sc_vector: A flexible container for modules, ports and channels
Contents

1 Motivation .......................... 1

2 sc_vector ............................. 2
   2.1 Description .......................... 2
   2.2 Class definition .......................... 2
   2.3 Template parameters .......................... 4
   2.4 Constraints on usage .......................... 4
   2.5 Constructors and initialisation, destructor .......................... 4
   2.6 kind, size, get_elements .......................... 6
   2.7 Element access .......................... 6
   2.8 Iterators, begin, end .......................... 7
   2.9 Binding of vectors .......................... 7

3 Looking at vector element’s members — sc_vector_assembly .......................... 8
   3.1 Class definition .......................... 8

4 Proof-of-concept implementation .......................... 9

5 TLM 2.0 — a simple router example .......................... 10

6 Further comments .......................... 12

7 Acknowledgements .......................... 12

8 Contact information .......................... 12
1 Motivation

In many designs, a parametrisable number of modules, channels, ports, or other SystemC™ objects are used. Since such SystemC objects are usually named entities in the hierarchy, it is either required (in case of modules), or at least desired to pass a name parameter to the constructor of such objects.

If one wants to create a collection of such named entities in C++/SystemC, it is usually needed to use an array/vector of pointers and to allocate the elements explicitly in a loop, since plain array values have to be created via the default constructor in C++. Access to the elements then requires an additional dereferencing operation. This is quite inconvenient and inconsistent with regular class members.

Example 1.1 Current situation

```c++
SC_MODULE( sub_module ) { /* ... */ };

SC_MODULE( module )
{
   // vector of (pointers to) sub-modules
   std::vector< sub_module* > m_sub_vec;

   // dynamically allocated C-style array of ports
   sc_core::sc_in<bool>* in_vec = new sc_core::sc_in<bool>[ n_sub ];

   module( sc_core::sc_module_name, unsigned n_sub )
   {
      // create sub-modules in a loop
      for( unsigned i=0; i<n_sub; ++i )
      {
         std::stringstream name;
         name << "sub_modules_" << i;
         m_sub_vec.push_back( new sub_module( name.str().c_str() ) );
      }

      // create (default named) array of ports
      in_vec = new sc_core::sc_in<bool>[ n_sub ];

      // bind ports of sub-modules -- requires dereference
      for( unsigned i=0; i<n_sub; ++i )
      {
         m_sub_vec[i]->in( in_vec[i] );
         // or (*m_sub_vec[i]).in( in_vec[i] );
      }
   }
   ~module()
   {
      // manual cleanup
      for( unsigned i=0; i<m_sub_vec.size(); ++i )
         delete m_sub_vec[i];
      delete [] in_vec;
   }
};
```

Frequent questions in the public SystemC discussion forums as well as experience from SystemC teaching courses have shown this to be difficult, especially for inexperienced C++/SystemC users.

This proposal aims to provide a convenience container called `sc_core::sc_vector<T>` for such objects directly within the SystemC standard to lift this burden.

With the proposed convenience class, Example 1.1 could be written as shown in the following Example 1.2. Note the avoidance of manually crafting names and the automatically handled memory management.
Example 1.2 Possible future situation

```cpp
SC_MODULE(sub_module) { /* ... */ };  

SC_MODULE(module)
{
    // vector of sub-modules
    sc_core::sc_vector<sub_module> m_sub_vec;

    // vector of ports
    sc_core::sc_vector<sc_core::sc_in<bool>> in_vec;

    module(sc_core::sc_module_name, unsigned n_sub)
    : m_sub_vec("sub_modules", n_sub) // set name prefix, and create sub-modules
      , in_vec() // use default constructor
      , in_vec("in_vec") // set name prefix
    {
        // delayed initialisation of port vector
        // here with default prefix sc_core::sc_gen_unique_name("vector")
        in_vec.init(n_sub);

        // bind ports of sub-modules -- no dereference
        for (unsigned i=0; i<n_sub; ++i)
            m_sub_vec[i].in(in_vec[i]);

        // or, with the provided vector-bind interfaces
        sc_assemble_vector(m_sub_vec, &sub_module::in).bind(in_vec);
    }
};
```

2 sc_vector

⚠️ Important
The wording in this section may not yet be formal enough for verbatim inclusion in the IEEE 1666 standard.

