SystemC Synthesizable Subset

Draft 1.1.18

Described:
Synthesis Working Group of Open SystemC Initiative

Date:
December 23, 2004

Introduction

This standard describes a standard syntax and semantics for SystemC synthesis. It defines the subset of SystemC 2.0.1 that is suitable for RTL/behavioral synthesis and defines the semantics of that subset for the synthesis domain. This standard is based on the standards of C++ and SystemC 2.0.1.

The purpose of this standard is to define a syntax and semantics that can be recognized in common by all compliant RTL/behavioral synthesis tools to achieve behavioral uniformity of results in a similar manner to which simulation tools use the SystemC standard. This will allow users of synthesis tools to produce well defined designs whose functional characteristics are independent of a particular synthesis implementation by making their designs compliant with this standard.

The standard is intended for use by logic designers, electronic engineers and design automation tool developers.

The following team members drove this effort:
Eike Grimpe
Rocco Jonack
Masamichi Kawarabayashi, Chair
Mike Meredith
Fumiaki Nagao
Andres Takach
Yutaka Tamiya
Minoru Tomobe

A majority of the work conducted by the working group was done via teleconferencing which was held regularly. Also, the working group used an email reflector and its web page effectively to distribute and share information.

Contents

1 Overview

1.1 Scope

Synthesis Working Group (SWG) of Open SystemC Inititive(OSCI) developed a definition of Synthesizable-SystemC (SSC). This will be useful not only for hardware designers to accelerate the modeling and design process with SystemC, but also for EDA tool developpers to develop SystemC compliant synthesis tools. SSC will be defined within C++ and SystemC 2.0 specifications, but SWG may propose extensions of SystemC and additional libraries for efficient synthesis for future possibilities.

1.2 Purpose

Users will be able to develop any system with SystemC at any abstraction level, and verify it with reference implementation available on the OSCI web-site under the valid license agreement. Currently available resources providing a description of the language include a User Guide, a Functional Specifications, and the Language Reference Manual. Some books for SystemC modeling were also published recently. However, currently there are no defined standards for synthesizable description in SystemC. We will call this "Synthesizable-SystemC (SSC)" here.

It is widely known that synthesis technology reduces the time required for hardware design dramatically, and is one of the most important phases in the design flow. In order to utilize the powerful SystemC based design environment, SSC is essential for top-down design with SystemC.

SSC consists of a definition of a synthesizable subset of the SystemC language along with coding guidelines. The synthesizable subset defines which syntactic elements in SystemC should be synthesized with synthesis tools. It covers at least the register transfer level and the behavioral level. More abstraction levels are also discussed in this working group as the next step. The synthesis tools that support this subset completely can be identified as being Synthesizable-SystemC compliant.

The coding guidelines assist hardware designers to describe synthesizable SystemC codes efficiently. Coding guidelines at the register transfer level and at the behavioral level may be individually described to facilitate each design phase.

SWG developed two requirement documents. “Basic Requirements of SystemC Subset for Synthesis” describes synthesis requirements that have been identified which can be met within the SystemC 2.0.1 syntax and which have clear methodology for synthesis. “Advanced Requirements of SystemC Subset for Synthesis” describes synthesis requirements that have been identified which cannot be met within the SystemC 2.0.1 syntax or which have no clear methodology for synthesis.

This document is the definition of a synthesizable subset of SystemC, as discussed and agreed upon by the Synthesis Working Group. The intent of this document is to describe a minimum initial subset which can be supported by tools. It is not meant to restrict synthesis support for syntax beyond this subset.

1.3 Terminology

The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*). The word *should* is used to indicate that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*). The word *may* indicates a course of action permissible within the limits of the standard (*may* equals *is permitted*).

A synthesis tool is said to *accept* a SystemC construct if it allows that construct to be legal input; it is said to *interpret* the construct (or to provide an *interpretation* of the construct) by producing something that represents the construct. A synthesis tool is not required to provide an interpretation for every construct that it accepts, but only for those for which an interpretation is specified by this standard.

The constructs in the standard shall be categorized as:

Supported: Synthesis shall interpret a construct, that is, map the construct to an equivalent hardware representation.

Extended: Synthesis shall interpret the extended construct from the original C++ syntax.

Ignored: Synthesis shall ignore the construct. Encountering the construct shall not cause synthesis to fail, but synthesis results may not match simulation results. The mechanism, if any, by which synthesis notifies (warns) the user of such constructs is not defined by this standard. Ignored constructs may include unsupported constructs.

Not Supported: Synthesis does not support the construct. Synthesis does not expect to encounter the construct and the failure mode shall be undefined. Synthesis may fail upon encountering such a construct. Failure is not mandatory; more specifically, synthesis is allowed to treat such a construct as ignored.

1.4 Conventions

This standard uses the following conventions:

a) The body of the text of this standard uses **boldface** to denote SystemC or C++ reserved words (e.g. **sensitive**).
b) The text of the SystemC examples and code fragments is represented in a `fixed-width font`.
c) Syntax text that is struck-through (e.g. ~~text~~) refers to syntax that shall not be supported.
d) Syntax text that is underscored (e.g. <u>text</u>) refers to syntax that shall be ignored.
e) Syntax test that is shadowed (e.g. text) refers to syntax that shall be extended.

f) Any paragraph starting with "NOTE--" is informative and not part of the standard.
g) The examples that appear in this document under "*Example:*", are for the sole purpose of demonstrating the syntax and semantics of SystemC for synthesis. It is not the intent of this standard to demonstrate, recommend, or emphasize coding styles that are more (or less) efficient in generating an equivalent hardware representation. In addition, it is not the intent of this standard to present examples that represent a compliance test suite, or a performance benchmark, even though these examples are compliant to this standard (except as noted otherwise).

1.5 Reference

ISO/IEC 14882: Programming languages - C++, 1998
ISO/IEC 9899: Programming languages – C, 1999
SystemC 2.0.1 Language Reference Manual, Revision 1.0, Open SystemC Initiative, 2003

2 Translation units and their analysis

A translation unit consists of a set of declarations. A translation unit contains less than one **sc_main** function and must not contain any main function.

```
translation-unit ::=
        [ declaration-seq ] [sc-main-definition]
```

A synthesis tool may ignore whole of the **sc_main** function, or may recognize the **sc_main** function in order to obtain actual parameters of template modules and/or modules whose constructor takes parameters.

Example:

```
int sc_main(int argc, char* argv[])
{
  ...
  // Instance of a template module.
  AndGate<4> and_inst("and");  // The template parameter is 4.

  // Instance of a module with a parameter.
  Increment inc_inst("inc", 5);  // The parameter of constructor is 5.
  ...
}
```

3 Modules

A SystemC module represents an individual identifiable hardware element. A module is defined by a module definition.

3.1 Module definitions

A module definition defines the port-level interface of a module, its internal storage elements and its behavior. The synthesizable subset supports modules declared as a class or as a struct. In addition, specialization of modules using templates is supported.

There are three supported possibilities for module definition.

- Use of the SC_MODULE macro
- Direct derivation from sc_module
- Derivation from a class or struct derived from sc_module

sc-module-specifier ::=
  sc-module-head **{** [module-member-specification] **}**

sc-module-head ::=
  SC_MODULE (identifier **)**
  | class-key [nested-name-specifier] identifier **:** [**public**] **sc_module**

Each class or structure which is derived from **sc_module** or which is declared using the **SC_MODULE** macro is called a module.

The class key for a module declaration must either be **class** or **struct**.

3.1.1 Module member specification

The module member specificaton contains a set of member declarations and definitions. This set must include exactly one module constructor declaration or definition.

sc-module-member-specification ::=
  sc-module-member-declaration [sc-module-member-specification]
  | access-specifier **:** [sc-module-member-specification]

3.1.2 Module declarative items

Items which are declared in the module body are available for use by all functions defined within the scope of the module.

sc-module-member-declaration ::=
  member-declaration
  | sc-signal-declaration
  | sc-sub-module-declaration
  | sc-module-constructor-definition
  | sc-module-constructor-declaration
  | sc-has-process-declaration

Apart from the module constructor a module must not declare any special member function or overload any operator.

No member of a module must be accessed from outside of the scope of that module on an instance of that module, i.e. by means of the member access operators `.`, `*.` or `->`.

Shared variables (not meaning signals and ports) are not supported for synthesis. If a module body contains a variable declaration, the variable must be accessed exclusively by one process of the same module.

A module which does not include any constructor declaration using the **SC_CTOR** macro and which includes process declarations must contain exactly one has-process-declaration. A module which includes a module constructor using the **SC_CTOR** macro must not include any has-process-declaration.

Example:

```
SC_MODULE( mod ) {
      void dummy() {}
      SC_HAS_PROCESS( mod );   // error: a module which contains the
                               // SC_CTOR macro must not contain a
                               // has-process-declaration
      SC_CTOR( mod ) {
            SC_METHOD( dummy );
      }
};
```

3.1.2.1 Ports and signals

Ports represent the externally visible interface to a module and are used to transfer data into and out of the module. Signals can keep values and interface between processes. Ports and signals can be declared according to the following syntax:

```
sc-signal-declaration ::=
          sc-signal-key < type-specifier > signal-declarator-list ;
        | sc-resolved-key init-declarator-list ;
        | sc-resolved-vector-key < constant-expression > signal-declarator-list ;

signal-declarator-list ::=
          identifier
        | signal-declarator-list , identifier

sc-signal-key ::=
          sc_signal
        | sc_in
        | sc_out
        | sc_inout
        | sc_in_clk
        | sc_out_clk
        | sc_inout_clk

sc-resolved-key ::=
```

sc_signal_resolved
| **sc_in_resolved**
| **sc_out_resolved**
| **sc_inout_resolved**

sc-resolved-vector-key ::=
sc_signal_rv
| **sc_in_rv**
| **sc_out_rv**
| **sc_inout_rv**

3.1.2.2 Module constructor

Every module declaration must contain exactly one declaration or definition of a constructor method. Submodule instantiations, port mappings, and process statements are located in the module constructor. In contrast to normal C++ practice, initialization behavior must be placed in the reset clause of the process methods as opposed to residing in the constructor. Initialization of constant data members in the constructor is permitted. There are two ways to declare a module constructor.

sc-module-constructor-declaration ::=
SC_CTOR(identifier **) ;**
| identifier **(sc_module_name** [identifier] [**,** parameter-declaration-list] **) ;**

Identifier must be the name of the enclosing module in which the constructor is defined. If a module constructor is only declared but not defined, a definition of the module constructor must follow elsewhere in the translation unit.

sc-module-constructor-definition ::=
SC_CTOR(identifier **)** [ctor-initializer] sc-module-constructor-body
| identifier **(sc_module_name** identifier [**,** parameter-declaration-list] **)**
: sc_module (identifier **)** [**,** mem-initializer-list] sc-module-constructor-body

If a module constructor is declared without using the **SC_CTOR** macro, it must declare at least one parameter of type **sc_module_name** and it must include at least one mem-initializer passing the name of that parameter to the parent class '**sc_module**'.

sc-module-constructor-body ::=
{ [sc-module-constructor-element-seq] **}**

sc-module-constructor-element-seq ::=
sc-module-constructor-element
| sc-module-constructor-element-seq sc-module-constructor-element

sc-module-constructor-element ::=
sc-module-instantiation-statement
| sc-port-binding-statement
| sc-process-statement

Examples:

// A module declaration with port declaration only:

```
SC_MODULE( FullAdder ) {
    sc_in< bool> X, Y, Cin;
    sc_out< bool > Cout, Sum;
    SC_CTOR( FullAdder ) {}
};
```

// A module template declaration (without any functionality):

```
template< unsigned char N = 2 >
SC_MODULE( AndGate ) {
              sc_in< sc_uint< N > > Inputs;
    sc_out< bool> Result;
    SC_CTOR( AndGate ) {}
};
```

// A module declaration with constant declaration and initialisation:

```
SC_MODULE(Increment) {
    sc_in<sc_int<16> > A;
    sc_out<sc_int<16> > X;
    const int m_delta; // initialized by the constructor
    SC_HAS_PROCESS(Increment);

    Increment(sc_module_name& name_, int delta)
     : sc_module(name_), m_delta(delta)
    {
      SC_METHOD(proc);
      sensitive << A;
    }

    void proc() {
      X = A.read() + m_delta;
    }
};
```

3.2 Deriving modules

A module may be declared as a specialization of an existing module by derivation from the existing module. In this case the module must be declared explicitly using the class keyword (SC_MODULE may not be used) and its constructor must be declared explicitly (SC_CTOR may not be used). If the derived module has processes, it must include an sc-has-process–declaration statement in its body.

sc-has-process-declaraton ::=
 SC_HAS_PROCESS (identifier **) ;**

Module identifier must denote the name of the surrounding module.

Modules in which the constructor method is defined using the **SC_CTOR** macro must not contain any sc has process statement.

Examples:

```
// Deriving a module:
     SC_MODULE( BaseModule ) {
       sc_in< bool > reset;
       sc_in_clk clock;
       BaseModule ( const sc_module_name& name_  )
         : sc_module( name_ )
         {}
     };


     class DerivedModule : public BaseModule {
       void newProcess();
       SC_HAS_PROCESS( DerivedModule );
       DerivedModule( sc_module_name name_ )
         : BaseModule( name_ ) {
         SC_CTHREAD( newProcess, clock.pos() );
         watching(reset.delayed() == true);
       }
     };
```

4 Datatypes

There are two kinds of types: fundamental types and compound types. Types describe objects, references, or functions.
Alignment requirements mentioned in ISOC++ Section 3.9 are not relevant for synthesis. Likewise, the use of *sizeof* and *memcpy* to copy objects is not supported for synthesis.

4.1 Fundamental Types

Fundamental types are comprised of integral types and floating-point types. With the exception of *wchar_t*, all integral types are supported for synthesis. Floating-point numbers are not supported for synthesis.

