

# OO Features supported by the SystemC™ *Plus* Methodology

v0.2, 29-10-2002

Eike Grimpe

## 1. General restrictions

- no use of pointers (except for sub-module declaration in module declarative part, and the use of `*this` within member functions)
  - rationale: pointer synthesis leads to very inefficient hardware
- no dynamic memory allocation/de-allocation, e.g. use of `new` and `delete` (except for sub-module instantiation within module constructor body)
  - rationale: static nature of hardware (FPGA may change this one day)
- no use of address-operator, e.g. `operator&` (except for passing function parameters by reference)
  - rationale: does not make sense without pointers and dynamic memory allocation/de-allocation. Hard to synthesise
- no reference return types
  - we actually lack an adequate synthesis model for this kind of return mechanism (see also notes on functions)
- no use of explicit cast operators, e.g. `dynamic_cast`, `static_cast`, `reinterpret_cast`, `const_cast`
  - rationale: limited use without pointers. Casting can be a dangerous operation. Really needed? For polymorphic objects there is a cast mechanism (see also Polymorphism)
- no use of `sizeof`
  - rationale: use for modelling hardware not clear. Synthesis semantics not clear
- no use of `goto`
  - rationale: thanks Dijkstra!
- no reference types (except for function parameters, see above)
  - rationale: just an alias, isn't it? Useful application?
- no static members
  - rationale: synthesis semantics for classes. How to synthesise a global variable?
- no `mutable`, `volatile`
  - rationale: no need to play around with the memory when modelling hardware. No meaning for hardware(?)
- no destructors
  - rationale: does not make sense without `delete`. Hardware exists for ever
- no support for floating-point data types, e.g. `double`
  - rationale: unclear how to synthesise efficiently
- no use of post- and pre-increment/decrement, e.g. `++`, `--`

- rationale: language features which can cause a lot of semantic problems. We are lacking a proper synthesis semantics for pre-decrement and -increment
- no I/O, e.g. use of `cout`, `cerr`, `printf`, ...
  - rationale: no meaning for hardware
- no use of SystemC 2.0 channels (except for `sc_signal`)
  - rationale: synthesis semantics not clear. Show improper simulation behaviour w.r.t. hardware (temporal behaviour!). Proposed alternative: refer to Global Objects

## 2. Classes

- Declaration: as used from C++

```
Class MyClass { ... };
```

- Data member declaration: as used from C++

```
Class MyClass {
private:
    int integerMember;
    const int constantIntegerMember;
    AnotherClassType objectMember;
    short arrayMember[10];
};
```

- Access specifier (**public**, **protected**, **private**): as used from C++ but without meaning for synthesis

- Constructors: as used from C++

```
Class MyClass {
public:
    MyClass() : constantIntegerMember( 0 ) {}

    MyClass( const int val ) : constantIntegerMember( 0 ),
                             objectMember( val ) {
        integerMember = val;
    }
    ...
};
```

- Member function declaration: as used from C++

```
Class MyClass {
public:
    void
    inc( int val = 1 ) {
        integerMember += val;
    }
};
```

```

MyClass
getClone() {
    return( *this );
}

bool
compareTo( const MyClass & obj ) { ... }
...
};

```

- Operator overloading: as used from C++

```

Class MyClass {
public:
    bool
    operator==( const MyClass & obj ) { ... }

    void
    operator=( const MyClass & obj ) { ... }
    ...
};

```

- Instantiation: as used from C++. Instantiation is possible within at top of each process body (**SC\_METHOD** and **SC\_CTHREAD**), e.g. before the reset part of an **SC\_CTHREAD**, and at the top of each function body. An object may also be declared as member of a module, but must then be accesses exclusively by only one process (**SC\_METHOD**), and the underlying class must not contain any default constructor or an empty default constructor.

```

void mySCTHREAD() {
    MyClass obj1;
    int val;
    MyClass obj2( 6 );
    if ( reset == true ) {
        ...
    }
    wait()
    while( true ) {
        ...
    }
}

void
func() {
    MyClass lObj( 127 );
    ...
}

```

- Member access: as used from C++

```

void mySCTHREAD() {
    ...
    wait()
    while( true ) {
        obj1.integerMember = 10;
        val = obj1.integerMember;
        obj1.objectMember.anotherMember = true;
        obj1.inc();
        obj1.inc( 42 );
        if ( !(obj1 == obj2) ) // uses overloaded ==
        {
            obj1 = obj2.getClone(); // uses overloaded =
            obj2.arrayMember[5] = 5;
            ...
        }
    }
}

```

#### Notes:

- Refer also to general restrictions, i.e. declaration of pointer members or static members is not allowed.

