sc_vector: A flexible container for modules, ports and channels

Contents

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 at least a name parameter to the constructor of such objects.

If one wants to create an array of such named entities in C++/SystemC, it is usually needed to use an array/vector of pointers and to allocate the elements in a loop explicitly, since direct array values can only be created via the default constructor. 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

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

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

  // dynamically allocated array of ports
  sc_core::sc_in<bool>*      in_vec;

  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 has 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.

Important
All class and function names are of course only initial proposals and not meant to be fixed, yet. Additionally, the name `sc_vector` might be too close to the (deprecated) `sc_pvector` class in the OSCI implementation. An alternative naming could be based on an `sc_array...` prefix.

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

Example 1.2 Possible future situation

```
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] );
  }
};
```

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 allocation implementations.

2.2 Class definition

```
namespace sc_core {

class sc_vector_base : public sc_core::sc_object { /* implementation defined */ };

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]
};

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 );

  virtual const char * kind() const;
          size_type    size() const;

  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 begin();
  iterator end();

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

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

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

2.3 Template parameters

There are no restrictions on the properties of the template parameter passed as an argument to `sc_vector`. In most use cases, the argument should be derived from the class `sc_object`, or provide a constructor with an argument of a type convertible from `const char*`.

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

NOTE — For the constraints of the arguments to the template parameter `Creator`, passed to the templated constructor and `init` function, see Section 2.5.

2.4 Constraints on usage

An implementation shall derive class `sc_vector_base` from class `sc_object`.

Objects of class `sc_vector` can only be constructed during elaboration. It shall be an error to instantiate a vector during simulation.

A vector shall not introduce an additional level in the object hierarchy. If the vector contains instances of classes derived from `sc_object`, such objects shall be siblings of the vector instance itself.

2.5 Constructors and initialisation, destructor

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

The constructors for class `sc_vector` shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class `sc_object` to set the string name of the 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.

```
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 calling `sc_gen_unique_name( basename() )` once for each call to `create_element`.

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`.

```
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

```
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 addtional, required constructor parameters of the contained `element_type`. `Creator` can then either be a function pointer or a function object (providing an `operator()`).

Example

```
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 > m_sub_vec;          // 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
    }
    int param;
  };

  module( sc_core::sc_module_name, unsigned n_sub )
    : m_sub_vec( "sub_modules" )                       // set name prefix
  {
    m_sub_vec.init( n_sub, mod_creator(42) );          // init with custom creator
    // ...
  }
};
```

```
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

Member function `kind` shall return the string *"sc_vector"*.

Member function `size` shall return the number of initialised objects of the vector.

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 fulfils the *Random Access Iterator* requirements as defined in ISO/IEC 14882:2003(E), Clause 24.1 [lib.iterator.requirements].

```
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.

3 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 been changed slightly based on received feedback and internal polishing compared to the initial proposal to the LWG. The most important changes are:

- Class names now `sc_vector`, instead of `sc_array`.
- Second template parameter stripped from `sc_vector_iter`.
- Default constructor added to `sc_vector<T>`.
- Other reductions of explicitly exposed symbols.

4 Further comments

`sc_port_vector< IF, N=1, POL=sc_core::SC_ONE_OR_MORE_BOUND >`

For even further convenience, a special implementation for ports can easily be added as well. Essentially, the implementation of such a wrapper can be reduced to something like:

```
template< typename IF, N=1, POL=sc_core::SC_ONE_OR_MORE_BOUND >
class sc_port_vector
  : public sc_vector< sc_port< IF, N, POL >
{
  typedef sc_vector< sc_port< IF, N, POL > base_type;
public:
  sc_port_vector() : base_type( sc_core::sc_gen_unique_name("port_vector") ) {}
  explicit sc_port_vector( const char* name, size_type n=0 )
    : base_type( name, n )  {}
  virtual const char* kind() const;
};
```

Whether additional specialisations for the full set of predefined ports (and maybe even the channels) are worth the effort is to be decided.

`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.

```
#include <systemc>

struct derived : sc_core::sc_object
{
  derived( const char * n ) : sc_object( n ) {}
  virtual const char* kind() { 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
```

5 Acknowledgements

The author would like to thank Dennis P. O'Connor and David C. Black for their valuable feedback on the first submission of this proposed SystemC extension to the OSCI SystemC Language Working Group. Thanks to Ralph Görgen and Andreas Herrholz for their proof-reading and detailed comments.

6 Contact information

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