4.1.1 Integral Types

The following integral types are supported for synthesis:

- bool
- unsigned char, signed char, char
- unsigned short, signed short
- unsigned int, signed int
- unsigned long, signed long
- unsigned long long, signed long long

All the integral types except signed/unsigned long long are part of ISOC++ (Section 3.9.1). The long long types are part of the more recent ISOC (Section 6.2.5) and are already supported in most C++ compilers.

The integral type *wchar_t* is not supported. For synthesis, the plain *char* (neither signed nor unsigned) are treated as *signed char*. This is a **synthesis refinement** since ISOC++ specifies that whether a plain *char* is treated as signed or unsigned is implementation dependent (ISOC++ Section 3.9.1).

4.1.1.1 Representation

Integral types are represented using 2's complement. This is a **synthesis refinement** since ISOC++ leaves the representation open (ISOC++ Section 3.9.1, Paragraph 7).

The representation and the bitwidth of an integral type determine its numerical range and its overflow behavior. Synthesis may choose alternative representations for internal objects (not part of the interface of the design) of C integral types provided the I/O behavior of the design is unchanged.

4.1.1.2 Bit sizes

The bit sizes for the different integral types adhere to the requirements set forth by the ISOC++ and ISOC standards and are implementation dependent. However, synthesis tools are required to support the size settings for the platforms for which the synthesis tool is supported in order to guarantee consistency between simulation and synthesis on the working platform.

The table below provides an overview of the ISOC++ requirements and bits sizes for integral types used on most compilers for popular computer platforms. Note that ISOC++ Section 3.9.1 requires the signed and unsigned (and plain in the case of characters) versions of integral types to have the same storage. ISOC++ Section 3.9.1 also constrains the relative sizes of the different integral types. The minimum sizes are derived from the minimum required numerical limits as specified by ISOC Section 5.2.4.2.

Integral Type	**Relative Requirement**	**Minimum Req (bits)**	**Most popular current compilers (bits)**
(un)signed char, char		8	8
(un)signed short	*bits(short) ≥ bits(char)*	16	16
(un)signed int	*bits(int) ≥ bits(short)*	16	32
(un)signed long	*bits(long) ≥ bits(int)*	32	32/64
(un)signed long long	*bits(long long) ≥ bits(long)*	64	64

The maximum and minimum values that an integral type holds is specified in a specialization of the standard template numeric_limits described in ISOC++ Section 18.2 and are specified in the header <climits>. The <climits> header contains the macros for the maximum and minimum values for integral types. For example, INT_MIN and INT_MAX define the numerical limits for signed int. The ISOC standard specifies minimum requirements for the maximum and minimum values. For instance, INT_MIN should be less or equal than -(2^{15}- 1) (most popular current compilers use -2^{31}) and INT_MAX should be greater or equal than (2^{15}-1) (most popular current compilers use 2^{31}-1).

4.1.2 Type Conversions

ISOC++ defines two kinds of conversions between integral types that are applied in the evaluation of expressions: integral promotions, and usual arithmetic conversions. An example of an integral promotion is when a short is promoted to an int in the unary minus expression "-a" (variable a is of type short). An example of a usual arithmetic conversion is when operand of type short is converted to long long in the expression "a+b" where "a" is of type short and "b" is of type long long. In that case "a" is first promoted to type int (integral promotion that is performed as part of the usual arithmetic conversion) and then converted to long long.

4.1.2.1 Integral Promotions

The rules for integral promotions (ISOC++ Section 4.5) are defined as follows:

- An rvalue of a type that can be represented by an int is converted to int. For the bit sized specified in Section 4.1.1.2, this rule implies that rvalues of types char, signed char, unsigned char, signed short and unsigned short are converted to an rvalue of type int.
- An rvalue of an enumeration type is converted to the first of the following types that can represent all the values of its underlying type: int, unsigned int, long, unsigned long, long long, unsigned long long.
- An rvalue of an integral bit-field type is converted to the first of either int or unsigned int that can represent all the values of the bit-field. If the bit-field is larger yet, no

integral promotion is applied. Bit-fields of enumeration type are treated as any other value of that type.
- An rvalue of type bool can be converted to int, with false and true becoming 0 and 1 respectively

4.1.2.2 Usual Arithmetic Conversions

Usual arithmetic conversions are defined by the C++ language to yield a common type for many binary operators that expect operands of arithmetic or enumeration type. The conversion rules specified by ISOC++ apply to synthesis with the exception of the conversion rules related to builtin floating point types since they are not supported for synthesis.
Usual arithmetic conversions (ISOC++ Section 5) are defined as follows:

- First, Integral promotions are performed on both operands.
- Then, If either operand is unsigned long long, the other operand is converted to unsigned long long
- Otherwise, If either operand is long long, the other operand is converted to long long
- Otherwise, if either operand is unsigned long, the other operand is converted to unsigned long
- Otherwise, if either operand is long, the other operand is converted to long
- Otherwise, if either operand is unsigned, the other operand is converted to usigned int
- Otherwise, no convertion as both operands must be int.

The rules outlined above differ from the ISOC++ definition in that floating-point operands are not considered (as they are not supported for synthesis) and the rules for long long and unsigned long long were added (they are not part of the ISOC++ standard yet).

4.1.3 Operators

In this section we will use the following functions to explain the conversions that take place when integer expressions are evaluated in C++:

- function *int_prom*(type t): returns the type resulting from integer promoting type t
- function *arith_conv*(type t_1, type t_2): returns the type resulting from applying the usual arithmetic conversion to the pair of types t_1 and t_2.
- function *type*(variable v): returns the type of variable v
- function *size*(type t) : returns the size in bits for type t.
- function *cast*(type t, value v): returns the value resulting from casting value v with type t.

4.1.3.1 Operators : a<<b (shift to left of 'b' bits for the value of 'a'), a>>b (shift to right of 'b' bits for the value of 'a')

For both right and left shifts, the type of the result is that of the integral promoted left operand(ISOC++ Section 5.8). The behavior is undefined if b is negative or greater than or equal to the length of type(result).

Note: compilers are not consistent on the implement the undefined behavior. For example, shifts on long long (64-bits) may be implemented on 32-bit architectures in a number of ways with the available machine instructions yielding non-obvious results for shifts outside the defined range.

For right shifts, if the first operand has a signed type, the sign bit is shifted in. This is a **synthesis refinement** on ISOC++ since the standard leaves the behavior implementation-defined when the first operand is negative.

> prom_type = *int_prom*(*type*(a))
> A = *cast*(a, prom_type)
> *result*(a << b) = *cast*(A << b, prom_type)   when 0 ≤ b < *size*(prom_type)
> *result*(a >> b) = *cast*(A >> b, prom_type)   when 0 ≤ b < *size*(prom_type)

Note: VisualC 6 (microsoft/windows/intel) is not compliant with ISOC++ since it takes the result type to be *arith_conv*(*type*(a), *type*(b)).

4.1.3.2 Operator: +a (unary plus)

The type of the result is that of the integral promoted operand (ISOC++ Section 5.3.1):

> prom_type = *int_prom*(*type*(a))
> result(+a) = *cast*(a, prom_type)

-

4.1.3.3 Operator: -a (unary minus)

The type of the result is that of the integral promoted operand (ISOC++ Section 5.3.1):

> prom_type = *int_prom*(*type*(a))
> result(-a) = cast(-*cast*(a, prom_type), prom_type)

4.1.3.4 Operators: a+b (addition of a and b) and a-b (subtract b from a)

The usual arithmetic conversions are performed for the operands. The type of the result is given by the arithmetic conversion:

> prom_type = *arith_conv*(*type*(a), *type*(b))
> A = *cast*(a, prom_type), B = *cast*(b, prom_type)
> result(a+b) = *cast*(A + B, prom_type)
> result(a-b) = *cast*(A – B, prom_type)

4.1.3.5 Operator : a*b (product a times b)

The usual arithmetic conversions are performed for the operands. The type of the result is given by the arithmetic conversion (ISOC++ Section 5.6):

> prom_type = *arith_conv*(*type*(a), *type*(b))
> A = *cast*(a, prom_type), B = *cast*(b, prom_type)
> result(a*b) = *cast*(A * B, prom_type)

4.1.3.6 Operators: a/b (division: a divided by b), a%b (remainder)

The usual arithmetic conversions are performed for the operands. The type of the result is given by the arithmetic conversion:

prom_type = *arith_conv*(*type*(a), *type*(b))
A = *cast*(a, prom_type), B = *cast*(b, prom_type)
result(a/b) = *cast*((b==0) ? (DONT_CARE) : *trunc_zero*(A / B), prom_type)
result(a%b) = *cast*((b==0) ? (DONT_CARE) : A - *trunc_zero*(A / B) * B, prom_type)

ISOC++ specifies that the result is undefined when the second operand is zero. For synthesis such behavior could be used as a don't care value that may be exploited to optimize the hardware. The result of the division is truncated towards zero which implies -a/b ==a/-b ==-(a/b). Truncation towards zero is specified in ISOC (Section 6.5.5) but is a **synthesis refinement** with respect to ISOC++ (Section 5.6).

4.1.3.7 Operator ~a (bitwise complement of a)

The type of the result is that of the integral promoted operand (ISOC++ Section 5.3.1):

prom_type = *int_prom*(*type*(a))
result(~a) = *cast*(~*cast*(a, prom_type), prom_type)

4.1.3.8 Operators a&b (bitwise AND), a|b (bitwise inclusive OR), a^b (bitwise exclusive OR)

The usual arithmetic conversions are performed for the operands. The type of the result is given by the arithmetic conversion (ISOC++ Sections 5.11, 5.12, 5.13):

prom_type = *arith_conv*(*type*(a), *type*(b))
A = *cast*(a, prom_type), B = *cast*(b, prom_type)
result(a OP b) = *cast*(A OP B, prom_type)

-

4.1.3.9 Relational and equality operators <, >, <=, >=, ==, !=

The usual arithmetic conversions are performed for the operands. The type of the result is bool (ISOC++ Section 5.9):

prom_type = *arith_conv*(*type*(a),*type*(b))
A = *cast*(a, prom_type), B = *cast*(b, prom_type)
result(a OP b) = *cast(* A OP B, bool)

4.1.3.10 Conditional operator a?b:c (for cases where b and c are integral or enumeration types)

The usual arithmetic conversions are performed for the operands. The type of the result is given by the arithmetic conversion:

prom_type = *arith_conv*(*type*(b), *type*(c))
B = *cast*(b, prom_type), C = *cast*(c, prom_type)
result(a ? b : c) = *cast*(a ? B : C, prom_type)

Only one of b or c is evaluated. All side effects of the first expression except for destruction of temporaries happen before the second or third expression are evaluated (ISOC++ Section 5.16).

4.1.3.11 Increment and decrement operators a++, ++a, a--, --a, --a

The type of result of the pre or post increment/decrement operation is the type of the operand. For integer operands excluding bool the increment and decrement operators are defined as follows:

result(a++) = a,	side_effect(a++): a = *cast*(a+1, *type*(a))
result(++a) = *cast*(a+1, *type*(a)),	side_effect(++a): a = *cast*(a+1, *type*(a))
result(a--) = a,	side_effect(a--): a = *cast*(a-1, *type*(a))
result(--a) = *cast*(a-1, *type*(a))	side_effect(--a): a = *cast*(a-1, *type*(a))

The increment and decrement operators are not supported for operands of type bool. This is a **synthesis refinement** since ISOC++ does allow incrementing of operands of type bool (though ISOC++ deprecates its use).
See ISOC++ Sections 5.2.6 and 5.3.2 for further details.

4.1.4 Floating Point Types

Floating-point types are not supported.

4.2 Compound Types

Compound types in C++ are described in ISOC++ Section 3.9.2. They can be recursively constructed as follows:

- arrays of objects of a given type
- classes (class, struct, union)
- functions
- pointers to void or objects or functions (including static members of classes) of a given type
- references to objects or functions of a given type
- enumerations
- pointers to nonstatic class members

with some restrictions given in ISOC++ Sections 8.3.1, 8.3.2, 8.3.4 and 8.3.5.

4.2.1 Arrays

The element type of an array must be any of the types which are permitted by the ISO C++ standard as element type, excluding pointers, and which are supported for synthesis, or any SystemC data type which is supported for synthesis, and which is conform to the requirements on element types stated in the ISO C++ standard. In addition, it is also legal to declare arrays of signals and arrays of ports in any place where SystemC permits the declaration of signals and ports. Any declaration of an array must include the specification of its bound, either explicitly, if no initializer is specified, or as implication from the initializer, if such is specified.

4.2.2 Pointers

Pointers that are statically determinable are supported for synthesis. Otherwise, they are not supported. Statically determinable implies that during compilation, synthesis is able to determine the actual object whose address is contained by the pointer. If the pointer points to an array, the size of the array must also be statically determinable. Pointer arithmetic is not allowed. Testing that a pointer is zero is not allowed.The use of the pointer value as data is not allowed. For example, hashing on a pointer is not supported for synthesis.

4.2.3 References

Supported.

4.2.4 Enumerations

Enumerations are distinct types that comprise a set of named constant values.
Enumerations can be promoted to the signed and unsigned versions of int, long, and long long.

Enumerations are supported.

4.2.5 Pointers to Nonstatic Class Members

Supported provided that the context of the use permits the actual object that is the target of the pointer to be determined at compile time.

4.3 CV-qualifiers

Fundamental and compound types are cvunqualified types. Each cvunqualified complete or incomplete object type has three corresponding cvqualified versions of its type: a const-qualified version, a volatile-qualified version, and a const-volatile qualified version.

The cvqualified or cvunqualified versions of a type are distinct types; however, they shall have the same representation for synthesis.

4.4 System C Datatypes

In addition to supporting the C++ types as described in Section 10, System-C provides a number of datatypes that are useful for hardware design. These datatypes are implemented in C++ classes.

