sc_vector<T> Container and the IEEE P1666-2011 SystemC Standard

Philipp A. Hartmann
OFFIS Institute for Information Technology
R&D Division Transportation

March 15th, 2011
1 Motivation

In many SystemC designs, parametrisable **arrays of channels or ports** are used:

```cpp
sc_in<bool> intr_lines[NUM_INTERRUPTS];
std::vector<sc_signal<bool>> signals;
```

**But:** Elements of such arrays need to be **default constructible**.

- Due to a limitation of C++.
- **No custom name** can be given to ease debugging.

In case of **arrays of (sub-)modules**, mandatory `sc_module_name` parameter usually prohibits direct array usage – array of **pointers** needed.

```cpp
SC_MODULE(sub_module) { /* ... */ }
std::vector<sub_module*> sub_mod_vec;
```
2 Arrays of Pointers

Motivation

- Arrays of pointers are inconvenient to use.
  - Require **dynamic allocation/deallocation** of elements.
  - Require **additional dereference** upon access.

- Common boiler-plate code looks like:

```
SC_MODULE(sub_module) { /* ... */ };
std::vector<sub_module*> sub_mod_vec;

for(int i=0; i < n_sub; ++i) {
  // allocate sub-modules dynamically
  std::stringstream nm;
  nm << "sub_module_" << i;
  sub_mod_vec.push_back(new sub_module(nm.str().c_str()));
}
```
2 Arrays of Pointers

Motivation

- Arrays of pointers are inconvenient to use.
  - Require *dynamic allocation*/deallocation of elements.
  - Require *additional dereference* upon access.

- Common boiler-plate code looks like:

  ```
  SC_MODULE( sub_module ) { /* ... */ };
  std::vector< sub_module* > sub_mod_vec;

  for( int i=0; i < n_sub; ++i ) { // allocate sub–modules dynamically
    std::stringstream nm;
    nm << "sub_module_" << i;
    sub_mod_vec.push_back( new sub_module( nm.str().c_str() ) );
  }
  ```

- Proposed solution: sc_vector<T>

  A flexible container for named SystemC entities like modules, ports, and channels!
3 Outline

1. sc_vector<T> – A Container for Named Objects
   - Basic Interface
   - Element-wise Binding
   - Restrictions
2. Advanced Features
   - Hierarchical Binding
   - Custom Creators
3. Conclusion
4 sc_vector<T> – A Container for Named Objects

Introduction

- C++ class template for type-safe container for named SystemC objects, like
  - Modules
  - Ports
  - Channels

proposed to IEEE P1666 WG.
4 **sc_vector**<**T**> – A Container for Named Objects

**Introduction**

- C++ class template for type-safe **container for named SystemC objects**, like
  - Modules
  - Ports
  - Channels

  proposed to IEEE P1666 WG.

- Provides **std::vector**<**T**>-like **element access** but avoids previously outlined difficulties.

- **SystemC-specific API** for port/channel binding, hierarchical access and naming.

5 Basic Interface

sc_vector<T> – A Container for Named Objects

Construction, Initialisation and Destruction

- Vector elements are owned by sc_vector instance and released upon destruction of vector.
- Size can be given during vector construction or via a deferred init call.

```c++
// name/prefix == "intr_sig", size not yet specified
sc_vector< sc_signal<bool> > intr_sigs( "intr_sig" );
intr_sigs.init( NUM_INTERRUPTS );
```

Iterators and index-based element access

- random-access, constant-time `vec[i]; vec.at(i);`
- iterators following C++ standard interfaces

```c++
sc_vector<T>::iterator it = vec.begin();
(*it);
```
5 Basic Interface

sc_vector<T> – A Container for Named Objects

Construction, Initialisation and Destruction

- Vector elements are owned by sc_vector instance and released upon destruction of vector.
- Size can be given during vector construction or via a deferred init call.

```c++
// name/prefix == “intr_sig”, size not yet specified
sc_vector< sc_signal<bool> > intr_sigs( "intr_sig" );
intr_sigs.init( NUM_INTERRUPTS );
```

Iterators and index-based element access

- random-access, constant-time
  ```c++
  vec[i]; vec.at(i);
  ```
- iterators following C++ standard interfaces
  ```c++
  sc_vector<T>::iterator it = vec.begin();
  (*it);
  ```
For ports and channels, **element-wise binding** is needed.

```cpp
sc_vector<sc_in<bool> > port_vec;
sc_vector<sc_signal<bool> > sigs_vec;
for( unsigned i=0; i<port_vec.size(); ++i )
  port_vec[i].bind( sigs_vec[i] );
```

```
sc_object
  name()
  kind()

sc_vector_base
  size()
  get_elements()

T
sc_vector
  sc_vector(nm,N)
  init(N)
  ~sc_vector()
  begin()
  end()
  T& operator[](i)
  bind()
```
6 Element-wise Binding

sc_vector<T> – A Container for Named Objects

- For ports and channels, **element-wise binding** is needed.

```
sc_vector<sc_in<bool> > port_vec;
sc_vector<sc_signal<bool> > sigs_vec;
for( unsigned i=0; i<port_vec.size(); ++i )
    port_vec[i].bind( sigs_vec[i] );
```

