2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Automated SystemC Model Instantiation with modern C++ Features and sc_vector[1]

Ralph Görgen, OFFIS, Oldenburg, Germany (ralph.goergen@offis.de)

Philipp A. Hartmann, Intel, Duisburg, Germany (philipp.a.hartmann@intel.com)

Wolfgang Nebel, CvO University Oldenburg, Germany (nebel@informatik.uni-oldenburg.de)

I. INTRODUCTION

SystemC-based models are often instantiated automatically, at least parts of it. They mainly consist of configurable components taken form some catalogue or library and a description of their configuration and connections. The instantiation of the actual SystemC objects is done automatically according to the description. In addition, automatic instantiation of SystemC components occurs inside configurable components, for instance in components with a configurable number of I/O ports, bus connections, or arbitrary signal dimensions.

This contribution describes how modern C++ features introduced in C++11 and C++14 can facilitate the implementation of such automatisms in combination with `sc_vector`. We show how anonymous functions (lambda functions), type inference (`auto`), and constant expressions (`constexpr`) can be used in the context of SystemC to make the designers life easier. We illustrate the practical application with code examples of a configurable platform consisting of an arbitrary hierarchy of computational elements and communication links. In addition, the possibility for supporting initializer lists and `push_back` in `sc_vector` is discussed.

The paper is organized as follows: First, we will to introduce `sc_vector` and our platform model shortly in Section II and Section III. Then, we explain the creation of vector elements with anonymous functions in Section IV, vector based binding and the benefits of `auto` in Section V, and the application of relaxed constant expressions in Section VI. Finally, the discussion of initializer lists and `push_back` in `sc_vector` is given in Section VII, and Section VIII concludes the paper.

II. SC_VECTOR

To enable the configurability in our platform model, we use `sc_vector`. `sc_vector` is a container class for arbitrary numbers of SystemC objects. Unlike `std::vector`, it does not require its content objects to be *Assignable* and *CopyConstructible*, requirements that cannot be fulfilled by SystemC modules or ports. In addition, `sc_vector` supports specific features for SystemC: The definition of a custom creator to call module constructors with more arguments than the default name argument and port binding for entire vectors instead of element per element.

Another feature is the function `sc_assemble_vector`. It creates vectors of members of vector elements. E.g., if we have a vector containing modules and each of these modules has an input port called *in_1*, we can assemble the *in_1* ports of all modules to a new vector. This is especially helpful for vector based port binding.

A complete description of `sc_vector` and `sc_assemble_vector` can be found in Section 8.5 of the SystemC LRM [2].

III. USE CASE

The FiPS platform simulation will serve as an example here [1]. The platform represents a heterogeneous multi-processor system and consists of a configurable number of *processing elements* (PE) that are clustered to hierarchical *nodes*. The processing elements are connected by a hierarchical and heterogeneous communication network.

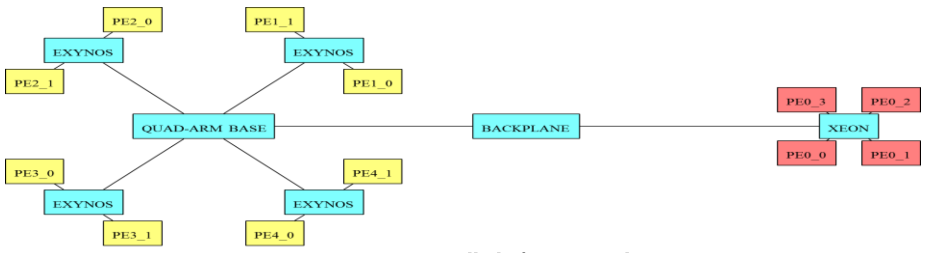

Figure 1: Small Platform Example

[1] This work has been partially supported by the EU projects FiPS (FP7-609757) and CONTREX (FP7-611164).