The datatypes that are supported for synthesis are:

- Numerical
- Integer Types
    - sc_int: finite precision signed (**conditioned on better definition**)
    - sc_uint: finite precision unsigned (**conditioned on better definition**)
    - sc_bigint: arbitrary precision signed
    - sc_biguint: arbitrary precision unsigned
- Fixed-point Types
    - sc_fixed: arbitrary precision signed
    - sc_ufixed: arbitrary precision unsigned
- Bit Vector Type: sc_bv

- Logic (4-valued):
    - sc_logic
    - sc_lv

All datatypes supported for synthesis have vector length/precision that is specified by template parameters. Thus their vector length/precision is statically determinable during compilation.

Editorial Note: there are a number of issues or/and inconsistencies between the datatypes that appear to be the product of an implementation that comes from different sources and the absence of a well defined LRM. These issues are flagged in bold face throughout this document. The finite precision integers sc_int/sc_uint in particular have a number of issues that should be resolved before it could be considered for synthesis.

4.4.1 Integer Types

The SystemC datatype package defines integer types that allow the selection of any bitwidth. Both signed an unsigned versions are available:

- sc_bigint<W>: arbitrary precision signed integer
- sc_biguint<W>: arbitrary precision unsigned integer
- sc_int<W>: finite precision signed integer (W ≤ 64)
- sc_uint<W>: finite precision unsigned integer (W ≤ 64)

The finite precision versions are available for more efficient simulation but are limited to 64 bits. Otherwise they are semantically identical to the arbitrary width integer datatypes (**note: not true today, filed a bug report for inconsistency when mixing signed and unsigned operands**). The compile flag _32BIT_ is not supported as it is not even mentioned in the LRM.
Arbitrary precision datatypes do have an implementation limit that may be changed with the compiler flag MAX_NBITS (**note: LRM does not mention this limit**). It is assumed that this limit is set higher than any bit-width of any operand so that synthesis need not consider the effects due to the implementation limit.
Signed integers types are stored in 2’s complement form and all arithmetic is done in 2’s complement.

The integer types support the following operators:

Op Category	**Operators/Methods**					
Arithmetic	+	-	*	/	%	
Assign	+=	-=	*=	/=	%=	
Unary	+	-				
Auto Incr/dec	++ prefix	++ postfix	-- prefix	-- postfix		
Bitwise	&	|	^			
Assign	&=	|=	^=			
Unary	~					
Relational	==	!=	<	<=	>	>=
Shift	>>	<<	>>=	<<=		
Bit Select	[x]					
Part Select	(i,j)					
Concatenation	(,)					
Conv to C integral	to_int	to_long	to_int64	to_uint	to_uint64	to_ulong
Assignment	=					

Arithmetic, shift and bitwise operations generate intermediate values that do not loose precision other than when they reach the precision limits (64 bits) in the case of limited precision integer types.
Two operand arithmetic, bitwise and relational operators take any of the integral C++ types, in addition to SystemC integer types, as one of their operators (second argument for assign version of the operators).

The table below shows additional methods that are supported for synthesis. The table also shows alternatives ways to get the same functionality.

Methods	**Alternatives**
iszero	x == 0
sign	x < 0
bit	x[i]
range	x(i,j)
reverse	x = x(0, W-1) (**??**see part select)
test	x[i]
set	x[i] = 1
clear	x[i] = 0
invert	x[i] = !x[i]
length	(template parameter)

4.4.1.1 Arithmetic Operators

Subtraction and unary minus always generates a signed value. All other operators generate signed values if at least one of the operand(s) is signed, otherwise they generate an unsigned value.
Arithmetic operations with two operands can take any of the integral C++ types as one of the arguments (second argument for arithmetic assign operations).

4.4.1.2 Bitwise Operators

The unary operator ~ is the one's complement operator. The return value for ~x is one's complement of x which is equal to (–x-1). As the representation is 2's complement this is equivalent to complementing every bit on a signed representation of the value. For instance ~((sc_biguint<8>)128) = ~((sc_bigint<9>)128) = -129.
The binary operators &, | and ^ compute the bitwise and or and xor operations. If either of the operands is signed, and the other operand is unsigned, the unsigned operand is first represented as a signed operand (by adding a bit of precision). If the bit-widths of the two operands are not the same, the shorter operand is extended to match the length of the other operand. The type of the result is unsigned if both operands are unsigned, otherwise it is signed.

4.4.1.3 Relational Operators

The relational operators compare the two operands as in C++ and return a value of type bool. The comparison is done arithmetically.

4.4.1.4 Shift Operators

The shift operators take a C++ int type value as their second operand (shift value). Operator << and operator >> define arithmetic shifts, not bitwise shifts, i.e., no bits are lost and proper sign extension is done.
The shift value should be nonnegative (**this is inconsistent with fixed-point datatypes where negative shift values are implemented as shifts in the opposite direction**). For sc_bigint/sc_biguint, a negative shift is equivalent to a zero shift. For sc_int/sc_uint, the shift is effectively implemented as a shift of a 64-bit C integer. Both negative shifts and shifts ≥ 64 are governed by the rules of integer shifts in C and are implementation dependent (**inconsistency between sc_bigint/sc_biguint and sc_int/sc_uint**).

Note: the second operand should be of unsigned type to guarantee that the inferred hardware is minimal. Otherwise, whether minimal hadware is inferred depends on the analysis capabilities of the synthesis tool for proving that the shift value is never negative or never positive.

4.4.1.5 Assignment Operator

All defined assignment operators are supported.

4.4.1.6 Bit Select Operator

The bit select operator[i] allows the selection of a bit of a variable either as an rvalue or an lvalue, e.g.,

```
      x[3] = y[2];
```

The use of the bit select operator on a temporary (unless it is explicitly cast to sc_int/sc_uint or sc_bigint/sc_biguint) is deprecated given that it is not consistently supported in SystemC. For example, sc_int/sc_uint does not support the bit select operator on a temporary, sc_bigint/sc_biguint supports it for arithmetic operators (e.g., (a*b)[7]) but not for some operators (e.g., concatenation).
The index may be outside the range [0, W-1] for sc_bigint/sc_biguint, but not for sc_int/sc_uint (**Not clear from the LRM whether this is intended or not**, **need to resolve inconsistency**). If the index ≥ W then 0 is returned for unsigned and the MSB bit is returned for signed numbers. If the index is negative, then the LSB bit is returned.

4.4.1.7 Part Select Operator

The part select or range operator (i,j) allows the selection of a bit slice of the variable either as an rvalue or an lvalue, e.g.,

```
      x(5,3) = y(4,2);
```

The use of the range operator on a temporary (unless it is explicitly cast to sc_int/sc_uint or sc_bigint/sc_biguint) is deprecated given that it is not consistently supported in SystemC. For example, sc_int/sc_uint does not support the range operator on a temporary, sc_bigint/sc_biguint supports it for arithmetic operators (e.g., (a*b)(7,5)) but not for some operators (e.g., concatenation).

A reverse range may be used, e.g., x(0,3) (supported for sc_bigint/sc_biguint but not supported for sc_int/sc_uint (runtime exception), **not specified in LRM, not clear whether it is bug or intended behavior, need to resolve inconsistency**).

The range can exceed the range of the variable for sc_bigint/sc_biguint but not for sc_int/sc_uint (runtime exception). (**Not clear from the LRM whether this is intended or**

not, **need to resolve inconsistency**). If either range value is larger than the MSB index (W-1), then the bits with index ≥ W get either 0 for unsigned numbers or the MSB bit for signed numbers. If either range value is negative, it is changed to zero.

The use of dynamic values for the range may be restricted for synthesis as the length of the range needs to be statically determinable.

4.4.1.8 Concatenation Operation

The concatenation operation (op1,op2) may be used an rvalue or an lvalue, e.g.,

```
      (x, y) = (z, w);
```

Because of the difference in return types for operators for sc_bigint/sc_biguint and sc_int/sc_uint, using expressions (unless they are cast) may give different results for arbitrary precision integers than for finite precision integers. Using uncast expressions other than concatenation, bit select and part select as arguments of the concatenate operation is deprecated.

4.4.1.9 Unsupported Methods

The following methods are not supported for synthesis:

- Reduce methods since they are not supported for arbitrary precision datatypes. The reduce methods are and_reduce, or_reduce, xor_reduce, nand_reduce, nor_reduce and xnor_reduce. **Otherwise they should probably be supported (?). These are being added in 2.1. I think we should say they are supported.**
- Underlying classes such as sc_signed, sc_unsigned, sc_int_base are not directly synthesizable. They are part of the SystemC implementation for the synthesizable types.
- The explicit conversion to_double() is not supported as the C++ type double is not supported for synthesis.
- Methods set_packed_rep, get_packed_rep

4.4.2 Fixed-point Types

The SystemC datatype package supports fixed-point types with arbitrary precision and with a variety of quantization and overflow modes. The types support both signed and unsigned fixed-point datatypes:

- sc_fixed<wl, iwl, qmode, o_mode, n_bits>
- sc_ufixed<wl, iwl, qmode, o_mode, n_bits>

The first two template parameters together determine the precision of the integral and fractional parts of the fixed-point number. The first template argument wl is the overall bitwidth of the fixed-point number. The second template argument iwl
is the position of the binary point relative to the most significant bit. If iwl is positive it is the bitwidth of the integral part. The ranges are as follows:

- sc_fixed: $[-2^{(iwl-1)}, 2^{(iwl-1)} - 2^{-fwl}]$
- sc_ufixed: $[0, 2^{iwl} - 2^{-fwl}]$

where fwl = wl-iwl. The rightmost bit is the LSB(0), and the leftmost bit is the MSB(wl-1).

The third template argument qmode determines the quantization mode used. The fourth and fifth template argument determine the overflow mode. These modes are summarized in the tables below.

Overflow Mode	Parameters
Wrap-around Basic (*default*)	o_mode = SC_WRAP, n_bits = 0
Saturation	o_mode = SC_SAT
Symmetrical Saturation	o_mode = SC_SAT_SYM
Saturation to Zero	o_mode = SC_SAT_ZERO
Wrap-around Advanced	o_mode = SC_WRAP, n_bits > 0
Sign Magnitude Wrap-Around	o_mode = SC_WRAP_SM, n_bits ≥ 0

Quantization Mode	Parameter
Truncation (*default*)	q_mode = SC_TRN
Rounding to plus Infinity	q_mode = SC_RND
Truncation to zero	q_mode = SC_TRN_ZERO
Rounding to zero	q_mode = SC_RND_ZERO
Rounding to minus infinity	q_mode = SC_RND_MIN_INF
Rounding to infinity	q_mode = SC_RND_INF
Convergent rounding	q_mode = SC_RND_CONV

Quantization and overflow are performed only when loss of precision is required: casts and assignments. Intermediary values do not require loss of precision. The exception is the divide operator that would potentially require an infinite number of bits. An internal implementation limit (compiler flag) SC_FIXDIV_WL bounds the number of bits that are computed for division. Another implementation limit (compiler flag) is SC_FXMAX_WL. It is assumed that precision is set high enough so that they don't change the behavior of the design. Synthesis will assume that these limits are not present.

All quantization/overflow modes are supported for synthesis. **Should any be excluded from minimum subset?**

Op Category	Operators/Methods					
Arithmetic	+	-	*	/		
Assign	+=	-=	*=	/=		
Unary	+	-				
Auto Incr/dec	++ prefix	++ postfix	-- prefix	-- postfix		
Bitwise	&	|	^			
Assign	&=	|=	^=			
Unary	~					
Relational	==	!=	<	<=	>	>=
Shift	>>	<<	>>=	<<=		
Bit Select	[x]					
Part Select	(i,j)					
Conv to C integral	to_int	to_long	to_int64	to_uint	to_uint64	to_ulong
Assignment	=					

The table below shows additional methods that are supported for synthesis. The table also shows alternatives ways to get the same functionality.

Methods	Alternatives
is_zero	x == 0
is_neg	x < 0
b_not	~
b_and	&
b_or	|
b_xor	^
lshift	<<
rshift	>>
neg	unary -
bit	x[i]
range	x(i,j)
wl	template argument 1
iwl	template argument 2
o_mode	template argument 3
q_mode	template argument 4
n_bits	template argument 5

4.4.2.1 Arithmetic Operations

Subtraction and unary minus always generates a signed value. All other operators generate an signed values if at least one of the operand(s) is signed, otherwise they generate an unsigned value.

The autoincrement/autodecrement operators x++, ++x, x-- and --x have the semantics as follows where T_x is the type of variable x:

Operator	Equivalent Behavior
x++	T_x t = x; x += 1; return t;
++x	x += 1; return reference to x;
x--	T_x t = x; x -= 1; return t;
--x	x -= 1; return reference to x;

The division (/) operator is problematic for synthesis because the required output precision can not be deduced from the precision of the its operands. Unless the operation is immediately cast or assigned the required precision may be hard to deduce resulting in an unnecessarily large hardware divider. The division assign (/=) operator is well defined because the target precision is given by the first operand.

4.4.2.2 Bitwise Operators

The unary ~ operator complements the bits of the mantissa (it differs from the SystemC integer types: ~((sc_ufixed<8,8>) 128) != ~((sc_biguint<8> 128)).
The binary operations &, | and ^ compute the bitwise and, or and xor operation respectively. Mixing of signed an unsigned operators is not allowed (a difference compared with SystemC integer types). For binary operations, the two operands are aligned by the binary point and the operands extended so that they have the same word and fractional length before the operation is performed.

4.4.2.3 Relational Operators

The relational operators compare the two operands as in C++ and return a value of type bool. The comparison is done arithmetically.

4.4.2.4 Shift Operators

The shift operators take an C++ int type value as their second operand (shift value). If the shift value is negative the first operand is shifted in the opposite direction. Operator << and operator >> define arithmetic shifts, not bitwise shifts, i.e., no bits are lost and proper sign extension is done.
Note: the second operand should be of unsigned type to guarantee that the inferred shifter is unidirectional. Otherwise, whether a bidirectional or unidirectional shiftter is inferred depends on the analysis capabilities of the synthesis tool for proving that the shift value is never negative or never positive.

