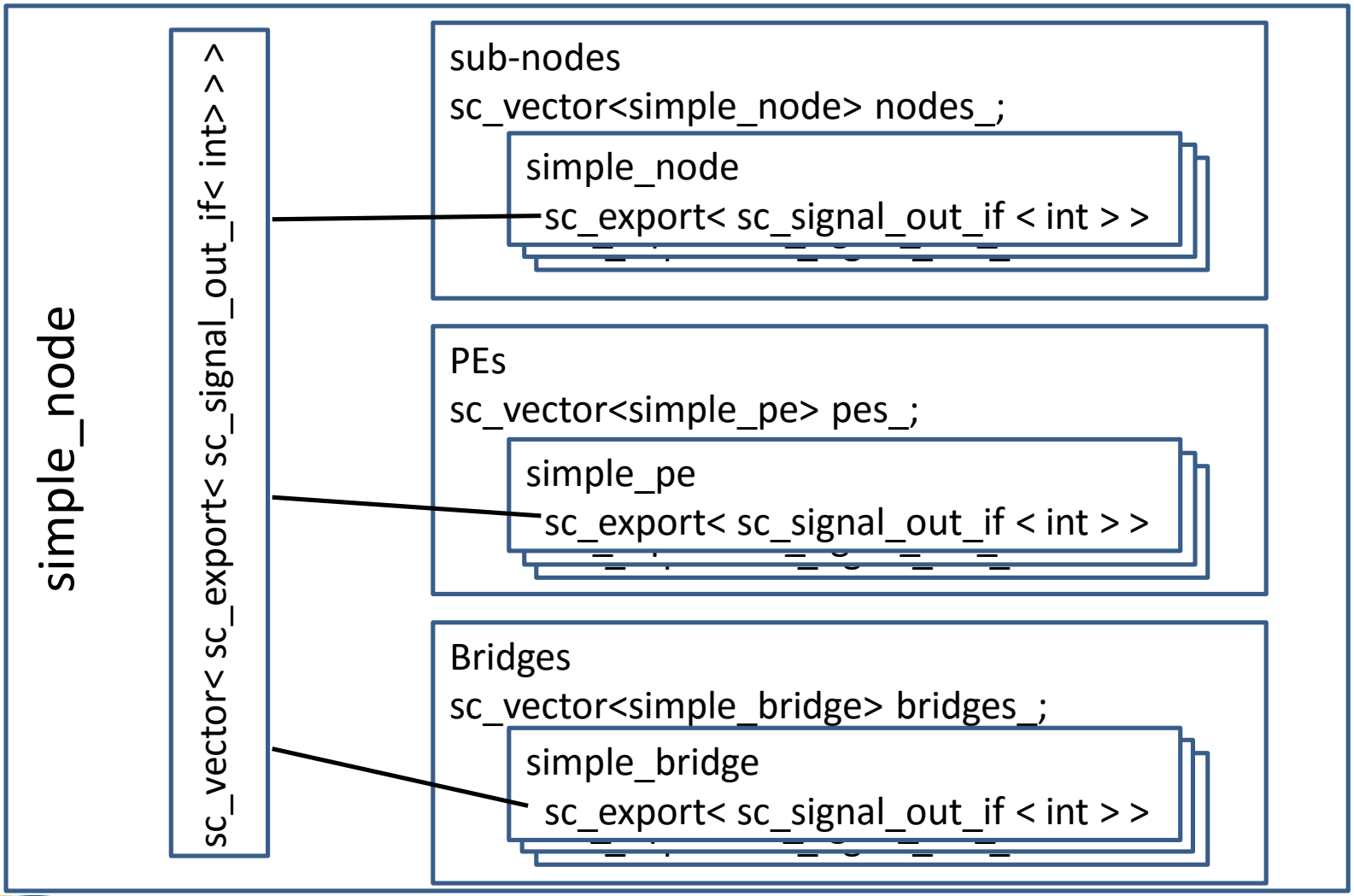
Lambda as Custom Creator

```
si_node::si_node(  sc_core::sc_module_name n
                 , size_t base_addr
                 , size_t num_pes
                 , sca_util::sca_trace_file * tf )
  : sc_module(n)
  , base_address_(base_addr)
  , num_pes_(num_pes)
  , pes_( "pe"
        , num_pes_
        , [=](const char * nm, size_t i)
          { return new simple_pe( nm
                                , base_address_ + i
                                , tf);
          })
  , //...
```

Initialize sc_vector<simple_pe>

Lambda definition as Creator

Binding with sc_assemble_vector



sc_assemble_vector

```
template< typename T , typename MT >
sc_vector_assembly< T , MT > sc_assemble_vector
          ( sc_vector< T > & , MT ( T::*member_ptr ) );
```

```
sc_assemble_vector(pes_,&simple_pe::status_);
```

- Assembles new vector from members of vector elements
- Arguments: parent vector and pointer to element member
- Return type ?