Figure 1 shows an example for a small platform. On the left side, there are four nodes containing two PEs each and these four nodes are again clustered to a node. Another node containing four PEs is shown on the right side. In general, a node can contain

- a configurable number of sub-nodes,
- a configurable number of PEs,
- a node local communication bus, and
- a configurable number of communication bridges to connect local busses to other hierarchy levels.

The models for different kinds of PEs, busses, and bridges are available in a library that contains configurable models for all these elements. Configuration, architecture, and communication setup of the entire system are described in an XML file. Based on this description, the instantiation of the platform model is done fully automatic.

In the implementation, all of the platform components are based on four configurable SystemC module types, `simple_node`, `simple_pe`, `simple_bridge`, and `simple_comm_link`. More components derived from these basic modules and providing a more specific behavior are available in the library as well, but we will skip details about them because they play no role for this paper. Two of these simple module types make use of `sc_vector`. `simple_comm_link` and `simple_node`. The first contains a number of fifos and provides their interface as exports. Since the number of fifos and exports is configurable, they are implemented as vectors. A `simple_node` contains `sc_vectors` for its sub-nodes (of type `simple_node`), PEs (of type `simple_pe`), and bridges (of type `simple_bridge`). Furthermore, the `simple_node` contains a vector of exports of type `sc_signal_out_if`. Each component provides some status information in form of an export to its parent. The vector of exports in `simple_node` is used to aggregate these exports and provide them to the next hierarchy level for further analysis. In the following, we will describe how modern C++ features can be used to handle these vectors.

IV. Creation of Vector Elements with Anonymous Functions

The first aspect of `sc_vector` we want to look at is filling it with elements. This can be done by passing the number of required elements to the constructor or by calling member function `init`[2]. In the following examples, we use only the constructor but the same techniques can be used with `init` as well. In both cases, `sc_vector` instantiates the given number of element objects automatically. This is fine for all element types that require a name as its only constructor argument. In our case, all component constructors expect more than one argument. For such element types requiring more constructor arguments, another version of the `sc_vector` constructor and `init` function is available expecting a so-called `Creator` as a further argument:

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

The SystemC LRM shows two cases how to use the `Creator`: passing a function object or passing a member function. C++11 offers a further option: passing an anonymous function.

A. Anonymous Function (Lambda)

C++11 introduced the anonymous function or lambda. It is a function definition that is not bound to an identifier. But lambda expressions return function objects and can be used to initialize such. Its syntax is

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

Therein, *capture* defines how variables are made accessible from within the lambda function body. In principle, all objects visible in the scope where the lambda definition resides can be made visible in the lambda body as well. The *capture* can specify and restrict this:

- []	captures nothing, no variables accessible in the lambda body
- [&]	captures all variables by reference if used in the lambda body
- [=]	captures all variables by value if used in the lambda body
- [a,&b]	captures a by value and b by reference

The capture is followed by the parameter list. This is the same as for other functions: A comma-separated list of parameter declarations is given in round brackets. Next is the return type definition. It consists of the dereference operator `->` and the actual return type. This part can be omitted if the function body contains nothing but a single return statement. Then, the return type is determined by the type of the returned expression.

Finally, we have the function body. This is again the same as in other functions: A list of statements embraced in curly brackets.

[2] Member function `create_element` can be used as well but will not be explained here.

B. *Populating the vector of PEs*

Now, we want to apply this to our platform model. As mentioned before, the `simple_node` contains a member `pes_`, which is an `sc_vector` of `simple_pe` elements. This vector is supposed to be populated via the `sc_vector` constructor in the `simple_node`'s constructor initializer list.
First, we want to look at the `simple_pe` constructor.