4.4.2.5 Bit Select Operator

The bit select operator[i] allows the selection of a bit of a variable either as an rvalue or an lvalue, e.g.,

```
x[3] = y[2];
```

The use of the bit select operator on an expression (unless it is explicitly cast to sc_fixed/sc_ufixed) is deprecated given that it is not consistently supported in SystemC (see Section 4.4.1.6)
The index has to be in the range [0, W-1] (runtime exception otherwise).

4.4.2.6 Part Select Operator

The part select or range operator (i,j) allows the selection of a bit slice of the variable either as an rvalue or an lvalue, e.g.,

```
x(5,3) = y(4,2);
```

The use of the range operator on a temporary (unless it is explicitly cast to sc_fixed/sc_ufixed) is not supported in SystemC. The result of a part select can not be directly assigned to a fixed point variable, but it can be assigned to a range:

```
x = x(0, W-1);  // not supported
x(W-1,0) = x(0,W-1);  // OK
```

The range may be reversed. The range arguments need to be in the range [0, W-1], otherwise a runtime exception is generated (**this behavior is inconsistent with sc_bigint/sc_biguint**). Synthesis may impose additional requirements that the length of the range be statically determinable.

4.4.2.7 Assignment Operators

All defined assignment operators are supported.

4.4.2.8 Unsupported Methods/Options

The following options and methods are not synthesizable:

- Methods intended for internal use: overflow_flag, quantization_flag, type_params, get_bit, set_bit, get_slice, set_slice, get_rep, lock_observer, unlock_observer, observer_read, value and is_normal are not supported.
- Methods cast, cast_switch and observer are not supported.
- The SC_OFF option to turn casting off.
- The semantics due to limits to precision given by SC_FXMAX_WL, SC_FIXDIV_WL and SC_FXCTE_WL since that semantics requires normalization (which should be avoided in hardware implementations of fixed point computation).
- Explicit conversion to C++ floating point types to_double and to_float since neither type is supported for synthesis.

4.4.2.9 Non Synthesizable Classes

The following System C classes related to fixed point datatypes are not synthesizable:

- Fixed Point related:
  - Limited Precision fixed-point types used for faster simulation
    - sc_fixed_fast
    - sc_ufixed_fast

    They use double and have a limited precision of 53 bits. They are not bit accurate with sc_fixed/sc_ufixed since normalization will determine which 53 bits are kept.
  - Unconstrained types and related context classes
    - sc_fix
    - sc_ufix
    - sc_fix_fast
    - sc_ufix_fast
    - sc_fxcast_context
    - sc_fxcast_swith
  - Arbitrary precision value
    - sc_fxval
    - sc_fxval_fast
  - Observer types
    - sc_fxnum_observer
    - sc_fxnum_fast_observer
    - sc_fxval_observer
    - sc_fxval_fast_observer

4.4.3 Bit Vectors

The arbitrary width bit-vector type is sc_bv<W>. This type has two values ‘0’ and ‘1’ which interpreted as false and true respectively. Single bit values are represented using the C++ type bool. The rightmost bit is the LSB(0), and the leftmost bit is the MSB(width-1).

The operations that are supported for synthesis are listed in the table below.

Op Category	**Operators/Methods**					
Bitwise	&	|	^			
Assign	&=	|=	^=			
Unary	~					
Relational	==	!=				
Shift/Rotate	>>	<<	>>=	<<=	lrotate(i)	rrotate(i)

Bit Select	[x]					
Part Select	(i,j)					
Concatenation	(,)					
Conv to C integral	to_int	to_long	to_int64	to_uint	to_uint64	to_ulong
Assignment	=					
Reduce	and_reduce	or_reduce	xor_reduce	nand_reduce	nor_reduce	xnor_reduce

There is no arithmetic defined for bit vectors and MSB zero padding is used to extend the vector when it is required to match the length of a second operand (in binary bitwise operations) or to match the length of the target.

The table below shows additional methods that are supported for synthesis. The table also shows alternatives ways to get the same functionality.

Methods	**Alternatives**
length	template parameter
bit, get_bit	x[i]
set_bit	x[i]
range	x(i,j)
reverse	x = x(0, W-1)
b_not	~

4.4.3.1 Bitwise Operators

The unary operator ~ complements every bit of the vector.
The binary operators &, | and ^ compute the bitwise and or and xor operations. If the bit-widths of the two operands are not the same, the shorter operand is extended by padding zeros to match the length of the other operand.

4.4.3.2 Relational Operators

The relational operators == and != return a bool to indicate whether the two vectors are equal or not equal respectively. Two vectors of different length are not equal.

4.4.3.3 Shift Operators and Rotate Methods

The shift operators take a C++ int type value as their second operand (shift value). A negative shift value is not allowed (runtime exception). The result of a shift is a vector of the same length (shifts are not arithmetic, bits are lost).

The rotate methods lrotate and rrotate rotate left and right respectively by the amount given by the integer argument.

4.4.3.4 Bit Select Operator

The bit select operator[i] allows the selection of a bit of a variable either as an rvalue or an lvalue, e.g.,

```
    x[3] = y[2];
```

An index argument needs to be in the range [0, W-1], otherwise a runtime exception is generated.

4.4.3.5 Part Select Operator

The part select or range operator (i,j) allows the selection of a bit slice of the variable either as an rvalue or an lvalue, e.g.,

```
      x(5,3) = y(4,2);
```

The range may be reversed. The range arguments need to be in the range [0, W-1], otherwise a runtime exception is generated.

4.4.3.6 Concatenation Operator

The concatenation operation (op1,op2) may be used an rvalue or an lvalue, e.g.,

```
      (x, y) = (z, w);
```

4.4.3.7 Assignment Operator

All defined assignment operators are supported.

4.4.3.8 Reduce Methods

The reduce operators and_reduce, or_reduce, xor_reduce, nand_reduce, nor_reduce and xnor_reduce return a result of type bool by applying the corresponding logical operation to all bits.

4.4.4 Logic Types

4.4.4.1 Logic Type

The logic type is sc_logic and has four values: ‘0’, ‘1’, ‘X’ and ‘Z’ interpreted as false, true, unknown and high_impedence respectively.

- Bitwise &(and) |(or) ^(xor) ~(not)
- Assignment = &= |= ^=
- Equality == !=

4.4.4.2 Unsupported Logic Constant

The use of logic constant has restrictions for synthesis.

- The unknown logic constant (sc_logic<W>("X")) is not supported for synthesis. Exceptionally a tool may use the unknown value assignment to specify an explicit don't-care condition for the logic synthesis. This depends on the optimization capability of the synthesis tool.

Example :

```
 if (x == 0x00)
      y = sc_logic<1>("0");
   else if(x == 0x01)
      y = sc_logic<1>("1");
   else if (x == 0x10)
      y = sc_logic<1>("0");
   else
      y = sc_logic<1>("X"); //don't-care condition
```

- The high_impedence logic constant (sc_logic<W>("Z")) is synthesizable if and only if it appears in an expression assigned to a port variable directly. This expression should not include conditional expressions that contain equality operators for logic constant.

4.4.4.3 Unsupported Methods

The is_01(), to_bool() , value() methods don’t have any meaning for synthesis.
The b_not method is not required as the operator ~ can be used instead.

4.4.4.4 Arbitrary Width Logic Vectors

The arbitrary width logic vector type is sc_lv<W> with each element in the vector being having four types as the logic type sc_logic. The rightmost bit is the LSB(0), and the leftmost bit is the MSB(width-1).

Operations etc should be identical to sc_bv.

5 Declarations

5.1 Declarations

Most declarations are supported as used from C++. Specific restrictions and coding guidelines are listed in the subsequent sections and chapters.

```
declaration-seq ::=
      declaration
    | declaration-seq declaration

declaration ::=
      block-declaration
    | function-definition
    | template-declaration
    | explicit-instantiation
    | explicit-specification
    | ~~linkage-specification~~
    | namespace-definition
    | sc-process-definition

block-declaration ::=
      simple-declaration
    ~~| asm-definition~~
    | namespace-alias-definition
    | using-declaration
    | using-directive

simple-declaration ::=
      [ decl-specifiier-seq ] [ init-declarator-list ];
```

Deprecated items

Embedded assembler routines are not supported.

5.1.1 Specifiers

```
decl-specifier ::=
      storage-class-specifier
    | type-specifier
    | function-specifier
    | friend
    | typedef

decl-specifier-seq ::=
    [ decl-specifier-seq ] decl-specifier
```

5.1.1.1 Storage class specifiers

Supported.

```
storage-class-specifier ::=
      auto
    | register
    | static
```

| **extern**
| **mutable**

The specifiers **auto**, **register**, **mutuable**, and **extern** are ignored for synthesis.

The use of **static** is restricted (refer to section 12.1.4.2).

5.1.1.2 Function specifiers

Supported.

```
function-specifier ::=
                inline
              | virtual
              | explicit
```

A synthesis tool will try to replace each call to a function that is declared using the **inline** specifier by the respective function body.

5.1.1.3 The `typedef` specifier

Supported.

```
typedef-name ::=
              identifier
```

5.1.1.4 The `friend` specifier

(*No difference to C++.*)

5.1.1.5 Type specifiers

For restrictions on the use of types refer to the respective subsections.

```
type-specifiier ::=
                simple-type-specifier
              | class-specifier
              | enum-specifier
              | elaborated-type-specifier
              | cv-qualifier
              | sc-type-specifier
              | sc-module-specifier
```

5.1.1.6 The cv-qualifiers

Supported.

The **const** qualifier is supported with the usual meaning. The use of **volatile** is ignored for synthesis.

5.1.1.7 Simple type specifiers

Restricted support.

```
simple-type-specifier ::=
                [ :: ] [ nested-name-specifier ] type-name
```

| [**::**] nested-name-specifier **template** template-id
| **char**
~~| **wchar_t**~~
| **bool**
| **short**
| **int**
| **long**
| **signed**
| **unsigned**
~~| **float**~~
~~| **double**~~
| **void**

type-name ::=
  class-name
| enum-name
| typedef-name

Deprecated items

Types **wchar_t**, **float** and **double** are not supported.

5.1.1.8 Elaborated type specifiers

Supported.

elaborated-type-specifier ::=
  class-key [**::**] [nested-name-specifier] identifier
| **enum** [**::**] [nested-name-specifier] identifier
| **typename** [**::**] nested-name-specifier identifier
| **typename** [**::**] [nested-name-specfier **template**] template-id

5.1.1.9 SystemC type specifiers

Extension.

sc-type-specifier ::=
  sc_int< constant-expression >
| **sc_uint**< constant-expression >
| **sc_bigint**< constant-expression >
| **sc_biguint**< constant-expression >
| sc_logic
| sc_lv< constant-expression >
| sc_bit
| sc_bv< constant-expression >
| sc_fixed< constant-expression **,** constant-expression
  [**,** sc-quantization-mode-specifier] [**,** sc-oveflow-mode-specifier]
  [**,** constant-expression] >
| sc_ufixed< constant-expression **,** constant-expression
  [**,** sc-quantization-mode-specifier] [**,** sc-oveflow-mode-specifier]
  [**,** constant-expression] >

(The biggest problem with fixed point data types seems to be the division operation on fixed point numbers, in particular which precision to use. It may be advisable to deprecate the use of the division operation on fixed point numbers.)

sc-quantization-mode-specifier ::=
  SC_RND
| **SC_RD_ZERO**
| **SC_RND_MIN_INF**
| **SC_RND_INF**
| **SC_RND_CONV**
| **SC_TRN**
| **SC_TRN_ZERO**

sc-overflow-mode-specifier ::=
  SC_SAT
| **SC_SAT_ZERO**
| **SC_SAT_SYM**
| **SC_WRAP**
| **SC_WRAP_SM**

5.1.2 Enumeration declarations

Supported.

enum-name ::=
identifier

enum-specifier ::=
enum [identifier] **{** [enumerator-list] **}**

enumerator-list ::=
  enumerator-definition
| enumerator-list **,** enumerator-definition

enumerator-definition ::=
  enumerator
| enumerator **=** constant-expression

enumerator ::=
identifier

5.1.3 The `asm` declaration

Not supported.

~~asm-definition ::=~~
~~**asm** (string-literal) **;**~~

5.1.4 Linkage specifications

External linkage is not supported.

~~linkage-specification ::=~~
~~**extern** string-literal **{** [declaration-seq] **}**~~
~~| **extern** string-literal declaration~~

5.2 Declarators

Restricted support.

init-declarator-list ::=

  init-declarator
| init-declarator-list **,** init-declarator

init-declarator ::=
declarator [initializer]

declarator ::=
  direct-declarator
| ptr-operator declarator

direct-declarator ::=
  declarator-id
| direct-declarator **(** parameter-declaration-clause **)** [cv-qualifier-seq]
~~[exception- specification]~~
| direct-declarator **[** [constant-expression] **]**
| **(** declarator **)**

ptr-operator ::=
~~***** [cv-qualifier-seq]~~
| **&**
~~| [**::**] nested-name-specifier ***** [cv-qualifier-seq]~~

cv-qualifier-seq ::=
cv-qualifier [cv-qualifier-seq]

cv-qualifier ::=
  const
| **volatile**

declarator-id ::=
  id-expression
| [**::**] [nested-name-specifier] type-name

Deprecated items
Exception handling is not supported.
Apart from few exceptions, pointer operations are generally not supported.

5.2.1 Type names

type-id ::=
type-specifier-seq [abstract-declarator]

type-specifier-seq ::=
type-specifier [type-specifier-seq]

abstract-declarator ::=
  ptr-operator [abstract-declarator]
| direct-abstract-declarator

direct-abstract-declarator ::=
  [direct-abstract-declarator] **(** parameter-declaration-clause **)** [cv-qualifier-seq]
~~[exception-specification]~~
| [direct-abstract-declarator] **[** [constant-expression] **]**
| **(** abstract-declarator **)**

Deprecated items

Exception handling is not supported.