2.1 Description

Utility class `sc_vector` allows to create vectors of SystemC objects, that usually require a name parameter in their constructor. It provides array-like access to the members of the vector and manages the allocated resources automatically. Once the size is determined and the elements are initialised, further resizing operations are not supported. Custom constructor signatures are supported by a template `init` function, supporting user-defined element allocations.

2.2 Class definition

```cpp
namespace sc_core {

class sc_vector_base : public sc_object
{
public:
    const char * kind() const;
    size_type size() const;
    const std::vector<sc_object*>& get_elements() const;
```
sc_vector: A flexible container for modules, ports and channels

```cpp
#include <utility>

template< typename T >
class sc_vector_iter : public std::iterator< std::random_access_iterator_tag, T >
{
  // implementation-defined, but conforming to Random Access Iterator category,
  // see ISO/IEC 14882:2003(E), 24.1 [lib.iterator.requirements]
};

// member-wise view of a vector (see below)
template< typename T, typename MemberType > class sc_vector_assembly;

template< typename T >
class sc_vector : public sc_vector_base
{
public:
  typedef T element_type;
  typedef /* implementation-defined */ size_type;
  typedef sc_vector_iter< element_type > iterator;
  typedef sc_vector_iter< const element_type > const_iterator;

  sc_vector();
  explicit sc_vector( const char* prefix );
  sc_vector( const char* prefix, size_type n );

  template< typename Creator >
  sc_vector( const char* prefix, size_type n, Creator c );

  virtual ~sc_vector();

  void init( size_type n );

  template< typename Creator >
  void init( size_type n, Creator c );

  static element_type* create_element( const char* prefix, size_type index );

  // ------------------ element access ---------------------
  element_type& operator[]( size_type i );
  element_type& at( size_type i );

  const element_type& operator[]( size_type i ) const;
  const element_type& at( size_type i ) const;

  // ----------------- iterator access ---------------------
  iterator begin();
  iterator end();

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

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

  // ---------------- binding interface ---------------------
  template< typename ContainerType, typename ArgumentType >
  iterator bind( sc_vector_assembly<ContainerType,ArgumentType> c );

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

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

    template< typename BindableIterator >
    iterator bind( BindableIterator first, BindableIterator last, iterator from );

    template< typename ContainerType, typename ArgumentType >
    iterator operator()( sc_vector_assembly<ContainerType,ArgumentType> c );

    template< typename ArgumentContainer >
    iterator operator()( ArgumentContainer & c );

    template< typename ArgumentIterator >
    iterator operator()( ArgumentIterator first, ArgumentIterator last );

private:
    // disabled
    sc_vector( const sc_vector& );
    sc_vector& operator=( const sc_vector& );
};

2.3 Template parameters

The template parameter passed as an argument \( T \) to \( \text{sc\_vector} \) shall be derived from the class \( \text{sc\_object} \). If not used with a custom \text{Creator} functor, \( T \) shall provide a constructor with an argument of a type convertible from \text{const char*}.

NOTE — In case of plain C++ types, the use of standard C++ containers like \text{std::vector} is recommended.

NOTE — For the constraints on the arguments to the template parameter \text{Creator}, passed to the templated constructor and \text{init} function, see Section 2.5.

2.4 Constraints on usage

An implementation shall derive class \( \text{sc\_vector\_base} \) from class \( \text{sc\_object} \). This implementation-defined base class provides type-agnostic access to the \text{sc\_object} elements contained in the vector.

Objects of class \( \text{sc\_vector} \) can only be constructed during elaboration, if the element type can only be instantiated during elaboration (like modules, ports, channels). Otherwise, a vector can be instantiated during simulation as well.

A vector shall not introduce an additional level in the object hierarchy. If the vector contains instances of classes derived from \text{sc\_object}, such objects shall be siblings of the vector instance itself.

2.5 Constructors and initialisation, destructor

\text{sc\_vector}();

explicit \text{sc\_vector}( \text{const char*} prefix );

\text{sc\_vector}( \text{const char*} prefix, \text{size\_type} n );

The constructors for class \( \text{sc\_vector} \) shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class \( \text{sc\_object} \) to set the string name of the vector instance in the module hierarchy.
The default constructor shall call function `sc_gen_unique_name( "vector")` to generate a unique string name that it shall then pass through to the constructor for the base class `sc_object`.

An optional argument `n` of the unsigned integral type `size_type` shall be used to determine whether a default initialisation with `n` objects of type `element_type` shall be performed. If the value of `n` is zero, no such initialisation shall occur, otherwise the function `init` shall be called argument `n` within the constructor.