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

It expects three arguments, a name like all SystemC modules, an address used for internal identification of this PE and a pointer to a trace file for data logging. This is more than one constructor argument; hence, we need a `Creator` for the population of the vector. Instead of a function object or a member function, we can now use a lambda expression directly in the argument list of the `sc_vector` constructor.

```
1   simple_node::simple_node(  sc_core::sc_module_name n
2                            , size_t base_addr
3                            , size_t num_pes
4                            , sca_util::sca_trace_file * tf )
5     : sc_module(n)
6     , base_address_(base_addr)
7     , num_pes_(num_pes)
8     , pes_(  "pe"
9            , num_pes_
10           , [=](const char * nm, size_t i)
11             { return new simple_pe(nm, base_address_ + i, tf); })
12    , //...
```

The above listing shows the constructor definition of class `simple_node` with its member initializer list. The last part of the initializer list and the constructor body are omitted here because there is no difference to other constructors here.
In the first four lines, we see the constructor parameters of `simple_node`, a module name `n`, an integral number defining the base address of this node, an integral number defining the number of PEs in this node, and a pointer to a trace file. Then, the initializer list follows. In line 5, the module name is passed to the super class constructor, in line 6 and 7, the members `base_address_` and `num_pes_` are initialized, and in line 8 to 11, `pes_`, the vector of PEs, is initialized by calling the three-argument version of the `sc_vector` constructor.
The first argument is the base name for the vector elements and the second argument the number of vector elements. The third argument is the custom creator specified as a lambda expression here. It starts with the capture definition [=], i.e. all variables are accessible in the lambda function body and captured by value. Then, we have the parameter list consisting the name and index parameters as required for all `sc_vector` custom creators. Finally, there is the lambda function body. The return type can be omitted because we have only a single return statement. In the function body, all we have to do is creating a `simple_pe` object with the desired constructor arguments and returning it. Note that we can use lambda parameters (`nm` and `i`), members of `simple_node` (`base_address_`)[3], as well as parent constructor parameters (`tf`) here.
With this, we have a very simple and compact method to define custom creators for `sc_vector`. This simple form of lambda expressions is sufficient in most cases because the most common usage scenario for custom creators is passing arguments to the element constructor.

V. BINDING VECTORS WITH SC_ASSEMBLE_VECTOR AND AUTO

Next, we want to look at binding of vectors. In our use case, we look again into `simple_node`. Figure 2 shows a simplified structure of such a module. As mentioned in Section III, a `simple_node` contains vectors of sub-nodes, PEs, and bridges. Each of these components provides a single export for status information and all of these exports should be bound to a vector of exports in the `simple_node` module.

[3] Using members to initalize other members should be done with care; it can go wrong if the members are declared in the wrong order.

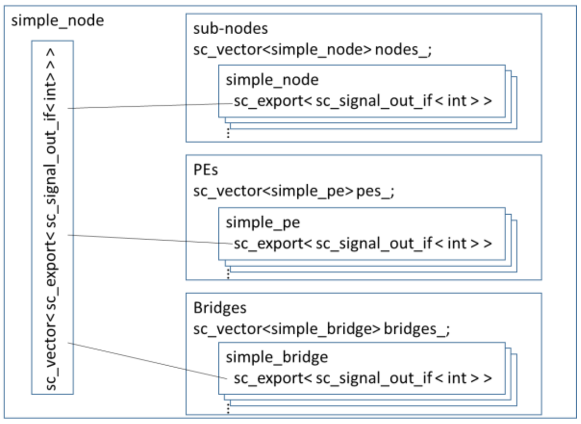
simple_node
sc_vector< sc_export< sc_signal_out_if< int> > >
sub-nodes
sc_vector<simple_node> nodes_;
simple_node
sc_export< sc_signal_out_if < int > >
PEs
sc_vector<simple_pe> pes_;
simple_pe
sc_export< sc_signal_out_if < int > >
Bridges
sc_vector<simple_bridge> bridges_;
simple_bridge
sc_export< sc_signal_out_if < int > >

Figure 2: Simplified structure of simple_node module

Before we can use the vector binding, we first need `sc_assemble_vector`. This function can always be used to assemble a new vector from members of vector elements, e.g., ports of modules in a vector.