5.2.2 Ambiguity resolution

(*No differences to C++.*)

5.2.3 Parameters

Supported.

parameter-declaration-clause ::=
  [parameter-declaration-list] ~~[**...**]~~
  | parameter-declaration-list ~~**, ...**~~

parameter-declaration-list ::=
  parameter-declaration
  | parameter-declaration-list **,** parameter-declaration

parameter-declaration ::=
  decl-specifier-seq declarator
  | decl-specifier-seq declarator **=** assignment-expression
  | decl-specifier-seq [abstract-declarator]
  | decl-specifier-seq [abstract-declarator] **=** assignment-expression
  | [**const**] sc-signal_declaration **&** identifier

5.2.4 Default arguments

Supported.

5.2.5 Initializers

Restricted support.

initializer ::=
  = initializer-clause
  | **(** expression-list **)**

initializer-clause ::=
  assignment-expression
  | **{** inititializer-list [**,**] **}**
  | **{ }**

initializer-list ::=
  initializer-clause
  | initializer-list **,** initializer-clause

A non-const variable declaration located in the body of a module must not have an initializer and must not be initialised by means of a mem-initializer of the module constructor.

If a variable declaration of class type is located in the body of a module the underlying class definition must not declare or inherit a default constructor or must declare or inherit an empty default constructor.

5.2.5.1 Aggregates

(*May be a problem at least on RT level. Think of a large non-const array which is initialised by means of an aggregate*.)

5.2.5.2 Character arrays

(*Deprecation should be considered.*)

5.2.5.3 References

(No differences to C++.)

6 Expressions

Expression is a sequence of operators and operands that can result in a value can cause side effects (ISOC++ Section 5). The order of evaluation of operands and the order in which side effects take place is unspecified by ISOC++, except when noted. For example the statement:

```
i = x[i++];
```

has a behavior that is not specified in ISOC++.

The *sizeof* operator (ISOC++ Section 5.3.3) is not supported for synthesis. The *new* (ISOC++ Section 5.3.4) and the *delete* operator (ISOC++ Section 5.3.5) are not supported for synthesis.

Casting operators (ISOC++ Sections 5.2.7, 5.2.9, 5.2.10, 5.2.11) are supported within the constraints placed on the use of pointers. The type identification function *typeid* (ISOC++ Section 5.2.8) is not supported.

7 Functions

Functions may be declared/defined as part of a class type declaration, then being denoted as member functions. Functions may also be declared/defined within any namespace including the global namespace (**::**).
Processes within modules are a special kind of function, and must obey special syntactic rules regarding declaration and definition.

7.1 Function definitions

Restricted support.

```
function-definition ::=
                [ decl-specifier-seq ] declarator [ ctor-initializer ] function-body
```
~~| [decl-specifier-seq] declarator function-try-block~~

7.2 Function body

The body of a function is executed on a function call. The body of a function consists of a set of local declarations and a sequence of statements.

```
function-body ::=
              compound-statement
```

The body of a function must adhere to the same rules as the region from where it is invoked. Supported sequential statements are described in section 9. Wait statements may be used in functions called from SC_CTHREAD processes, and may not be used in functions called from SC_METHOD processes.

Example:

```
void
foo( unsigned &x, const unsigned y, const unsigned z ) {
     while ( x < z )
     {
          x += z;
          wait();
     }
 }

// definition of a thread process:
void
process ()  {
     unsigned val = 0;
     foo( val, 100, 4 );  // error: a function whose body contains wait
                          // statements must not be invoked by a method
                          // process
}

// definition of a thread process:
```

```
void
sprocess ()  {
      unsigned val = 0;
      wait();
      while( true ){
            foo( val, 100, 4 );  // OK
      }
}
```

8 Statements

This clause describes the different forms of sequential statements to be used in the synthesiable subset of SystemC. They are used to define algorithms for the execution of a subprogram or process; they execute in the order in which they appear.

```
statement ::=
  labeledstatement
  ~~| expression_statement~~
  | compound-statement
  | wait-statement
  | signal-assignment-statement
  | variable-assignment-statement
  | procedure-call-statement
  | selection-statement
  | iteration-statement
  | jump-statement
   |   declaration-statement
  |   ~~try-block~~
```

8.1 Labeled statement

A statement can be labeled.

```
labeled-statement::=
      identifier : statement
                  | **case** constant-expression : statement
                  | **default** : statement
```

An identifier label declares the identifier. The identifier label might be used by synthesis tools to reference operations, but has no functional impact on the design. The scope of a label is the function in which it appears. Labels shall not be re-declared within a function. Labels have their own name space and do not interfere with other identifiers.

Case labels and default labels shall occur only in switch statements.

8.2 Compound statement

The compound statement (also, and equivalently, called “block”) is provided in order to group sets of statements together.

```
compound-statement::=
   statement-seq


statement-seq::=
     statement
     | statement-seq statement
```

A compound statement defines a local scope.

8.3 Wait statement

Only simple wait statements with integer arguments must be used (default integer argument is 1). Wait statements can only be used within threaded processes (SC_(C)THREAD) or within subprograms which are invoked by an SC_(C)THREAD.
A wait statement causes the suspension of a process or subprogram statement until the next event occurs, on which the process is sensitive.

wait_statement ::=
  wait () ;
 | **wait (** constant-expression **) ;**
 | **wait_until (** expression **) ;**

8.4 Signal assignment statement

A signal assignment statement must be used to assign values to signals or ports.

signal-assignment-statement ::=
  signal-or-port-identifier **. write (** expression **) ;**
  | *signal-or-port*-identifier **=** expression **;**

Signal or port identifier must denote a variable of type sc_signal or of a port type, which is declared in the surrounding module. The type of the expression must match the type that was used for declaration of the signal or port being assigned to.

Note that a synthesis tool will have to perform some error and consistency checking:

- different signal assignment statements located in different processes must not write to the same signal or port, unless the target port or signal is of resolved vector type (sc_signal_rv, sc_in_rv, sc_out_rv, sc_inout_rv)
- signal can only be assigned one value in between 2 events

8.5 Variable assignment

A variable assignment statement replaces the current value of a variable with a new value specified by an expression.

variable-assignment-statement ::=
  target assignment-operator expression **;**

target ::=
  postfix-expression

Postfix expression must be an lvalue. The type of the expression on the right side of the assignment must match the type of the target, or must be implicit.

8.6 Selection statements

```
selection_statement ::=
  if ( condition ) statement
  | if ( condition ) statement else statement
  ~~| switch ( condition ) statement~~
  | switch-statement
```

8.6.1 If statement

An if statement selects for execution one or none of the enclosed sequences of statements, depending on the value of a corresponding condition.

```
condition ::=
  expression
  ~~| type_specifier_seq declarator = assignment_expression~~
```

8.6.2 Switch statement

A switch statement selects for execution one of a number of alternative sequences of statements; the chosen alternative is defined by the value of an expression.

```
switch-statement ::=
  switch ( condition ) {
    switch-statement-alternative-list
    [ default : statement-seq ]
  }
```

The switch expression must be of type integer or enumeration type.

```
switch-statement-alternative-list ::=
    switch-statement-alternative
    | switch-statement-alternative-list switch-statement-alternative

  switch-statement-alternative ::=
    case-label-statement-list--
      statement-seq
      break-statement

case-label-statement-list ::=
    case-label-statement
    | case-label-statement-list case-label-statement

  case-label-statement ::=
    case constant-expression :
```

Synthesis results can differ depending on the style of describing switch statements. Specifically a switch statement is full if all possible choices are covered by switch statement alternatives(using a default branch ensures this property). A switch statement is parallel if all switch statement alternatives contain a break statement.

8.7 Procedure call statement

A procedure call invokes the execution of a procedure body.

procedure-call-statement ::=
  postfix-expression **(** [expression-list] **) ;**

8.8 Iteration statements

An iteration statement includes a sequence of statements that is to be executed repeatedly, zero or more times. The while, do and for iteration scheme from C++ are supported and show the same behaviour as in C++.

iteration-statement ::=
  while (condition **)** statement
  | **do** statement **while (** expression **) ;**
  | **for (** for-init-statement [condition] **;** [expression] **)** statement

for-init-statement ::=
  expression-statement
  | simple-declaration

8.9 Jump statement

Jump statements unconditionally transfer control.

jump_statement ::=
break ;
| **continue** ;
| **return** {expression} ;
| **goto** label-name

8.9.1 Break

A break statement is used to complete the execution of an enclosing loop or switch statement.. The execution continues directly with the statement sequence following the loop or switch statement.

8.9.2 Continue statement

A continue statement is used to complete the execution of an enclosing loop statement.

8.9.3 Return statement

A return statement is used to complete the execution of the innermost enclosing function or procedure. The execution is transferred back to the call site, from where the subprogram is called.

A return statement must only occur within a subprogram, not inside a process.

8.10 Declaration statement

```
declaration-statement ::=
    block-declaration
```

9 Processes

Extension

.

sc-process-statement ::=
  SC_METHOD(identifier **) ;** sc-sensitivity-list
  | **SC_CTHREAD(** identifier **,** sc-event **) ;** [sc-watching-statement]

The identifier in a process statement must denote a process definition in the same scope.

. *Example*:

```
SC_MODULE( HA ) {
      sc_in< sc_bit > x, y;
      sc_out< sc_bit > c, s;

      void add() {       // This process describes purely combinational logic.
            s.write( x.read() ^ y.read() );
            c.write( x.read() & y.read() );
      }

      SC_CTOR( HA ) {
            SC_METHOD( add );
            sensitive << x << y;
      }
};
```

Each process of a module has first to be declared as a member function of the respective module. This is done by declaring a non-const procedure with an empty formal parameter list in the body of a module. This so-called process procedure must contain a process body instead of a subprogram body. The process body must be specified according to the following syntactic rule:

sc-process-definition ::=
  void sc-process-id **( )** sc-process-body

sc-process-id ::=
  identifier
  | template-id
  | [**::**] nested-name-specifier [**template**] identifier
  | [**::**] nested-name-specifier [**template**] template-id

sc-process-body ::=
  sc-method-body
  | sc-cthread-body

The scope of a process definition must be a module.

Any process definition must be accompanied by exactly one appropriate process statement in the module constructor of the module the process definition belongs to. Vice versa, each function definition which belongs to a module and which has an associated process statement in the body of the module constructor is called a process definition and must adhere to the rules listed in this section.

9.1 SC_METHOD

```
sc-method-body ::=
        compound-statement
```

The body of a SystemC Method process (SC_METHOD) must not contain any wait statement or any invocation of a function which may directly or indirectly cause the execution of a wait statement. Consequently, it must not contain any loop which is not unrollable.

```
sc-sensitivity-list ::=
          sc-sensitivity-clause
        | sc-sensitivity-list sc-sensitivity-clause

sc-sensitivity-clause ::=
          sensitive ( sc-event ) ;
        | sensitive_pos ( identifier ) ;
        | sensitive_neg ( identifier ) ;
        | sensitive sc-event-stream ;
        | sensitive_pos sc-identifier-stream ;
        | sensitive_neg sc-identifier-stream ;

sc-event-stream ::=
          << sc-event
        | sc-event-stream << sc-event

sc-identifier-stream ::=
          << identifier
        | sc-identifier-stream << identifier.
```

9.1.1 Synthesis semantics

Dependent on the coding style used within the body of a method process and dependent on the sensitivity list, a process may describe sequential logic as well as purely combinational logic.

Example:

```
SC_MODULE( L_AND ) {
      sc_in< sc_logic > in_a, in_b;
      sc_out< sc_logic > output;

      void comb() {       // This process describes purely combinational logic.
                          // It is “virtually” level-triggered
            output.write( in_a.read() & in_b.read() );
      }

      SC_CTOR( L_AND ) {
            SC_METHOD( comb );
            sensitive << in_a << in_b;
      }
};
```

```
SC_MODULE( Counter ) {
     sc_in< bool > clk;
     sc_in< bool > rst_n;
     sc_in< bool > enable;
     sc_out< unsigned int > val;

     unsigned int countVar;

     void seq() {      // This process describes sequential logic.
          if ( !rst_n.read() ) {
               countVar = 0;
          }
          else if ( enable.read() )
          {
               countVar += 1;
               val.write( countVar );
          }
     }

     SC_CTOR( Counter ) {
          SC_METHOD( seq );
          sensitive_neg << rst_n;
          sensitive_pos << clk;
     }
};
```

9.2 SC_CTHREAD

sc-event ::=
  identifier
  | identifier **. pos()**
  | identifier **. neg()**

Each identifier used as event or in conjunction with pos() or neg() must denote a signal/port declaration in the same scope.

sc-watching-statement ::=
  watching (identifier **. delayed() ==** sc-watching-condition **) ;**

sc-watching-condition ::=
  boolean-literal
  | logic-literal

sc-cthread-body ::=
  compound-statement **wait(); while (true) {** compound-statement **}**

The structure of an SC_CTHREAD must prevent execution from ever reaching the end of the process. A common way to achieve this is to introduce an infinite loop. Each unbounded loop must contain at least one explicit wait() in each control path. Any behavior encountered prior to encountering the first wait is considered reset behavior. The following structures are suggested for SC_CTHREADs. A synthesis tool must at least support these structures. In addition, it may support any other structure which satisfies the above requirements.

Example:

```
 // Simple SC_CTHREAD structure for synthesis
  void process() {
        // reset
        reset_behavior(); // must be executable in a single cycle
       wait();              // first wait implies end of reset

        // infinite loop
        while (true) {
                rest_of_behavior(); // must contain 1 wait per control path
        }
  }

  // Reset reaches into infinite loop
  void process() {
        // reset
        reset_behavior(); // must be executable in a single cycle

        // infinite loop
        while (true) {
            // everything located here is also executed during a reset
            wait();                        // first wait() in process
            rest_of_behavior(); // must contain 1 wait per control path
        }
  }

  // SC_CTHREAD structure for synthesis, with ""initialization part""
  void process() {
        // reset
        reset_behavior(); // must be executable in a single cycle
       wait();
        initialization(); // may contain any number of wait()s. This part
                          // is only executed once, after a reset.
        // infinite loop
        while (true) {
                rest_of_behavior(); // must contain 1 wait per control path
        }
  }
```

The following alternatives are also supported for modeling the the infinite loop:

```
     for(;;) { }
```

and

```
     do {} while (true);
```

9.3 SC_THREAD

Not supported.