→ C++11: **`auto`** (automatic type deduction)

Automatic Type Decuction

- Keyword **`auto`** for variable declaration with immediate initialization
- Variable type is deduced from initializer

```
auto a = 1 + 2; // int
auto b = some_function() // return type of
                        // some_function()
```

```
auto va_pe =
  sc_assemble_vector(pes_,&simple_pe::status_);
```

Binding in sc_vector

```
template< typename BindableContainer >
iterator bind( BindableContainer& );

template< typename BindableIterator >
iterator bind( BindableIterator , BindableIterator );

template< typename BindableIterator >
iterator bind( BindableIterator , BindableIterator
             , iterator );
```

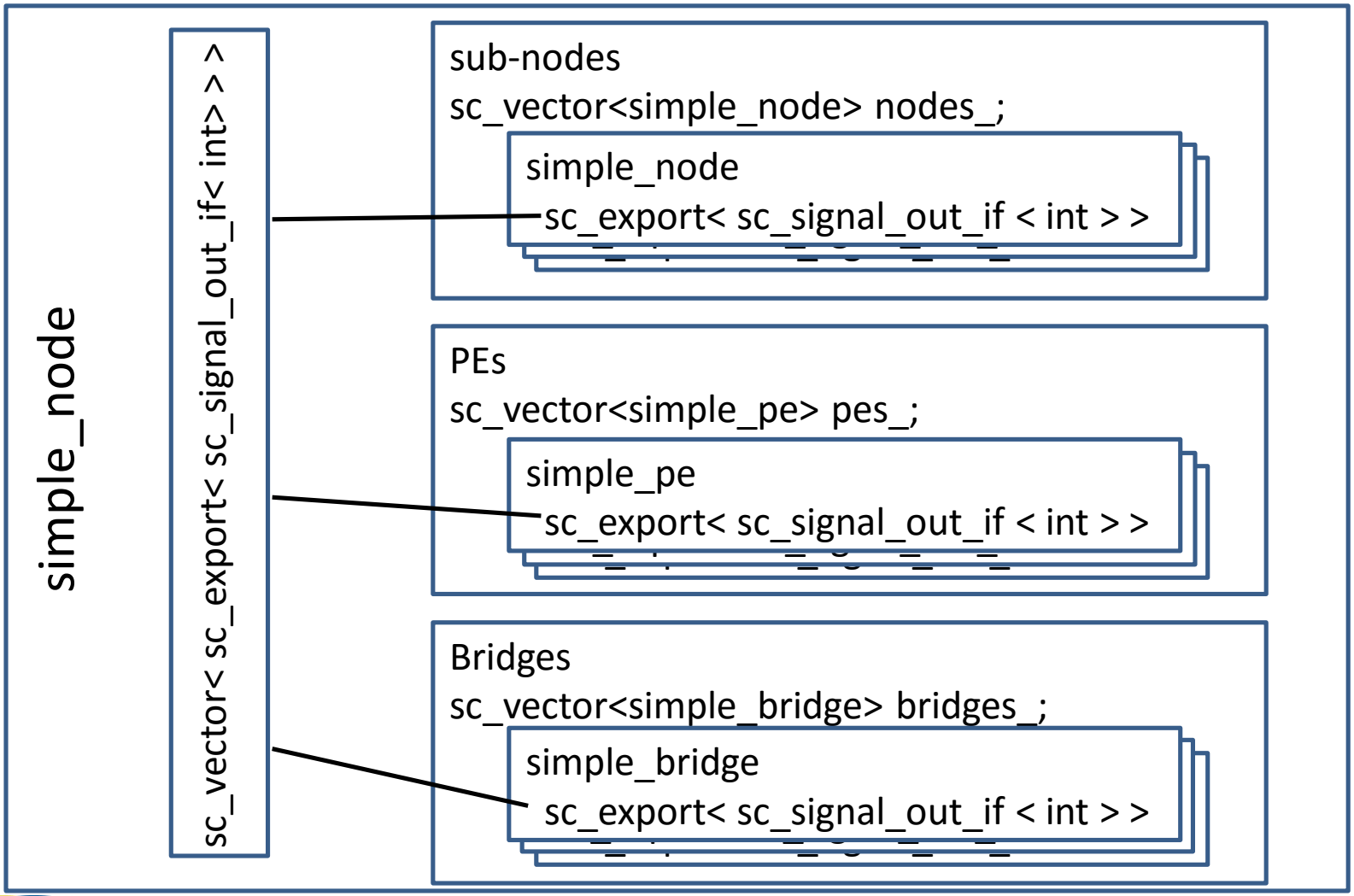
Iterator pointing to first un-bound element in this vector