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

In many cases, we can call bind directly on the object returned by `sc_assemble_vector` or pass it as argument to another bind call without storing it in a variable. We will see later, that this is not possible in our case. We need the `begin` and `end` iterators of the created assembly. Thus, we need to store it in a variable. To do so, we need a variable declaration, for the declaration we need the variable type, and here is our next problem: What is the actual return type of `sc_assemble_vector`? According to the declaration shown above, it is `sc_vector_assembly<T,MT>`. However, what are the exact values for `T` and `MT`? This does not matter in C++11 because we have `auto`.

A. Type Deduction with `auto`

Variable declarations with explicitly specified type are very hard in some cases because of the massive use of template types and template metaprogramming techniques in today's C++ libraries. To overcome this, C++11 introduced automatic type deduction with `auto`. This keyword can be used wherever the variable declaration contains an immediate initialization. It specifies that the type of the variable is automatically deduced from the initializer using the rules for template argument deduction.

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

This is especially helpful for complex template types like `sc_vector_assembly` or iterators.

B. Binding Vector Assemblies with `auto`

In our platform, we use `sc_assemble_vector` for the exports mentioned above. The exports in the sub-nodes, PEs and bridges are not given as vectors, but the components themselves are. The following statement assembles all members `status_` of the `simple_pe` objects in the vector `pes_` to a new vector.

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

With this, we are able to assemble three new vectors containing the status exports of the sub-nodes, PEs, and bridges. To bind then, we want to use the bind methods provided by `sc_vector`.

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

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

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

The first version of bind is for binding an entire other vector to this vector, the second expects iterators to determine the first and last element of another vector to be bound[4]. The last version expects an iterator pointing to an element of this vector as third argument. This method does not start binding with the first element but with the element referred to by the iterator passed as third argument. All three methods return an iterator referring to the first unbound element of this vector after the binding has been performed.

C. Using auto for sc_vector_assembly and iterators

We want to use the third version to bind the three blocks of exports. It allows us to set the iterator returned by the bind call as starting point for binding the second block. Now, auto comes into play because we need the begin and end iterators of the vector assembly. Furthermore, we can use auto for the declaration of the iterator referring to the first unbound element of the vector of exports.

```
1  auto va_n =
2    sc_core::sc_assemble_vector(nodes_,&simple_node::status_p_);
3  auto it_status = status_v_.bind(va_n.begin(), va_n.end());
4  auto va_pe =
5    sc_core::sc_assemble_vector(pes_,&simple_pe::status_p_);
6  it_status = status_v_.bind(va_pe.begin(), va_pe.end(), it_status);
7  auto va_br =
8    sc_core::sc_assemble_vector(bridges_,&simple_bridge::status_p_);
9  it_status = status_v_.bind(va_br.begin(), va_br.end(), it_status);
```

We see the four vectors in this example, the vector of exports status_v_ and the vectors of sub-nodes nodes_, PEs pes_, and bridges bridges_. Each simple_node, simple_pe, and simple_bridge contains an export named status_p_. In the first two lines in the listing above, we declare a variable va_n and initialize it with a vector assembly containing the status_p_ exports of all elements of vector nodes_. In line 3, this assembly is bound to the first elements of status_v_ using its begin and end iterators. In addition, a variable it_status is declared and initialized as an iterator referring to the first unbound element in status_v_. In the next two lines 4 and 5, another variable va_pe is declared to hold the vector assembly containing the status_p_ members of all simple_pe objects contained in pes_. Then, this assembly is bound to status_v_ in line 6. it_status occurs twice here. On one hand, it is passed to the bind method to define the starting point for this binding operation. On the other hand, the return value of bind is assigned to it to set it to the new *first unbound element* after the binding operation has been performed. Finally, the same is done for the vector of bridges in lines 7 to 9.
All variables that need to be declared here can be declared with auto. This avoids unnecessary typing of long type names and thinking about template arguments. Altogether, this results in a very compact way to implement the binding.