10 Submodule instantiation

A submodule that is declared in the body of a module must be instantiated somewhere in the body of the module constructor or initialized by the constructor initializer.
This is done by means of a submodule instantiation statement or member variables declaration. .

```
sc-module-instantiation-statement ::=
        identifier =new [ :: ] [ nested-name-specifier ] class_name ( string-literal ) ;
```

Ports can be connected to signals or other ports. This is done by means of port mappings. SystemC provides two slightly different mechanisms for mapping ports – positional mapping and named mapping –, which are both supported.

```
sc-port-binding-statement ::=
          sc-named-port-binding-statement ;
        | sc-positional-port-binding-statement ;

sc-named-port-binding-statement ::=
        identifier -> id-expression ( id-expression ) ;
        identifier . id-expression ( id-expression ) ;

sc-positional-port-binding ::=
        [ * ] identifier  ( identifier-list ) ;

identifier-list ::=
          id-expression
        | identifier-list id-expression
```

Identifier must denote a sub module that is instantiated in a statement preceding the port mapping in a module constructor. The first id-expression in a named port binding statement must denote a port within any sub-module of the enclosing module. The second id-expression must denote a port or signal of the enclosing module. The types of the bound ports/signals must be identical

```
sc-sub-module-declaration ::=
        id-expression [ * ] identifier ;
```

The id-expression must be the name of module.

If a sub-module is declared as a pointer, the module constructor of the declaring module must contain a matching module-instantiation-statement, otherwise all module constructor definitions must include a matching mem-initializer.

```
.....
SC_CTOR(some_module) : submodule_identifier( “string”)  ,
 … {
 // Constructor
  }
```

Submodule identifier must denote a module, which is declared somewhere in the body of the surrounding module. The string which is being passed as argument for the module instantiation is required by the module constructor but does not play any role for synthesis and therefore may be arbitrary chosen.
Example:
```
      SC_MODULE(MyModule) {
        sc_in_clk    CLK;
        sc_in<bool>  RST;
        sc_in<int>  a;
        sc_in<int>  b;
        sc_out<int> c;
        sc_out<bool> RDY;
        sc_signal<int> tmp;

        Adder add;
        GCD *gcd;

        SC_CTOR(MyModule): add(“add“) {
          add(a,b,tmp);
          gcd = new GCD(“GCD“);
          gcd->CLK(CLK);
          gcd->RST(RST);
          gcd->x(tmp);
          gcd->y(b);
          gcd->z(c)
          gcd->RDY(RDY);
         }
      };
```

11 Namespaces

Non-const global/shared variables are not supported for synthesis. Within a function body only those names of variables must be used, which are declared previously in the function body, or which are passed as parameters. Global constants are supported for synthesis. *Example*:

```
namespace NSP {
      int var;
      const int CNST = 42;
}

void foo( const int val ) {
      using namespace NSP;
      int dummy = CNST;     // OK. Note that the occurrence of name CNST
                            // may be replaced by the value '42' by a synthesis
                            // tool.
      dummy = var;          // error. The name of a variable being declared in
                            // another namespace must not be used within a
                            // function body.
      var = val;            // error. The name of a variable being declared in
                            // another namespace must not be used within a
                            // function body.
}
```

11.1.1 Namespace definition

Supported.

namespace-name ::=
  original-namespace-name
  | namespace-alias

original-namespace-name ::=
  identifier

namespace-definition ::=
  named-namespace-definition
  | unnamed-namespace-definition

named-namespace-definition ::=
  original-namespace-definition
  | extension-namespace-definition

original-namespace-definition ::=
  namespace identifier **{** namespace-body **}**

extension-namespace-definition ::=
  namespace original-namespace-name **{** namespace-body **}**

unnamed-namespace-definition ::=
  namespace **{** namespace-body **}**

namespace-body ::=
    [declaration-seq]

11.1.2 Unnamed namespaces

Supported.

11.1.3 Namespace member definitions

Supported.

11.1.4 Namespace alias

Supported.

namespace-alias ::=
    identifier

namespace-alias-definition ::=
    namespace identifier **=** qualified-namespace-specifier **;**

qualified-namespace-specifier ::=
    [**::**] [nested-name-specifier] namespace-name

11.1.5 The `using` declaration

Supported.

using-declaration ::=
    using [**typename**] [**::**] nested-name-specifier unqualified-id **;**
    | **using ::** unqualified-id **;**

11.1.6 Using directive

Supported.

using-directive ::=
    using namespace [**::**] [nested-name-specifier] namespace-name **;**

12 Classes

Restricted support.

```
class-name ::=
          identifier
        | template-id

class-specifier ::=
        class-head { [ member-specification ] }

class-head ::=
          class-key [ identifier ] [ base-clause ]
        | class-key nested-name-specifier identifier [ base-clause ]
        | class-key [ nested-name-specifier ] template-id [ base-clause ]

class-key ::=
          class
        | struct
        ~~| union~~
```

12.1.1 Class names

(*No differences to C++.*)

12.1.2 Class members

```
member-specification ::=
          member-declaration [ member-specification ]
        | access-specifier : [ member-specification ]

member-declaration ::=
          [ decl-specifier-seq ] [ member-declarator-list ] ;
        | function-definition [ ; ]
        | [ :: ] nested-name-specifier [ template ] unqualified-id ;
        | using-declaration
        | template-declaration

member-declarator-list ::=
          member-declarator
        | member-declarator-list , member-declarator

member-declarator ::=
          declarator [ pure-specifier ]
        | declarator [ constant-initializer ]
        | [ identifier ] : constant-expression

pure-specifier ::=
        = 0

constant-initializer ::=
        = constant-expression
```

12.1.3 Member functions

Supported.

Note: the term member function does only refer to those functions being declared as member of a class or struct, but not to functions being declared as member of a module.

12.1.3.1 Nonstatic member functions

Supported.

12.1.3.2 The `this` pointer

The **this** pointer is the only kind of pointer supported for synthesis.

12.1.4 Static members

Restricted support.

12.1.4.1 Static member functions

The same restrictions and coding guidelines hold as for any other function.

12.1.4.2 Static data members

Only const static data members are supported.

Example:

```
class X {
public:
     static int m1;              // error: non-const static data
                                 // member is not supported
     const static int m2 = 10;   // OK
};
```

12.1.5 Unions

Not supported.

12.1.6 Bit-fields

Supported .

12.1.7 Nested class declarations

Supported.

12.1.8 Local class declarations

Supported.

12.1.9 Nested type names

(*No differences to C++.*)

12.2 Derived classes

base-clause ::=
  : base-specifier-list

```
base-specifier-list ::=
          base-specifier
        | base-specifier-list , base-specifier

base-specifier ::=
          [ :: ] [ nested-name-specifier ] class-name
        | virtual [ access-specifier ] [ :: ] [ nested-name-specifier ] class-name
        | access-specifier [ virtual ] [ :: ] [ nested-name-specifier ] class-name

access-specifier ::=
          private
        | protected
        | public
```

12.2.1 Multiple base classes
Supported.

12.2.2 Member name lookup
(*No differences to C++.*)

12.2.3 Virtual functions
Supported.

12.2.4 Abstract classes
Supported.

12.3 Member access control
(*No differences to C++.*)

12.3.1 Access specifiers
(*No differences to C++.*)

12.3.2 Accessibility of base classes and base class members
(*No differences to C++.*)

12.3.3 Access declarations
(*No differences to C++.*)

12.3.4 Friends
(*No differences to C++.*)

12.3.5 Protected member access
(*No differences to C++.*)

12.3.6 Access to virtual functions
(*No differences to C++.*)

12.3.7 Multiple access
(*No differences to C++.*)

12.3.8 Nested classes

(*No differences to C++*.)

12.4 Special member functions

Restricted supported.

12.4.1 Constructors

Constructors of user defined classes are supported. Only the exception is that a synthesis tool shall not invoke the constructors of user defined classes, when the classes are used as type for signals and ports, or data members of a module. All signals, ports, and data members of a module should be initialized in reset clause instead of in module constructors, otherwise, synthesis results may not match simulation results.
Example:

```
class X {
public:
  X() {
    m_1 = 0;
  }
private:
  int m_1;
};

class XChild : public X {
};

SC_MODULE(Module) {
  sc_signal< X > xSig;
  sc_signal< XChild > xChildSig;
  sc_in_clk clk;
  sc_in<bool> rst;

  SC_CTOR(Module)
   : xSig("xSig"),            // Warning! Not invoke xSig::xSig().
     xChildSig("xChildSig") // Warning! Not invoke  xChildSig::xChildSig().
  {
     SC_CTHREAD(proc, clk.pos());
     watching(rst.delayed() == true);
  }

  void proc() {
     // Reset clause
     X x_tmp;                  // OK. Invoke xSig::xSig().
     xSig = x_tmp;             // OK. Initialize xSig with x_tmp.
     xChildSig = XChild();   // OK. Initialize xChildSig by the default
constructor.
     wait();

     // Main loop
     while (true) {
       ...
     }
  }
```

```
    };
```

12.4.2 Temporary objects

(*No differences to C++.*)

12.4.3 Conversions

(*No differences to C++.*)

12.4.3.1 Conversion by constructor

(*No differences to C++.*)

12.4.3.2 Conversion functions

(*No differences to C++.*)

conversion-function-id ::=
  operator conversion-type-id

conversion-type-id ::=
  type-specifier-seq [conversion-declarator]

conversion-declarator ::=
  ptr-operator [conversion-declarator]

12.4.4 Destructors

Supported.

12.4.5 Free store

Not supported.

12.4.6 Initialization

Restricted support.

Non-const members of modules must not be initialised by means of mem-initializers.

12.4.6.1 Explicit initialisation

Supported.

12.4.6.2 Initializing bases and members

Supported.

ctor-initializer ::=
  : mem-initializer-list

mem-initializer-list ::=
  mem-initializer
  | mem-initializer **,** mem-initializer-list

mem-inititalizer ::=
  mem-initializer-id **(** [expression-list] **)**

```
mem-initializer-id ::=
            [ :: ] [ nested-name-specifier ] class-name
          | identifier
```

12.4.7 Construction and destruction

For construction the same rules as in C++ apply. Without support for destructors, destruction of objects, i.e. temporaries and local class instances, does not have any visible effect.

12.4.8 Copying class objects

(*No differences to C++.*)

13 Overloading

Restricted support. (Some operators are excluded from overloading, others must follow certain coding guidelines, see Overloaded operators)

13.1.1 Overloadable declarations

(*No differences to C++.*)

13.1.2 Declaration matching

(*No differences to C++.*)

13.1.3 Overload resolution

(*No differences to C++.*)

13.1.3.1 Candidate functions and argument lists

(*No differences to C++.*)

Function call syntax

(*No differences to C++.*)

Call to named function

(*No differences to C++.*)

Call to object of class type

(*No differences to C++.*)

Operators in expressions

(*No differences to C++.*)

Initialization by constructor

(*No differences to C++.*)

Copy-initialization of class by user-defined conversion

(*No differences to C++.*)

Initialization by conversion function

(*No differences to C++.*)

Initialization by conversion function for direct reference binding

(*No differences to C++.*)

13.1.3.2 Viable functions

(*No differences to C++.*)

13.1.3.3 Best Viable Function

(*No differences to C++.*)

Implicit conversion sequences

(*No differences to C++.*)

Standard conversion sequences
(*No differences to C++.*)

User-defined conversion sequences
(*No differences to C++.*)

Ellipsis conversion sequences
Not supported.

Reference Binding
(*No differences to C++.*)

Ranking implicit conversion sequences
(*No differences to C++.*)

13.1.4 Address of overloaded function

Address operations are not supported.

13.1.5 Overloaded operators

```
operator-function-id ::=
        operator operator

operator ::=
        ~~new   | delete | new[] | delete[]~~
        | +    | -    | *    | /    | %    | ^    | &    | |    | ~
        | !    | =    | <    | >    | +=   | -=   | *=   | /=   | %=
        | ^=   | &=   | |=   | <<   | >>   | >>=  | <<=  | ==   | !=
        | <=   | >=   | &&   | ||   | ++   | --   | ,    | ->*  | ->
        | ()   | []
```

Classes which are used as type for signals and ports must define **operator==**. An assignment of an instance of a class to a signal or port of the same or any assignment compatible class is actually performed, only if a comparison by means of **operator==** of the actual value of the target signal or port and the source of the assignment returns ‘false’. Note, that for proper functionality **operator==** must return ‘false’, if and only if target and source are different in at least one member.

Example:

```
class X {
public:
      int m_1;
      int m_2;

      bool operator==( const X & obj ) {
            return( false );
      }
};

class Y {
public:
```

```
          int m_1;
          int m_2;

          bool operator==( const Y & obj ) {
               return( true );
          }
     };

     class Z {
     public:
          int m_1;
          int m_2;

          bool operator==( const Z & obj ) {
               if ( m_1 != obj.m_1 ) {
                    return( false );
               } else if ( m_2 != obj.m_2 ) {
                    return( false );
               } else {
                    return( true );
               }
          }
     };

     sc_signal< X > xSig;  //  error: X::operator== does not compare members
     sc_signal< Y > ySig;  //  error: Y::operator== does not compare members
     sc_signal< Z > zSig;  // OK: Z::operator== well defined
```

Deprecated items
The following operators are not supported:

new | delete| new[]| delete[]

13.1.5.1 Unary operators
(*No differences to C++.*)

13.1.5.2 Binary operators
(*No differences to C++.*)

13.1.5.3 Assignment
(*No differences to C++.*)

13.1.5.4 Function call
(*No differences to C++.*)

13.1.5.5 Subscripting
(*No differences to C++.*)

13.1.5.6 Class member access
(*No differences to C++.*)

13.1.5.7 Increment and decrement

(*No differences to C++.*)

13.1.6 Built-in operators

(*No differences to C++.*)

14 Templates

Supported.

template-declaration ::=
[**export**] **template** < template-parameter-list > declaration

template-parameter-list ::=
template-parameter
| template-parameter-list **,** template-parameter

14.1.1 Template parameters

template-parameter ::=
type-parameter
| parameter-declaration

type-parameter ::=
class [identifier]
| **class** [identifier] = type-id
| **typename** [identifier]
| **typename** [identifier] = type-id
| **template** < template-parameter-list > **class** [identifier]
| **template** < template-parameter-list > **class** [identifier] = id-expression

Template parameters must not include pointers to functions.