#### Synthesis:

- Class instances are translated into bit vectors (more precisely arbitrary sized integers in SystemC).
- Member functions and constructors are translated into non-member functions, with an additional attribute representing the original object.
- Member function calls are synthesised to function calls to the synthesised non-member functions. The original prefix is passed as additional argument.
- Data member accesses are translated into accesses to the respective slices of the synthesised object.
- Mem-initializers of constructors are moved into the constructor body during synthesis.
- For class type members, being implicitly initialised by a default constructor, proper constructor invocation is added to the constructor body during synthesis.

#### Support by actual synthesiser prototype:

- All features listed above are fully supported.

### 3. Inheritance

- Simple inheritance: as used from C++

```

Class Top {
public:
    int member;
    int topMember;
};

class Middle : public Top {

```

```

public:
    int member;
    int middleMember;

    void
    inc( int val ) { ... }
};

class LeftBottom : public Middle {
public:

    void
    inc( int val ) { ... }
};

class RightBottom : public Middle {
public:

    void
    inc( int val ) { ... }
};

```

- **Qualified access: as used from C++**

```

Middle middleObj;
LeftBottom bottomObj;

middleObj.topMember = 1;
middleObj.middleMember = 2;
middleObj.Top::member = 3;
middleObjec.Middle::= 4;

bottomObj.inc( 10 );
bottomObj.Middle::inc( 20 );

```

- **Multiple inheritance: as used from C++**

```

Class Top {
public:
    int topMember;
};

class Left : public Top {
    int leftMember;
};

class Right : public Top {
    int rightMember;
};

class Bottom : public Left, public Right {
    int bottomMember;
};

```

```

};

...

Bottom bottomObj;
// Note, different members are accessed:
bottomObj.Left::topMember = 1; // Top is inherited twice
bottomObj.Right::topMember = 2; // Top is inherited twice

```

- **Virtual base classes: as used from C++**

```

Class Top {
public:
    int topMember;
};

class Left : public virtual Top {
    int leftMember;
};

class Right : public virtual Top {
    int rightMember;
};

class Bottom : public Left, public Right {
    int bottomMember;
};

...

Bottom bottomObj;
// Note, the same member is accessed three times:
bottomObj.topMember = 1;
bottomObj.Left::topMember = 2;
bottomObj.Right::topMember = 3;

```

#### **Notes:**

- Specifiers **public**, **protected** and **private** can be used for deriving new classes as usual, but do not have meaning for synthesis.

#### **Synthesis:**

- Synthesis strategy is the same as for normal classes, but synthesis process itself and determination of correct location of members is more complex.
- Virtual classes are treated correctly, e.g. members of virtual classes that are derived via different paths will only appear once in every synthesised object.
- For default constructors of ancestor classes, being implicitly invoked, proper constructor invocations are added to a constructor body during synthesis.

#### **Support by actual synthesiser prototype:**

- All features listed above are fully supported.
- Qualification depth greater than one probably does not work yet (but does not seem to be a principal problem)

## 4. Functions

- Declaration: declaration as used from C++ is possible either as a global function inside any namespace, as member function (not a process) of a module or as member-function inside a class declaration

```
// file global_declarations.hh
void
someFunc() { ... }

SC_MODULE( myModule ) {
    void
    anotherFunc() { ... }
    ...
};

Class MyClass {
public:
    MyClass
    aMemberFunc() { ... }
};
```

- Parameter types: parameters for functions (non-member functions, member functions and constructors) may be of all supported types, including polymorphic objects (see Polymorphism). Parameters may be either past by value or by reference and either declared constant or non-constant

```
void
func( Top & obj1,
      const Left & obj2,
      Right obj3,
      const int val ) { ... }
```

- Return types: can be void or any other supported type. Values must be returned by-value

```
void
proc() { ... }

bool
func() { ... }

MyClass
getClone() { ... }
```

- Default values: may be specified as used from C++

```
void
func( Top & obj1,
      const Left & obj2,
```

```
Right obj3,
const int val = -42 ) { ... }
```

- Specifiers: member functions may be declared to be **const**, **virtual** or **static** as used from C++. Virtual declarations will have significance for polymorphic objects (see Polymorphism) only.

```
MyClass
getClone() const { ... }

virtual
bool
Base::virtualFunc() { ... }

// Note, declaration of static data members is not
// allowed
static
unsigned int
MathClass::pow( const int val, const int exp ) { ... }
```

#### Notes:

- Effects of **const** return types on synthesis have not yet been studied.
- Whether to allow global objects (see Global objects) as parameters is still a question. At least, global objects must not be passed to other global objects by reference.
- Returning values by reference may be a desirable feature for allowing controlled access to complex members of an object, but we do not yet have an adequate synthesis concept for this feature.