D. Using auto for loop iterators

Another very helpful use-case for auto is the initialization of iterators in for loops. Let us assume that we want to loop over the elements of the vector of exports mentioned above. Without auto, we would have to write the following.

```
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 ) {
     ...
```

With auto, we can reduce this to

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

[4] Like in other iterator based operations, the method starts binding with the element referred to by the first argument and does not bind any element including or following that referred to by the second argument.

And with the range-based for-loop feature introduced in C++11 as well, we can write this even more compact:

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

Writing compact code like that does not only lead to less typing during the implementation. The result is easier to understand and to maintain as well.

VI. Function Call as Template Argument with constexpr

The last feature we want to discuss is the possibility of doing more advanced calculations in template argument lists.

Template classes are used very widely in SystemC; not only with type parameters but as well with integer parameters. Examples are `sc_bv<W>`, `sc_lv<W>`, or `sc_int<W>`. Since template arguments have to be *compile time constant*, expressions that can be used here are very limited. In C++03, it is possible to use literals or constants of build-in C types and basic operators such as +, -, or * in such expressions. It was not allowed to call functions or use complex types.

In our platform, this leads to a problem because we need an `sc_bv` with a width equal to the number of bits required to represent a given integral number. To calculate this, we need to calculate the binary logarithm of the number and this calculation requires a loop statement. Loop statements are not *compile time constant* and are not allowed to be used in a template argument list. An implementation would be possible with template meta-programming expressions but this is very complicated and error-prone.

C++11 introduces *relaxed constant expressions* to overcome this and they are further relaxed in C++14.

A. *Relaxed Constant Expressions with* `constexpr`

Constant expressions in C++ are expressions like `4+5` that can be evaluated by the compiler. A major novelty in C++11 is that function calls are allowed in constant expressions as long as the function is specified as `constexpr` and fulfills the requirements for `constexpr` functions. The main requirements are that the function must have a non-void return type and the body may contain only declarations, null statements and a single return statement. All expressions therein have to be constant expressions themselves. A more complete list of the requirements can be found in [3].

In C++14, these limitations are further relaxed. For instance, conditional statement and loop statements are allowed now as well.

B. *Implementing log2(n) as Constant Expression*

The implementation of the binary logarithm is very simple now. We declare a function `log2` as `constexpr` and its body contains a single return statement only. The loop is implemented by a recursive call to `log2`.

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

Now, we can use the function `log2` in the initialization of other constants or as template argument.

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

The further relaxed limitations for constant expressions in C++14 enable the implementation of the same function as well with an if-statement or a while-loop.

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

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

In SystemC, this feature is very valuable for the calculation of widths of bit vectors (`sc_bv`, `sc_lv`) or arbitrary sized integers (`sc_int`, `sc_uint`, `sc_fixed`, …).

VII. Outlook: Extending sc_vector to Support Initializer Lists and push_back

Finally, we want to discuss possible extensions of `sc_vector` to support other methods to populate the vector.

A. *`std::initializer_list`* in C++11

Initializer lists[5] are well known in C-style arrays: A comma-separated list of objects in braces.

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

This concept is extended in C++11 by introducing a template class:

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

This allows instantiating initiator lists as objects and using them as parameters, for instance in constructors. An object of type `std::initializer<T>` encapsulates an array of objects of type `const T` and gives access to its elements. Most container classes in the C++ Standard Library like `std::vector` or `std::list` support this already by providing a constructor with an initializer list parameter.