14.1.2 Names of template specializations

template-id ::=
template-name < [template-argument-list] >

template-name ::=
identifier

template-argument-list ::=
template-argument
| template-argument-list **,** template-argument

template-argument ::=
assignment-expression
| type-id
| id-expression

14.1.3 Template arguments

(*No differences to C++.*)

14.1.3.1 Template type arguments

(*No differences to C++.*)

14.1.3.2 Template non-type arguments

(*No differences to C++.*)

14.1.3.3 Template template arguments

(*No differences to C++.*)

14.1.4 Type equivalence
(*No differences to C++.*)

14.1.5 Template declarations
(*No differences to C++.*)

14.1.5.1 Class Templates
(*No differences to C++.*)

Member functions of class templates
(*No differences to C++.*)

Member classes of class templates
(*No differences to C++.*)

Static data members of class templates
(*No differences to C++.*)

14.1.5.2 Member templates
(*No differences to C++.*)

14.1.5.3 Friends
(*No differences to C++.*)

14.1.5.4 Class template partial specializations
(*No differences to C++.*)

Matching of class template partial specializations
(*No differences to C++.*)

Partial ordering of class template specializations
(*No differences to C++.*)

Members of class template specializations
(*No differences to C++.*)

14.1.5.5 Function templates
(*No differences to C++.*)

Function template overloading
(*No differences to C++.*)

Partial ordering of function templates
(*No differences to C++.*)

14.1.6 Name resolution
(*No differences to C++.*)

14.1.6.1 Locally declared names

(*No differences to C++.*)

14.1.6.2 Dependent names

(*No differences to C++.*)

Dependent types

(*No differences to C++.*)

Type-dependent expressions

(*No differences to C++.*)

Value-dependent expressions

(*No differences to C++.*)

Dependent template arguments

(*No differences to C++.*)

14.1.6.3 Non-dependent names

(*No differences to C++.*)

14.1.6.4 Dependent name resolution

(*No differences to C++.*)

Point of instantiation

(*No differences to C++.*)

Candidate functions

(*No differences to C++.*)

14.1.6.5 Friend names declared within a class template

(*No differences to C++.*)

14.1.7 Template instantiation and specialization

(*No differences to C++.*)

14.1.7.1 Implicit instantiation

(*No differences to C++.*)

14.1.7.2 Explicit instantiation

(*No differences to C++.*)

explicit-instantiation ::=
    template declaration

14.1.7.3 Explicit specialization

(*No differences to C++.*)

explicit-specialization ::=
    template **< >** declaration

14.1.8 Function template specializations
(*No differences to C++.*)

14.1.8.1 Explicit template argument specification
(*No differences to C++.*)

14.1.8.2 Template argument deduction
(*No differences to C++.*)

Deducing template arguments from a function call
(*No differences to C++.*)

Deducing template arguments taking the address of a function template
(*No differences to C++.*)

Deducing conversion function template arguments
(*No differences to C++.*)

Deducing template arguments from a type
(*No differences to C++.*)

14.1.8.3 Overloaded resolution
(*No differences to C++.*)

15 Preprocessing directives

The full set of C/C++ preprocessing directives is supported (refer to clause 16 in [3]).
A synthesis tool shall recognize pragma directives (#**pragma**). It may ignore or process pragma directives under its synthesis policy.
A Synthesis tool shall predefine the following macro names:

1. **__STDC__** : The value is implementation-dependent.
2. **__cplusplus** : The value is implementation-dependent.
3. **SC_SYNTHESIS** : The value means the version of the synthesis subset, i.e., the version of this document. A value of 0x123 means a version number of 1.23.

By using **SC_SYNTHESIS**, code pieces, that are helpful for debugging and simulation, but which shall not or can not be synthesized can be switched off for synthesis:

```
#ifdef SC_SYNTHESIS
#  if SC_SYNTHESIS >= 0x200
… // the code for synthesis subset of version 2.00 or later
#  else
… // the code for synthesis subset before version 2.00
#  endif
#else
… // the code for simulation
#endif
```

16 Lexical elements

The Lexical elements of the synthesis subset are the same as for C++. Therefore refer to 2.13, 2.13.1, 2.13.2 and 2.13.5 in [3] for further details.

17 Scope and visibility

The scope rules for the synthesis subset are the same as for C++ (refer to clause 3.1, 3.3, 3.4 and 3.5 in [3]), including:

1. namespace (**namespace** keyword),
2. scope resolution operator (::),
3. using-directive (**using** keyword),
4. nested namespace, and
5. nested class.

*A synthesis tool shall recognize access specifiers (**private**, **protected**, **public**, and **friend** keywords), but may ignore them.*

18 Miscellaneous

18.1 Tracing

A synthesis tool shall recognize tracing constructs, but may ignore them:

- declaration of a variable of type **sc_trace_file***,
- assignment to a variable of type **sc_trace_file***,
- calls to **sc_close_isdb_trace_file**, **sc_close_wif_trace_file** and **sc_close_vcd_trace_file**,
- declaration or definition of any function named **sc_trace**, and
- calls to any function named **sc_trace**.

Note that those constructs must not contain any side effects, e.g., changing a value of a variable.

18.2 Outputting messages to stdout and/or cout

A synthesis tool shall recognize the following constructs, but may ignore them:

- **printf** function, and
- using **operator** << to **cout**.

Note that those constructs must not contain any side effects, e.g., changing a value of a variable:

```
printf("x = %d", x);  // OK
 printf("y = %d", ++y);    // NG
 cout << "z = " << z << endl;   // OK
```

18.3 Exception handling

Not supported.

Annex A Syntax summary (informative)

/* This comment is not the part of this document.

The following explanation of syntactic description is a modified version of the section 0.2.1 of VHDL LRM for comparing.

Syntactic description
The form employed for SystemC synthesis subset description is described by means of context-free syntax using a simple variant of the backus naur form and uses following conventions:

a) Lowercase words in roman font, some containing embedded hyphens, are used to denote syntactic categories, for example:
  nested-namespace-specifier
  Whenever the name of a syntactic category is used, apart from the syntax rules themselves, space take the place of underlines (thus, “nested namespace specifier” would appear in the narrative description when referring to the above syntactic category).
b) Boldface words are used to denote reserved words, for example:
  template
  Reserved words must be used only in those places indicated by the syntax.
c) A *production* consists of a *left-hand side*, the symbol “::=” (which is read as “can be replaced by”), and a *right-hand side*. The left-hand side of a production is always a syntactic category; the righthand side is a replacement rule. The meaning of a production is a textual-replacement rule: any occurrence of the left-hand side may be replaced by an instance of the right-hand side.
d) A vertical bar (|) separates alternative items on the right-hand side of a production unless it occurs immediately after an opening brace, in which case it stands for itself, as follows:
  class-or-namespace-name ::= class-name | name-space-name
  expression-list ::= assignment-expression | expression-list **,** assignment-expression
  In the instance, an occurrence of “class-or-namespace-name” can be replaced by either “class-name” or “name-space-name”. In the second case, “expression-list” can be replaced by a list of “assignment-expression”, separated by comma (,).
e) Square braces [ ] enclose a option item. The items may appear zero or one time. Thus, the following two productions are equivalent:
  init-declarator ::= declarator [initializer]
  init-declarator ::= declarator | declarator initializer
f) Shadowed items ▨ are used to denote added syntax for SystemC synthesis subset.
g) Slanted words are not parts of the syntax.
h) Deleted words with one line are not supported.

*/

A.1 Keywords

typedef-name ::=
  identifier

namespace-name ::=
  original-namespace-name
  | namespace-alias

original-namespace-name ::=
  identifier

namespace-alias ::=

identifier

class-name ::=
identifier
| template-id

enum-name ::=
identifier
template-name ::=
identifier

A.2 Lexical conventions

Synthesis tools are able to reject a character literal and a universal character name. A string literal is used only for the name of submodule.

hex-quad ::=
hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

~~universal-character-name~~ ::=
~~\u hex-quad~~
~~\U hex-quad hex-quad~~

preprocessing-token ::=
header-name
| identifier
| pp-number
~~| character-literal~~
| string-literal
| preprocessing-op-or-punc
each non-white-space character that cannot be one of the above

token ::=
identifier
| keyword
| literal
| operator
| punctuator

header-name ::=
< h-char-sequence >
| **“** q-char-sequence **“**

h-char-sequence ::=
h-char
| h-char-sequence h-char

h-char **::=**
any member of the source character set except new-line and >

q-char-sequence ::=
q-char
| q-char-sequence q-char

q-char ::=
any member of the source character set except new-line and “

pp-number ::=
digit

| **.** digit
| pp-number digit
| pp-number nondigit
| pp-number **e** sign
| pp-number **E** sign
| pp-number **.**

identifier ::=
nondigit
| identifier nondigit
| identifier digit

nondigit ::= *one of*
~~universal_character_name~~
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

digit ::= *one of*
0 1 2 3 4 5 6 7 8 9

preprocessing-op-or-punc ::= one of

{	**}**	**[**	**]**	#	##	**(**	**)**	
<:	:>	<%	%>	%:	%:%:	**;**	**:**	**...**
new	**delete**	**?**	**::**	**.**	**.***			
+	-	*	/	%	^	**&**	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	**&=**	|=	<<	>>	>>=	<<=	==	**!=**
<=	>=	**&&**	||	++	**--**	**,**	->*	->
and	**and_eq**	**bitand**	**bitor**	**compl**	**not**	**not_eq**		
or	**or_eq**	**xor**	**xor_eq**					

literal ::=
integer-literal
| ~~character-literal~~
| floating-literal
| ~~string-literal~~
| boolean-literal

integer-literal ::=
decimal-literal [integer-suffix]
| octal-literal [integer-suffix]
| hexadecimal-literal [integer-suffix]

decimal-literal ::=
nonzero-digit
| decimal-literal digit

octal-literal ::=
0
| octal-literal octal-digit

hexadecimal-literal ::=
0x hexadecimal-digit
| **0X** hexadecimal-digit
| hexadecimal-literal hexadecimal-digit

nonzero-digit ::= *one of*
    1 2 3 4 5 6 7 8 9

octal-digit ::= *one of*
    1 2 3 4 5 6 7

hexadecimal-digit ::= *one of*
    1 2 3 4 5 6 7 8 9
    a b c d e f
    A B C D E F

integer-suffix ::=
    unsigned-suffix [long-suffix]
  | long-suffix [unsigned-suffix]

unsigned-suffix ::= *one of*
    u U

long-suffix ::= *one of*
    l L

~~character-literal ::=~~
    ~~**‘** c-char-sequence **‘**~~
  ~~| **L ‘** c-char-sequence **‘**~~

~~c-char-sequence ::=~~
    ~~c-char~~
  ~~| c-char-sequence c-char~~

~~c-char ::=~~
    ~~*any number of the source character set except the signal_quote ‘, backslash \, or new_line character*~~
  ~~| escape-sequence~~
  ~~| universal-character-name~~

~~escape-sequence ::=~~
    ~~simple-escape-sequence~~
  ~~| octal-escape-sequence~~
  ~~| hexadecimal-escaple-sequence~~

~~simple-escape-sequence ::= *one of*~~
    ~~**\’ \” \? **~~
    ~~**\a \b \f \n \r \t \v**~~

~~octal-escape-sequence ::=~~
    ~~\ octal-digit~~
  ~~| \ octal-digit octal-digit~~
  ~~| \ octal-digit octal-digit octal-digit~~

~~hexadecimal-escape-sequence ::=~~
    ~~**\x** hexadecimal-digit~~
  ~~| hexadecimal-escape-sequence hexadecimal-digit~~

floating-literal ::=
    fractional-constant [exponent-part] [floating-suffix]
  | digit-sequence exponent-part [floating-suffix]

fractional-constant ::=
    [digit-sequence] **.** digit-sequence
  | digit-sequence **.**

exponent-part ::=
    e [sign] digit-sequence
  | E [sign] digit-sequence

sign ::= one of
  + -

digit-sequence ::=
    digit
  | digit-sequence digit

floating-suffix ::= *one of*
  f l F L

string-literal ::=
    “ [s-char-sequence] **“**
  | **L “** [s-char-sequence] **“**

s-char-sequence ::=
    s-char
  | s-char_sequence s-char

s-char ::=
    any member of the source character set except the double_quote “, bachslash \, or new_line character
  ~~| escape_sequence~~
  ~~| universal_character_name~~

boolean-literal ::=
    false
  | **true**

A.3 Basic concepts

translation-unit ::=
    [declaration-seq] [sc-main-definition]