#### Synthesis:

- Synthesis of member functions has already been outlined in section Classes
- For non-const parameters passed by value, a variable declaration will be added to the respective function definition, to which the parameter is copied first. Each occurrence of the original parameter name within the function definition is replaced by the newly added variable name. This is more similar to VHDL, where a parameter passed by value can only be read and not generally used as local variable.
- If a class instance is passed by value, invocation of copy constructor is properly synthesised, if such a constructor exists.

#### Support by actual synthesiser prototype:

- Ports and Signals as parameters are not yet supported.
- Non-constant by-value parameters are not yet supported.
- All other features listed above are supported.

## 5. Templates

- Class templates: declaration and usage as used from C++

```
template< class ELEMENTTYPE, unsigned int SIZE = 8>
class Buffer {
```

```

ELEMENTTYPE array[SIZE];
};

Buffer< int, 256 > myBuffer;
Buffer< ComplexNumber > anotherBuffer;

```

- Inheritance from class templates: as used from C++

```

template< class ELEMENTTYPE >
class FixedSizeBuffer : public Buffer< ELEMENTTYPE, 16 > {
};

```

- Function templates: see Notes
- Module templates: user defined modules can be declared as templates, having scalar and type template parameters.

```

template< class ELEMENTTYPE, unsigned int SIZE = 8>
SC_MODULE( HasABuffer ) {
    void
    syncProcess() {
        Buffer< ELEMENTTYPE, SIZE > myBuffer;
        ...
    }
    ...
};

```

#### Notes:

- Template functions have not yet been regarded, but should not be any problem for synthesis (in principle same synthesis as for class templates).
- Passing modules (module types) as template argument has not yet been regarded, but, this should not be any problem for synthesis.

#### Synthesis:

- Template parameters are evaluated during synthesis. Afterwards, class templates can be treated as every other class.

#### Support by actual synthesiser prototype:

- Class templates, scalar and type template parameters and inheritance from class templates are supported.
- Template specialization has not yet been tested.
- Features listed in the Notes section are not yet supported.

## 6. Polymorphism

Due to the use of pointers, the C++ polymorphism mechanism doesn't fit our synthesis requirements. We therefore propose and support a mechanism which is more similar to a tagged type concept, though using the C++ mechanisms for implementation. In contrast to C++, SystemC-Plus provides so called polymorphic objects which have their own state space, but can "virtually" change their class membership during runtime. As used from C++, virtual functions called on such an object are dynamically dispatched.

- **Declaration:** a polymorphic object can be declared, wherever a normal object can be declared. A polymorphic object declaration requires to specify a root class for the polymorphic object.

```
class Base;
PolyObject< Base > pObj;
sc_signal< PolyObject< Base > > pSignal;
```

- **POLYMORPHIC specifier:** each class used as root class of a polymorphic object and each class of which instances are assigned to polymorphic objects needs to include a polymorphic specifier in its declaration

```
class Base {
    POLYMORPHIC( Base )
    ...
};

class Derived : public Base {
    POLYMORPHIC( Derived )
    ...
};

PolyObject< Base > basePolyObj;
Derived derivedObj;

basePolyObj = derivedObj;
```

- **Assignment to and from polymorphic objects:** the same rules as used from C++ have to be applied, e.g. instances of the polymorphic object's root class and all derived classes can be assigned to a polymorphic object and other polymorphic objects whose root class is the same or any derived class. A polymorphic object can be assigned to instances of its root class, instances of classes from which the root class is derived (ancestors) and other polymorphic objects of the same or any ancestor root class.

```
class Base { ... };
class Derived : public Base { ... };

Base baseObj;
Derived derivedObj;
PolyObject< Base > polyBaseObj;
PolyObject< Derived > polyDerivedObj;
sc_signal< PolyObject< Base > > polySignal;

polyBaseObj = baseObj;
polyBaseObj = derivedObj;
polyBaseObj = polyDerivedObj;
derivedObj = polyDerivedObj;
derivedObj = polyBaseObj;
polySignal.write( derivedObj );
polySignal.write( polyBaseObj );
...
```

- Member access: members of a polymorphic object must be accessed by means of operator ->

```
class Base {
public:
    int myMember;

    void
    func() { ... };
};
```

```
PolyObject< Base > polyBaseObj;
polyBaseObj->myMember = 7;
polyBaseObj->func();
```