B. *`std::initializer_list`* and *`sc_vector`*

To support initializer lists in `sc_vector`, we need to add an according constructor. The implementation of the constructor body is no problem. We can iterate over the initializer list and initialize the internal data structure in `sc_vector` with its elements. But, the element type of the `std::initializer_list` is a bit more problematic. In general, the lifetime of the array elements in an initiator list is bound to the initializer list object itself, and as a constructor argument, they are destructed by the end of the constructor. As a result, we have to copy the array elements, but this is not possible for SystemC objects because SystemC objects cannot be copied. A way to overcome the prohibition to copy SystemC elements is using an initializer list of pointers. A constructor declaration could look like:

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

With this, we need to provide a list of pointers to the constructor like in the following example.

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

The realization of this extension can be done with very little effort. All that needs to be done is the implementation of the new `sc_vector` constructor. However, it has drawbacks. The syntax is a bit strange because we have to use `new` in the list, and we cannot check if all the pointers are valid. When the user passes a pointer to a temporary to the list, the vector will not notice until it tries to dereference the pointer ending up with a memory access violation.

[5] Not to be confused with member initializer list which part of a constructor definition.

C. *Supporting* `push_back` *in* `sc_vector`

Another method to append elements to a `std::vector` is using `push_back`. The method appends the object given as argument to the vector. This is unsupported in sc_vector because it requires copying objects as well. In the following, we outline a way to allow this for pushing temporaries into a vector. It relies on the so-called *rvalue references* and *move semantics* introduced in C++11. Explaining rvalue references and move semantics in detail would go far beyond the scope of this paper. A more detailed description of these concepts can be found in [4].
In short, an *rvalue* is a temporary object, i.e. an object returned by a function or explicit constructor call and not assigned to an identifier. *Rvalue references*, identified by `T&&`, are non-const references to rvalues, i.e. references to modifiable temporaries. This allows the implementation of *move constructors* that take an rvalue reference as argument and 'steal' it. Instead of copying the argument to the new object, its content is *moved* to the new object and the argument may be left empty. In earlier C++ versions, it was not possible to distinguish between normal copy operations and copying from a temporary. Thus, initializing an object with a temporary requires a copy of the temporary and all its content because of the temporary's limited lifetime. With move, we can avoid such unnecessary deep copies by moving the temporary's content to the new object.
To support this, we need two things. The first is a `push_back` method in `sc_vector` with an rvalue reference parameter:

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

The second thing requires more effort: We have to make the supported element types *MoveConstructible*. In detail, this means implementing move constructors for all classes of SystemC objects that may be used as elements of `sc_vector`. This contains implicitly the definition of a move semantics for these objects and answering questions like:

- What members have to be copied and what members can be moved?
- What happens to the argument object or how does an 'empty' object look like?
- What bookkeeping tasks have to be performed regarding the SystemC object hierarchy?

When these questions are answered and the according constructors are provided, we can push temporaries to an `sc_vector` as follows:

```
sc_core::sc_vector< my_mod_t > v("my_mod_vector");
v.push_back(my_mod_t("ab"));
v.push_back(my_mod_t("cd"));
```

VIII. Conclusion

We presented an implementation example that uses new C++ features introduced in C++11 and C++14 in combination with `sc_vector` to facilitate the automated instantiation of SystemC models. The practical use of these features for SystemC designers has been shown by code examples. Finally, we discussed the implementation of a support for initializer lists and `push_back` in `sc_vector` as an outlook to future work.

References

[1] P. Knocke, R. Görgen, J. Walter, D. Helms, and W. Nebel, „Using early power and timing estimations of massively heterogeneous computation platforms to create optimized HPC applications", Proceedings of 2014 International Conference on Embedded and Ubiquitous Computing - EUC 2014.
[2] IEEE Computer Society, "IEEE Standard for Standard SystemC Language Reference Manual", IEEE Std 1666-2011, Jan 2012.
[3] "C++ Reference: constexpr specifier", http://en.cppreference.com/w/cpp/language/constexpr
[4] Scott Meyers, "Effective Modern C++", O'Reilly Media 2014, ISBN 1-491-90399-5