```cpp
void init( size_type n );
static void create_element( const char* name, size_type index );
```

Calling the `init` function shall fill the vector with `n` instances of `element_type`, that are allocated by the static function `create_element`. It shall be an error to call the `init` function more than once on the same vector, or to call it after the elaboration has finished.

The `create_element` function is the default allocation function for vector elements. Within this function, an instance of `element_type` shall be allocated via `operator new` and the argument `name` shall be passed to the constructor.

The `name` argument shall be constructed within the `init` function by appending the vector's `basename` with the current index, separated by an underscore.

It shall be an error to call the `init` from another object context, than the vector's context itself. As a result, the parent objects of the vector and all of its elements are the same at all times.

**NOTE** — The `init` function can be used to decouple the element creation from the construction of the vector instance itself, e.g. during `before_end_of_elaboration`.

```cpp
template< typename Creator >
sc_vector< const char* prefix, size_type n, Creator c >;
template< typename Creator >
void init( size_type n, Creator c );
```

The templated versions of the constructor and the `init` function shall use a value of type `Creator` that has been passed as argument for the allocation of the element instances. Instead of calling `create_element` with arguments `name` and `index` to allocate each element instance of the vector, the expression

```cpp
element_type * next = c( name, index );
```

shall be well-formed for an argument `c` of template parameter type `Creator`.

The expressions `v.init(n, sc_vector<T>::create_element)` and `v.init(n)` shall be equivalent for all vectors `sc_vector<T> v`.

**NOTE** — A frequent use case for custom `Creator` arguments are additional, required constructor parameters of the contained `element_type`. `Creator` can then either be a function pointer or a function object (providing an appropriate `operator()( const char*, size_t )`).

**Example**

```cpp
SC_MODULE( sub_module )
{
  // constructor with additional parameters
  sub_module( sc_core::sc_module_name, int param );
  // ...
};

SC_MODULE( module )
{
  sc_core::sc_vector< sub_module > sub_vec1, sub_vec2; // vector of sub-modules
  struct mod_creator // Creator struct
  {
```
mod_creator( int p ) : param(p) {} // store parameter to forward

sub_module* operator()( const char* name, size_t ) // actual creator function
{
    return new sub_module( name, param ); // forward param to sub-module
}

// creation via member function
sub_module* another_creator( const char* name, size_t id )
{
    return new sub_module( name, id );
}

module( sc_core::sc_module_name, unsigned n_sub )
: sub_vec1( "sub_modules" ) // set name prefix
{
    sub_vec1.init( n_sub, mod_creator(42) ); // init with custom creator
    // or init via sc_bind and local member function
    sub_vec2.init( n_sub, sc_bind( &module::another_creator
    , this, sc_unamed::_1, 42 ) );
}

virtual ~sc_vector();

During destruction of the vector, all elements held by the vector shall be destroyed by calling operator delete on their addresses.

2.6 kind, size, get_elements

const char * kind() const;
size_type size() const;
const std::vector<sc_object*>& get_elements() const;

Member function kind shall return the string "sc_vector".
Member function size shall return the number of initialised objects of the vector.
Member function get_elements shall return a const-reference to a std::vector containing pointers to the contained elements. If the sc_vector object has not yet been initialised, the returned std::vector shall be empty. The returned reference shall be valid for the lifetime of the sc_vector object.
NOTE — The relation v.get_elements().size() == v.size() holds at all times.

2.7 Element access

element_type& operator[]( size_type i );
element_type at( size_type i );

const element_type& operator[]( size_type i ) const;
const element_type& at( size_type i ) const;

operator[] and member function at shall return a (const qualified) reference to the object stored at index i.
If the given index argument exceeds the size of the vector, the behaviour is undefined in case of operator[]. In case of member function at, the implementation shall detect invalid indices as an error.
It is undefined, whether the relation &v[i]+j == &v[i+j] holds for any vector v and indices i, j.
References returned by functions defined in this clause shall be valid until the lifetime of the surrounding vector has ended.
2.8 Iterators, begin, end

For compatibility with the C++ Standard Library, `sc_vector` shall provide an iterator interface, that fulfills the Random Access Iterator requirements as defined in ISO/IEC 14882:2003(E), Clause 24.1 [lib.iterator.requirements].