- Casting: polymorphic objects provide a **cast** operation that can be used for assignments of “potentially assignment compatible” polymorphic objects. Using this operation is on own risk!

```
Derived derivedObj1, derivedObj2;
PolyObject< Base > polyBaseObj;
```

```
polyBaseObj = derivedObj1;
derivedObj2 = polyBaseObj.cast< Derived >();
```

- Dynamic dispatching: as used from C++, calls to member functions that are declared **virtual** are dynamically dispatched.

```
class Base {
public:
    void
    nonVirtFunc() { ... };

    virtual
    void
    virtFunc() { ... };
};
```

```
class Derived : public Base {
public:
    void
    nonVirtFunc() { ... };

    virtual
    void
    virtFunc() { ... };
};
```

```
Base baseObj;
Derived derivedObj;
```

```

PolyObject< Base > polyBaseObj;

polyBaseObj->virtFunc(); // Base::virtFunc()
polyBase->nonVirtFunc(); // Base::nonVirtFunc()
polyBaseObj = derivedObj;
polyBaseObj->virtFunc(); // Derived::virtFunc()
polyBase->nonVirtFunc(); // Base::nonVirtFunc()

```

### Notes:

- Each class type of which instances are assigned to a polymorphic object must support the assignment operator (=), either by default or by overloading it.
- If exists, the default constructor of the class being specified as root class for a polymorphic object is invoked for initialising the polymorphic object.
- If a class type of which instances are assigned to a polymorphic object declares other constructors than the default constructor, it must also declare a default constructor, even if it has an empty body.
- A root class of a polymorphic object must not be abstract, e.g. must not contain any pure virtual function declarations. This is, because a polymorphic object does have its own state space, and therefore memory is already allocated at declaration.
- In order to be compliant with the SystemC naming conventions, renaming of PolyObject into sc\_polymorphic, and POLYMORPHISM into SC\_TAGGED would be possible.

### Synthesis:

- Synthesised bit vector representing a polymorphic object is chosen large enough, to provide sufficient space for all instances of classes being assigned to it.
- A tag is added to the state space of the synthesised polymorphic object, that is used to determine its class membership at runtime.
- The appropriate implementation to be invoked for a dispatched function call is chosen by means of the actual tag value.

### Support by actual synthesiser prototype:

- Cast operation is not yet supported.
- All other features listed above are supported but maybe not yet fully tested.

## 7. Global Objects

Global objects are similar to protected objects in Ada95 and allow to model inter-process communication on higher level of abstraction but still synthesisable. SystemC-Plus provides global objects instead of channels, because global objects have a clear synthesis semantics which is also reflected during simulation.

- Declaration: a global object must be declared as member of a module. A global object declaration requires to specify a scheduler class, determining the scheduling policy for solving concurrent accesses, and to specify any user defined class, which will implement the global object's behaviour.

```

class Base;
template< class ELEMENTTYPE, unsigned int SIZE >
class Buffer {

```

```

public:
    GUARDED_METHOD( bool,
                    isFull(),
                    true );

    GUARDED_METHOD( bool,
                    isEmpty(),
                    true );

    GUARDED_METHOD( void,
                    put( const ELEMENTTYPE & element ),
                    !isFull() );

    GUARDED_METHOD( ELEMENTTYPE,
                    get(),
                    !isEmpty() );

private:
    ELEMENTTYPE array[ SIZE ];
};

SC_MODULE( myModule ) {
    GlobalObject< Buffer< int, 8 > > globalObj;
    ...
};

```

- Client registration: clients of global objects must be synchronous processes (**SC\_CTHREAD**). All clients must be driven by the same clock. Registration to a global object must be done before or in the reset part of the synchronous process, by means of the subscribe method the global object provides. An optional argument, a positive integer number, can be passed for being used for scheduling.

```

SC_MODULE( myModule ) {
    GlobalObject< RoundRobin, Buffer< int, 8 > > globalObj;
    ...
    void
    producer() {
        globalObj.subscribe();
        ...
    }

    void
    consumer() {
        globalObj.subscribe();
        ...
    }
}

```

- Reset: a global object must be reset by at least one client before first use. For this purpose all global objects provide a **reset** operation, that must be called in a client's reset part.

```

void
producer() {
    if ( reset == true )
    {
        globalObj.reset();
        globalObj.subscribe();

    }
    ...
}