Iterator pointing to first element to be bound in this vector

begin iterator

end iterator

Binding with sc_assemble_vector



Binding the Exports

```
auto va_n =
  sc_assemble_vector(nodes_,&simple_node::status_p_);
auto it_status =
  status_v_.bind(va_n.begin(), va_n.end());
auto va_pe =
  sc_assemble_vector(pes_,&simple_pe::status_p_);
it_status =
  status_v_.bind(va_pe.begin(), va_pe.end(), it_status);
auto va_br =
  sc_assemble_vector(bridges_,&simple_bridge::status_p_);
it_status =
  status_v_.bind(va_br.begin(), va_br.end(), it_status);
```

Auto for Loop Iterators

```
for ( sc_core::sc_vector<
        sc_core::sc_export <
          sc_core::sc_signal_out_if<int> > >::iterator
      it = status_v_.begin();
      it < status_v_.end();
      ++it ) {  ...
```

```
for ( auto&& it = status_v_.begin();
      it < status_v_.end();
      ++it ) {  ...
```

```
for ( auto&& it : status_v_ ) {  ...
```

Function Call in Template Argument

- Example: log2(n) as width of bit vector
  - `sc_dt::sc_bv<log2(N)> var;`
  - Function calls and loops not compile-time constant

  → Not allowed as template argument
- C++11: Relaxed Constant Expressions **`constexpr`**
  - Function calls as constant expression
    - If it contains only declarations, constant expressions, and one return
- C++14: More Relaxed Constant Expressions
  - Conditional statements, loop statements, …

Constant Log2 Function in C++11

```
constexpr int log2(int a)
{
  return (a > 0) ? 1 + log2(a >> 1) : 0;
}
```

```
static const int c1 = log2(256);
sc_dt::sc_bv<log2(c1)> var;
```

Constant Log2 Function in C++14

```
constexpr int log2_if(int a)
{
  if (a > 0)
    return 1 + log2(a >> 1);
  else
    return 0;
}
```

Constant Log2 Function in C++14

```
constexpr int log2_loop(int a)
{
  int ret = 0;
  while (a > 0)
  {
    a >>= 1;
    ++ret;
  }
  return ret;
}
```

Outlook: std::initializer_list

- Well known for C arrays

```
int my_array[] = { 1 , 2 , 3 , 42 };
```

- Now explicit in C++11 for vectors, lists, …

```
template< class T > class initializer_list;
```

Outlook: std::initializer_list

- Additional Constructor for sc_vector

```
template < typename T >
class sc_vector : //...
{
    sc_vector( std::initializer_list< T* > elements );
    // ...
```

- Initialization with list of element pointers

```
sc_core::sc_vector< my_mod_t > v =
  { new my_mod_t("ab"), new my_mod_t("cd„, 1)
  , new my_mod_t("ef", 11, 22) };
```

Outlook: push_back

- C++11: rvalue references T&&
  - Non-const reference to a temporary
  - Allows *move* instead of *copy*

```
template < typename T >
class sc_vector : //...
{
    push_back( T&& element );
//...
v.push_back(my_mod_t("ab"));
```

- Requires elements to be *MoveConstructible*
  - In theory: No problem for SystemC objects but…

  … which members must be copied and which can be moved?

  … what happens to argument? What is an 'empty' object?

  … what happens to the SystemC object hierarchy?

Conclusion

- Automated instantiation of SystemC objects very common today
- Requires configurable SystemC modules
- sc_vector and C++11/14 facilitate implementation
  - Anonymous Functions (Lambda)
  - Automatic type deduction (auto)
  - Relaxed constant expressions (constexpr)
- More compact code
- Questions?