```cpp
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
const_iterator cbegin(); const
const_iterator cend(); const
```

`begin` returns an iterator referring to the first element in the vector. `end` returns an iterator which is the past-the-end value for the vector. If the vector is empty, then `begin() == end()`.

Once the initialisation of the vector has been performed (Section 2.5) or the simulation has started, iterators returned by `begin` or `end` shall be valid until the lifetime of the referenced vector (element) has ended. The variants `cbegin` and `cend` shall return the same `const_iterator` values as `begin` and `end`, respectively.

2.9 Binding of vectors

Since `sc_vector` is mainly intended for structural components of a model, direct support for the various `bind` combinations is useful. The vector class provides the following combinations for element-wise binding between different containers, either based on full container ranges, or sub-ranges given as explicit iterators. All different overloads return an iterator to the first unbound element within the `sc_vector` object or `end()` if all elements have been bound.

```cpp
template< typename ContainerType, typename ArgumentType >
iterator bind( sc_vector_assembly<ContainerType,ArgumentType> c )
{ return bind( c.begin(), c.end() ); }

template< typename BindableContainer >
iterator bind( BindableContainer & c )
{ return bind( c.begin(), c.end() ); }

template< typename BindableIterator >
iterator bind( BindableIterator first, BindableIterator last )
{ return bind( first, last, this->begin() ); }

template< typename BindableIterator >
iterator bind( BindableIterator first, BindableIterator last, iterator from );

template< typename ContainerType, typename ArgumentType >
iterator operator() ( sc_vector_assembly<ContainerType,ArgumentType> c )
{ return operator() ( c.begin(), c.end() ); }

template< typename ArgumentContainer >
iterator operator() ( ArgumentContainer & c )
{ return operator() ( c.begin(), c.end() ); }

template< typename ArgumentIterator >
iterator operator() ( ArgumentIterator first, ArgumentIterator last )
{ return operator() ( first, last, this->begin() ); }

template< typename ArgumentIterator >
iterator operator() ( ArgumentIterator first, ArgumentIterator last, iterator from );
```

The three-argument variants take a range of argument iterators, that shall be bound element-wise — either via a `bind` or an `operator()` call — to the `sc_vector` range starting from position `from`. For that matter, the expressions
shall be well-formed for the actual template parameters. If from does not point to an element in the current sc_vector object, the behaviour shall be undefined.

NOTE — Since sc_vector is a class template, only those member functions are instantiated, that are actually used by an application. The addition of bind functions to the overall template does not restrict the general usability for types without their own bind functions.

3 Looking at vector element’s members — sc_vector_assembly

Especially during the hierarchical binding of port or module containers, element-wise access to a vector element’s member is needed. To avoid another cause for manually looping over container elements, a member-wise view with the same public interface as an sc_vector itself is provided.

Example