```

- **Scheduling:** concurrent accesses to a global object are solved by means of a scheduler. Three different schedulers are “built-in”; **RoundRobin**, **ModifiedRoundRobin** and **FixedPriority**. A user may create his own scheduler classes, by deriving them from the base class **Scheduler** and by following certain coding guidelines.

```

SC_MODULE( myModule ) {
    GlobalObject< FixedPriority, Buffer< int, 8 > >
                                                globalObj;

    ...
    void
    producer() {
        globalObj.subscribe( 10 );
        ...
    }

    void
    consumer() {
        globalObj.subscribe( 5 );
        ...
    }
    ...
};

MyOwnScheduler : public Scheduler {
    ...
};

GlobalObject< MyOwnScheduler, Buffer< int, 8 > >
                                                globalObj;

```

- **Access:** global objects must only be accessed by means of the macros **GLOBAL\_PROCEDURE\_CALL**, **GLOBAL\_FUNCTION\_CALL** and **GLOBAL\_ASSIGN**. The first one has to be used for calling methods with **void** return type, the next one for calling methods with return types others than **void** and the last one for assignments to global objects. Access is mutual exclusive and blocking.

```

SC_MODULE( myModule ) {
    GlobalObject< FixedPriority, Buffer< int, 8 > >

```

```

                                                                    globalObj;
...
void
producer() {
    ...
    while ( true ) {
        val += 1;
        GLOBAL_PROCEDURE_CALL( globalObj, put( val ) );
        wait();
    }
}

void
consumer() {
    ...
    while ( true ) {
        GLOBAL_FUNCTION_CALL( globalObj, get(), val );
        output.write( val );
        wait();
    }
}

void
dummy() {
    Buffer< int, 8 > l_buffer;
    ...
    while ( true ) {
        GLOBAL_ASSIGN( globalObj, l_buffer );
        wait();
    }
}

...
};

```

- **Guarded methods:** each method of a global object being called by a client must be declared guarded. A guarded method associates a guard expression with a method. A guard expression must be a Boolean expression. A client is only regarded for scheduling, if the guard expression associated with the method he is calling evaluates to true. A guarded method is declared by means of the macro **GUARDED\_METHOD**.

```

template< class ELEMENTTYPE, unsigned int SIZE >
class Buffer {
public:
    GUARDED_METHOD( bool,
                    isFull(),
                    true );

    GUARDED_METHOD( bool,
                    isEmpty(),
                    true );

```

```

    GUARDED_METHOD( void,
                    put( const ELEMENTTYPE & element ),
                    !isFull() );

    GUARDED_METHOD( ELEMENTTYPE,
                    get(),
                    !isEmpty() );

private:
    ELEMENTTYPE array[ SIZE ];
};

```

- **Binding:** like ports, global objects of the same type located in different modules can be bound to each other. Global object binding allows communication across module boundaries. A global object binding statement must be located in the body of the module constructor of the enclosing/top module, after the sub-module containing the global object that shall be bound is instantiated.

```

SC_MODULE( Sub ) {
    GlobalObject< FixedPriority, Buffer< int, 8 > >
                                   globalObj;

    ...
};

SC_MODULE( Top ) {
    GlobalObject< FixedPriority, Buffer< int, 8 > >
                                   globalObj;

    Sub * subModule;
    SC_CTOR( subModule ) {
        subModule = new Sub( "subModule" );
        subModule->globalObj( globalObj );
    }
};

```

#### Notes:

- A global object is a passive object, e.g. does not have its own thread of control. It only performs any action while triggered from outside.
- The macro `global assign` is provided in order to directly allow the use of an overloaded `operator=` if exists.
- The guard mechanism is not invoked on local instances (within a process or function body) of classes.
- For access of global objects from within a testbench (`sc_main`) an own set of macros is provided and must be used (uses internally `sc_start` instead of `wait()`).
- Each access to a global object takes at least three clock cycles (immediate grant presumed) because of the underlying protocol that is applied. More cycles may be necessary dependent on the execution time of the called method and if the behavioural compiler is used for back-end synthesis.

- It is also possible to declare polymorphic global objects by means of the template `GlobalPolyObject`. But we should reconsider, if a polymorphic global object is really desirable, or if it is a verification nightmare.
- In order to be compliant with the SystemC naming conventions, renaming of `GlobalObject` into `sc_sharedobject`, `GLOBAL_PROCEDURE_CALL` into `SC_PROCEDURECALL`, `GLOBAL_FUNCTION_CALL` into `SC_FUNCTIONCALL`, `GLOBAL_ASSIGN` into `SC_ASSIGN`, and `GUARDED_METHOD` into `SC_GUARDED` would be possible.

**Synthesis:**

- A client server structure is synthesised for global objects with a fixed access protocol.
- From the global object itself, an own module will be synthesised. All processes within the synthesised module are driven by the same clock, which is derived from the clients.

**Support by actual synthesiser prototype:**

- Global objects are not yet supported.