- **sc_vector** provides family of **bind functions**

```
port_vec.bind( sigs_vec );
```

- Partial **binding of slices** supported by iterator-based variants

```
sc_vector<sc_in<bool> >::iterator last =
    port_vec.bind( sigs_vec.begin(), sigs_vec.begin()+5 );
```

- Return iterator to element **after last bound element**.
- Return values can be used as starting offset for next slice-binding.
7 Restrictions

sc_vector<T> – A Container for Named Objects

- An sc_vector<T> does **not add a level of hierarchy**.
  - Elements are children of the vector’s parent object (module, process)

- Element type T is restricted to **types derived from sc_object**.
  - Consensus in IEEE P1666 WG.
  - Enables sc_vector_base::get_elements()

- Once initialized, an sc_vector<T> **may not be resized** or extended.
  - Additionally, deferred init() may only be called from the same context (module callback, process)

- Similar to many structural SystemC elements, an sc_vector<T> **may not be copied or assigned**.
8 Hierarchical Binding

Advanced Features

- Binding of ports within elements of a vector of modules not directly possible with `sc_vector::bind`
  - ports are in separate vector elements
    ```
    SC_MODULE(sub_module){
        sc_in<bool> in_port;
    }
    sc_vector< sub_module > mod_vec;
    ```
  - not compatible with a vector of ports

Use `sc_assemble_vector` to emulate a member vector!

- `sc_assemble_vector` function creates (temporary) vector adapter based on a pointer-to-member.
  - Interface compatible to first-class `sc_vector` objects
  - Binding possible again as with plain vectors
    ```
    sc_assemble_vector(mod_vec, &sub_module::in_port).bind(...);
    ```
8 Hierarchical Binding
Advanced Features

- Binding of ports within elements of a vector of modules not directly possible with `sc_vector::bind`
  - ports are in separate vector elements
    ```
    SC_MODULE(sub_module){
        sc_in<bool> in_port;
    }
    sc_vector<sub_module> mod_vec;
    ```
  - not compatible with a vector of ports

- Use `sc_assemble_vector` to emulate a member vector!
  - `sc_assemble_vector` function creates (temporary) vector adapter based on a `pointer-to-member`.
  - **Interface compatible** to first-class `sc_vector` objects
  - Binding possible again as with plain vectors
    ```
    sc_assemble_vector(mod_vec, &sub_module::in_port)
    .bind(...);
    ```
9 Custom Creators

Advanced Features

- `sc_vector` provides template function mechanisms for custom element creators
  - To be used for additional required constructor parameters
  - Accepts function pointers or objects, callable with the signature
    ```cpp
    T* ( const char*, size_type )
    ```
9 Custom Creators
Advanced Features

- `sc_vector` provides template function mechanisms for **custom element creators**
  - To be used for additional required constructor parameters
  - Accepts function pointers or objects, callable with the signature
    
    ```
    T* ( const char*, size_type )
    ```

- Example based on plain **function pointer**

```
SC_MODULE( core ) {
    core( sc_module_name, int num_threads ); // custom constructor parameter
}

// creator function
static core* create_core( const char* nm, size_t /* idx */ ) {
    return new core( nm, NUM_THREADS );
}

sc_vector< core > core_vec;
core_vec.init( NUM_CORES, create_core );
```
9 Custom Creators
Advanced Features

- **sc_vector** provides template function mechanisms for **custom element creators**
  - To be used for additional required constructor parameters
  - Accepts function pointers or objects, callable with the signature
    
    ```
    T* ( const char*, size_type )
    ```

- Example based on member function, **sc_bind**, and placeholders _1, _2

```cpp
typedef tlm_utils::simple_target_socket_tagged<bus> socket_type;

socket_type* bus::create_socket( const char* nm, size_t idx ) {
    socket_type* socket_p = new socket_type(nm);
    socket_p->register_b_transport( this, &bus::b_transport, idx );
    return socket_p;
}
```
9 Custom Creators

Advanced Features

- `sc_vector` provides template function mechanisms for custom element creators
  - To be used for additional required constructor parameters
  - Accepts function pointers or objects, callable with the signature
    \[
    T* ( \text{const char}*, \text{size_type} )
    \]

- Example based on member function, `sc_bind`, and placeholders \_1, \_2

```cpp
typedef tlm_utils::simple_target_socket_tagged<bus> socket_type;

socket_type* bus::create_socket( const char* nm, size_t idx ) {
    socket_type* socket_p = new socket_type(nm);
    socket_p->register_b_transport( this, &bus::b_transport, idx );
    return socket_p;
}

sc_vector< socket_type > target_sockets;

target_sockets.init( NUM_INITIATORS,
                      sc_bind( &bus::create_socket, this, _1, _2 ) );
```
10 Conclusion

- Proposal of an `sc_vector<T>` container class accepted for inclusion in upcoming revision of IEEE P1666-2011 SystemC standard.¹

- Improves modelling regular structural models in SystemC by reducing boiler-plate code.

- Provides flexible support for port/interface binding and hierarchical access in addition to well-known `std::vector-like` interface.

- Standalone proof-of-concept implementation including examples available at http://system-synthesis.org/download/

¹Assuming final ratification by IEEE SA later this year.