```c++
SC_MODULE( sub_module ) {
    sc_in<bool> in; /* ... */
};
sc_vector<sc_in<bool> > in;
sc_vector<sub_module> sub_mods;
// ...  
// for( int i=0; i<in.size(); ++i )
//     sub_mods[i].in( in[i] );
// better: bind member ports over a vector view:
sc_assemble_vector( sub_mods, &sub_module::in ).bind( in );
```

The actual sc_vector_assembly<T,MT> class is implementation-defined and shall not be created explicitly by an application. An application shall use the light-weight factory function sc_assemble_vector instead.

NOTE — sc_vector_assembly<T,MT> can be assigned and copied, though.

3.1 Class definition

```c++
template< typename T, typename MT >
class sc_vector_assembly
{
public:
    typedef sc_vector<T> base_type;
    typedef /* implementation-defined */ iterator;
    typedef /* implementation-defined */ const_iterator;

typedef typename base_type::size_type size_type;
typedef typename base_type::difference_type difference_type;
typedef typename iterator::reference reference;
typedef typename iterator::pointer pointer;

iterator begin();
```
iterator end();
const_iterator begin() const;
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

size_type size() const;

const std::vector< sc_object* > & get_elements() const;

reference operator[]( size_type idx );
reference at( size_type idx );
const reference operator[]( size_type idx ) const;
const reference at( size_type idx ) const;

template< typename ContainerType, typename ArgumentType >
iterator bind( sc_vector_assembly<ContainerType,ArgumentType> c );

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

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

template< typename BindableIterator >
iterator bind( BindableIterator first, BindableIterator last, iterator from );

template< typename ContainerType, typename ArgumentType >
iterator operator()( sc_vector_assembly<ContainerType,ArgumentType> c );

template< typename ArgumentContainer >
iterator operator()( ArgumentContainer & c );

template< typename ArgumentIterator >
iterator operator()( ArgumentIterator first, ArgumentIterator last );

template< typename ArgumentIterator >
iterator operator()( ArgumentIterator first, ArgumentIterator last, iterator from );

All member functions (except get_elements) shall forward their arguments to the underlying sc_vector instance. Iterator dereference, element access and bind shall operate on the element’s member, given as a pointer-to-member argument to sc_assemble_vector.

The member function get_elements of the view proxy class shall return a std::vector< sc_object* > by-value. This vector shall be formed by casting the member-pointees statically to sc_object.

NOTE — Since the get_elements function requires a static_cast from the member-pointer to an sc_object pointer, calling get_elements is only supported for types that are publicly derived from sc_object.

NOTE — Since sc_assemble_vector enables traversal over members of an sc_vector, the result of get_elements has to be dynamically created.

4 Proof-of-concept implementation

A proof of concept implementation based on a thin, type-safe wrapper around std::vector<void*> has been sent to the OSCI SystemC Language Working Group and can be made available to the IEEE P1666 Working Group under the terms of the OSCI Open Source License upon request.
The proposed API documented in this article has evolved during the discussion in the OSCI Language Working Group and the IEEE P1666 Working Group. The current version is based on the received feedback and internal polishing compared to the earlier proposals to the LWG and P1666. The most important changes are:

**THIRD REVISION (2010-11-29)**

- Following the consensus in P1666, renamed `sc_view` to `sc_assemble_vector`

**SECOND REVISION (2010-11-04)**

- Restrict template arguments to objects derived from `sc_object`
- Add `get_elements` to `sc_vector_base` and `sc_vector_view`
- Standardise `sc_vector_base` for easier traversal.
- Allow creation during simulation, if element type does.

**FIRST REVISION**

- Class names now `sc_vector`, instead of `sc_array`.
- Internally combined iterator implementation for `sc_vector_iter` and iteration of a member view.
- Default constructor added to `sc_vector<T>`.
- Added `bind`, `sc_view`, and `sc_vector_view`.
- Other reductions of explicitly exposed symbols.

## 5 TLM 2.0 — a simple router example

Sockets from TLM 2.0 are already usable with the proposed `sc_vector` API. The following is based on a Doulos example of a simple router:

```cpp
template<unsigned int N_TARGETS> // could be a member instead
struct Router: sc_module
{
  // TLM-2 socket, defaults to 32-bits wide, base protocol
tlm_utils::simple_target_socket<Router> target_socket;

  // use tagged sockets to be able to distinguish incoming backward path calls
  sc_vector<tlm_utils::simple_initiator_socket_tagged<Router> > initiator_socket;

  SC_CTOR(Router)
  :
    target_socket("target_socket")
    , initiator_socket("socket") // set name prefix
  {
    // Register callbacks for incoming interface method calls
    target_socket.register_b_transport( this, &Router::b_transport);
    target_socket.register_get_direct_mem_ptr( this, &Router::get_direct_mem_ptr);
    target_socket.register_transport_dbg( this, &Router::transport_dbg);
  }
```
// create tagged sockets from a member function

initiator_socket.init( N_TARGETS,
                     sc_bind(&Router::create_socket,
                             this, sc_unamed::_1, sc_unamed::_2 ) );

// or use an explicit loop
initiator_socket.init( N_TARGETS );
for( int i=0; i<N_TARGETS; ++i )
    initiator_socket[i].register_invalidate_direct_mem_ptr( ... , i ) ;
}

// socket creation function
tlm_utils::simple_initiator_socket_tagged<Router> *
create_socket( const char * name, size_t id )
{
    tlm_utils::simple_initiator_socket_tagged<Router> * socket
        = new tlm_utils::simple_initiator_socket_tagged<Router>( name );

    // Register callbacks for incoming interface method calls, including tags
    socket->register_invalidate_direct_mem_ptr
        ( this, &Router::invalidate_direct_mem_ptr, id );

    return socket;
}

// FORWARD PATH
// ***************

// TLM-2 blocking transport method
virtual void b_transport( tlm::tlm_generic_payload & trans, sc_time & delay )
{
    // ...
    // Forward transaction to appropriate target
    initiator_socket[target_nr]->b_transport( trans, delay );
    // instead of ugly:
    // ( *initiator_socket[target_nr] )->b_transport( trans, delay );
}

// TLM-2 forward DMI method
virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload & trans,
                                tlm::tlm_dmi & dmi_data);
{
    //...
    bool status = initiator_socket[target_nr]->get_direct_mem_ptr( trans, dmi_data );

    // Calculate DMI address of target in system address space
    // ...
    return status;
}

// TLM-2 debug transaction method
virtual unsigned int transport_dbg(tlm::tlm_generic_payload & trans)
{
    //...
    // Forward debug transaction to appropriate target

return initiator_socket[target_nr]->transport_dbg( trans );

// ******************************************************
// BACKWARD PATH, & ROUTER INTERNALS unchanged
// ******************************************************

6 Further comments

`sc_vector< T >` can easily support polymorphism

In the following example, a different type derived from a common base class (which is then used as `element_type` in the vector) is allocated within a user-defined factory function.

```c++
#include <systemc>

struct derived : sc_core::sc_object
{
    derived( const char * n ) : sc_object( n ) {}  
    virtual const char* kind() const { return "derived"; }
};

sc_core::sc_object* create_derived( const char* name, size_t n )
{ return new derived(name); }

int sc_main( int, char*[])
{    
    sc_core::sc_vector< sc_core::sc_object > v( "v", 1, create_derived );
    std::cout << v[0].name() << " " << v[0].kind();
    
    // Output: v_0 derived
}
```

7 Acknowledgements

The author would like to thank John Aynsley, David C. Black, Dennis P. O’Connor, and Jerome Cornet for their valuable feedback. Thanks to Ralph Görgen and Andreas Herrholz for their proof-reading and detailed comments.

8 Contact information

Philipp A. Hartmann <philipp.hartmann@offis.de>
Hardware/Software Design Methodology Group
R&D Division Transportation
OFFIS Institute for Information Technology
Oldenburg, Germany