A.4 Expressions

primary-expression ::=
    literal
  | **this**
  | **(** expression **)**
  | id-expression

id-expression ::=
    unqualified-id
  | qualified-id

unqualified-id
    identifier
  | operator-function-id

| conversion-function-id
| ~~~class-name~~
| template-id

qualified-id
[**::**] nested-namespace-specifier [**template**] unqualified-id
| **::** identifier
| **::** operator-function-id
| **::** template-id

nested-name-specifier ::=
class-or-namespace-name **::** [nested-name-specifier]
| class-or-namespace-name **:: template** nested-name-specifier

class-or-namespace-name ::=
class-name
| name-space-name

postfix-expression ::=
primary-expression
| postfix-expression **[** expression **]**
| postfix-expression **(** [expression-list] **)**
| simple-type-specifier **(** [expression-list] **)**
| **typename** [**::**] nested-name-specifier identifier **(** [expression-list] **)**
| **typename** [**::**] nested-name-specifier [**template**] template-id **(** [expression-list] **)**
| postfix-expression **.** [~~**template**~~] id-expression
| ~~postfix-expression -> [**template**] id-expression~~
| postfix-expression **.** pseudo-destructor-name
| ~~postfix-expression -> pseudo-destructor-name~~
| postfix-expression **++**
| postfix-expression **--**
| **dynamic-cast <** type-id **> (** expression **)**
| **static-cast <** type-id **> (** expression **)**
| **reinterpret-cast <** type-id **> (** expression **)**
| **const-cast <** type-id **> (** expression **)**
| ~~**typeid (** expression **)**~~
| ~~**typeid (** type-id **)**~~

expression-list ::=
assignment-expression
| expression-list **,** assignment-expression

~~pseudo-destructor-name~~ ::=
[**::**] [nested-name-specifier] type-name **::** ~ type-name
| [**::**] nested-name-specifier **template** template-id **::** ~ type-name
| [**::**] [nested-name-specifier] **~** type-name

unary-expression ::=
postfix-expression
| **++** cast-expression
| **--** cast-expression
| unary-operator cast-expression
| ~~**sizeof** unary-expression~~
| ~~**sizeof (** type-id **)**~~
| ~~new-expression~~
| ~~delete-expression~~

unary-operator ::= *one of*
    * & + - ! ~

~~new-expression~~ ::=
    [**::**] **new** [~~new-placement~~] ~~new-type-id~~ [~~new-initializer~~]
    | [**::**] **new** [~~new-placement~~] **(** type-id **)** [~~new-initializer~~]

~~new-placement~~ ::=
    (expression-list **)**

~~new-type-id~~ ::=
    type-specifier-seq [~~new-declarator~~]

~~new-declarator~~ ::=
    ptr-operator [~~new-declarator~~]
    | ~~direct-new-declarator~~

~~direct-new-declarator~~ ::=
    [expression **]**
    | ~~direct-new-declarator~~ **[** constant-expression **]**

~~new-initializer~~ ::=
    ([expression-list] **)**

~~delete-expression~~ ::=
    [**::**] **delete** cast-expression
    | [**::**] **delete [ ]** cast-expression

cast-expression ::=
    unary-expression
    | **(** type_id **)** cast-expression

pm-expression ::=
    cast-expression
    | ~~pm-expression .* cast-expression~~
    | ~~pm-expression ->* cast-expression~~

multiplicative-expression ::=
    pm-expression
    | multiplicative-expression * pm-expression
    | multiplicative-expression **/** pm-expression
    | multiplicative-expression **%** pm-expression

additive-expression ::=
    multiplicative-expression
    | additive-expression **+** multiplicative-expression
    | additive-expression **–** multiplicative-expression

shift-expression ::=
    additive-expression
    | shift-expression << additive-expression
    | shift-expression >> additive-expression

relational-expression ::=
    shift-expression
    | relational-expression < shift-expression
    | relational-expression > shift-expression

| relational-expression <= shift-expression
| relational-expression >= shift-expression

equality-expression ::=
relational-expression ::=
| equality-expression **==** relational-expression
| equality-expression **!=** relational-expression

and-expression ::=
equality-expression
| and-expression **&** equality-expression

exclusive-or-expression ::=
and-expression
| exclusive-or-expression ^ and-expression

inclusive-or-expression ::=
exclusive-or-expression
| inclusive-or-expression **|** exclusive-or-expression

logical-and-expression ::=
inclusive-or-expression
| logical-and-expression **&&** inclusive-or-expression

logical-or-expression ::=
logical-and-expression
| logical-or-expression || logical-and-expression

conditional-expression ::=
logical-or-expression
| logical-or-expression **?** expression **:** assignment-expression

assignment-expression ::=
conditional-expression
| logical-or-expression assignment-operator assignment-expression
| ~~throw-expression~~

assignment-operator ::= *one of*
= *= /= %= += -= >>= <<= &= ^= |=

expression ::=
assignment-expression
| expression **,** assignment-expression

constant-expression ::=
conditional-expression

A.5 Statements

statement ::=
labeled-statement
| expression-statement
| compound-statement
| wait-statement
| signal-assignment-statement
| variable-assignment-statement
| procedure-call-statement
| selection-statement

| iteration-statement
| jump-statement
| declaration-statement
| ~~try-block~~

labeled-statement ::=
    identifier **:** statement
| **case** constant-expression **:** statement
| **default :** statement

expression_statement ::=
    [expression] **;**

compound-statement ::=
    { [statement_seq] **}**

statement-seq ::=
    statement
| statement-seq statement

wait-statement ::=
    wait ( ) ;
| **wait (** constant-expression **) ;**

signal-assignment-statement ::=
    signal-or-port-identifier **. write (** expression **) ;**
| *signal-or-port*-identifier **=** expression **;**

variablel-assignment-statement ::=
    target assignment-operator expression **;**

target ::=
    postfix-expression

selection_statement ::=
    if (condition **)** statement
| **if (** condition **)** statement **else** statement
~~| **switch (** condition **)** statement~~
| switch-statement
condition ::=
    expression
~~| type_specifier_seq declarator = assignment_expression~~

switch-statement ::=
    switch (condition **) {** switch-statement-alternative-list [**default :** statement-seq] **}**

switch-statement-alternative-list ::=
    switch-statement-alternative
| switch-statement-alternative-list switch-statement-alternative

switch-statement-alternative ::=
    case-label-statement-list statement-seq break-statement

case-label-statement-list ::=
    case-label-statement
| case-label-statement -list case-label-statement

case-label-statement ::=
  case constant-expression

procedure-call-statement ::=
  postfix-expression **(** [expression-list] **)**

iteration-statement ::=
  while (condition **)** statement
  | **do** statement **while (** expression **) ;**
  | **for (** for-init-statement [condition] **;** [expression] **)** statement

for-init-statement ::=
  expression-statement
  | simple-declaration

jump-statement ::=
  break ;
  | **continue ;**
  | **return** [expression] **;**
  | **goto** label-name **;**

declaration-statement ::=
  block-declaration

A.6 Declarations

declaration-seq ::=
  declaration
  | declaration-seq declaration

declaration ::=
  block-declaration
  | function-declaration
  | template-declaration
  | explicit-instantiation
  | explicit-specialization
  | ~~linkage-specification~~
  | namespace-definition
  | sc-process-definition

block-declaration ::=
  simple-declaration
  | ~~asm-definition~~
  | namespace-alias-definition
  | using-declaration
  | using-directive

simple-declaration ::=
  [decl-specifier-seq] [init-declarator-list] **;**

decl-specifier ::=
  storage-class-specifier
  | type-specifier
  | function-specifier
  | **friend**
  | **typedef**

decl-specifier-seq ::=

[decl-specifier-seq] decl-specifier

storage-class-specifier ::=
  auto
| **register**
| **static**
| **extern**
| **mutable**

function-specifier ::=
  inline
| **virtual**
| **explicit**

typedef-name ::=
  identifier

type-specifier ::=
  simple-type-specifier
| class-specifier
| enum-specifier
| elaborated-type-specifier
| cv-qualifier
| sc-type-specifier
| sc-module-specifier

simple-type-specifier ::=
  [**::**] [nested-name-specifier] type-name
| [**::**] nested-name-specifier **template** template-id
| **char**
| ~~**wchar_t**~~
| **bool**
| **short**
| **int**
| **long**
| **signed**
| **unsigned**
| ~~**float**~~
| ~~**double**~~
| **void**

type-name ::=
  class-name
| enum-name
| typedef-name

elaborated-type-specifier ::=
  class_key [**::**] [nested_name_specifier] identifier
| **enum** [**::**] [nested_name_specifier] identifier
| **typename** [**::**] nested_name_specifier identifier
| **typename** [**::**] nested_name_specifier [**template**] template_id

enum-name ::=
  identifier

enum-specifier ::=
  enum [identifier] **{** [enumerator-list] **}**

enumerator-ist ::=
    enumerator-difinition
  | enumerator-list **,** enumerator-difinition

enumerator-difinition ::=
    enumerator
  | enumerator **=** constant-expression

enumerator ::=
    identifier

namespace-name ::=
    original-namespace-name
  | namespace-alias

original-namespace-name
    identifier

namespace-definition ::=
    named-namespace-definition
  | unnamed-namespace-definition

named-namespace-definition ::=
    original-namespace-definition
  | extension-namespace-definition

original-namespace-definition ::=
    namespace identifier [namespace-body]

extension-namespace-definition ::=
    namespace original-namespace-name [namespace-body]

unnamed-namespace-definition ::=
    namespace [namespace-body]

namespace-body ::=
    [declaration-seq]

namespace-alias ::=
    identifier

namespace-alias-definition ::=
    namespace identifier **=** qualified-namespace-specifier **;**

qualified-namespace-specifier ::=
    [**::**] [nested-name-specifier] namespace-name

using-declaration ::=
    using [**typename**] [**::**] nested-name-specifier unqualified-id **;**
  | **using ::** unqualified-id **;**

using-directive ::=
    using namespace [**::**] [nested-name-specifier] namespace-name **;**

~~asm-definition ::=~~
    ~~**asm (** string-literal **) ;**~~

~~linkage-specification ::=~~
~~**extern** string-literal **{** [declaration-seq] **}**~~
~~| **extern** string-literal declaration~~

A.6-1 SystemC Type Specifiers

sc-type-specifier ::=
sc_int < constant-expression >
| **sc_uint** < constant-expression >
| **sc_bigint** < constant_expression >
| **sc_biguint** < constant_expression >
| **sc_logic**
| **sc_lv** < constant_expression >
| **sc_bit**
| **sc_bv** < constant_expression >
| **sc_fixed** < constant_expression **,** constant_expression
[**,** sc-quantization-mode-specifier] [**,** sc-overflow-mode-specifier]
[**,** constant-expression] >
| **sc_ufixed** < constant_expression **,** constant_expression
[**,** sc-quantization-mode-specifier] [**,** sc-overflow-mode-specifier] [**,** constant-expression]
>

sc-quantization-mode-specifier ::=
SC_RND
| **SC_RND_ZERO**
| **SC_RND_MIN_INF**
| **SC_RND_INF**
| **SC_RND_CONV**
| **SC_TRN**
| **SC_TRN_ZERO**

sc-overflow-mode-specifier ::=
SC_SAT
| **SC_SAT_ZERO**
| **SC_SAT_SYN**
| **SC_WRAP**
| **SC_WRAP_SM**

A.7 Declarators

init-declarator-list ::=
init-declarator
| init-declarator-list **,** init-declarator

init-declarator ::=
declarator [initializer]

declarator ::=
direct-declarator
| ptr-operator declarator

direct-declarator ::=
declarator-id
| direct-declarator **(** parameter-declaration-clause **)** [cv-qualifier-seq]
~~[exception-specification]~~
| direct-declaration **[** [constant-expression] **]**

| **(** declarator **)**

ptr-operator ::=
  ~~***** [cv-qualifier-seq]~~
  | **&**
  ~~| [**::**] [nested-name-specifier] ***** [cv-qualifier-seq]~~

cv-qualifier-seq ::=
  cv-qualifier [cv-qualifier-seq]

cv-qualifier ::=
  const
  | **volatile**

declarator-id ::=
  id-expression
  | [**::**] [nested-name-specifier] type-name

type-id ::=
  type-specifier-seq [abstract-declarator]

type-specifier-seq ::=
  type-specifier [type-specifier-seq]

abstract-declarator ::=
  ptr_operator [abstract-declarator]
  | direct-abstract-declarator

direct-abstract-declarator ::=
  [direct-abstract-declarator] **(** parameter-declaration-clause **)** [cv-qualifier-seq]
  ~~[exception-specification]~~
  | [direct-abstract-declarator] **[** [constant-expression] **]**
  | **(** abstract-declarator **)**

parameter-declaration-clause ::=
  [parameter-declaration-list] ~~[**...**]~~
  | parameter-declaration-list ~~**, ...**~~

parameter-declaration-list ::=
  parameter-declaration
  | parameter-declaration-list **,** parameter-declaration

parameter-declaration ::=
  decl-specifier-seq declarator
  | decl-specifier-seq declarator **=** assignment-expression
  | decl-specifier-seq [abstract-declarator]
  | decl-specifier-seq [abstract-declarator] **=** assignment-expression
  | [**const**] sc-signal-declaration **&** identifier

function-definition ::=
  [decl-specifier-seq] declarator [ctor-initializer] function_body
  ~~| [decl-specifier-seq] declarator function-try-block~~

function-body ::=
  compound-statement

initializer ::=

= initializer-clause
| **(** expression-list **)**

initializer-clause ::=
assignment-expression
| **{** initializer-list [**,**] **}**
| **{ }**

initializer-list
initializer-clause
| initializer-list **,** iitializer-clause

A.8 Classes

Classes are regarded as a module or a user defined type in SystemC synthesis. The syntax for module is described in A.8-1 section. There is many limitation for a user defined type.

class-name ::=
identifier
| template-id

class-specifier ::=
class-head **{** [member-specification] **}**
-

class-head ::=
class-key [identifier] [base_clause]
| class-key nested-name-specifier identifier [base-clause]
| class-key [nested-name-specifier] template-id [base-clause]

class-key ::=
class
| **struct**
| ~~**union**~~

member-specification ::=
member-declaration [member-specification]
| access-specifier **:** [member-specification]

member-declaration ::=
[decl-specifier-seq] [member-declarator-list] **;**
| function-definition [**;**]
| [**::**] nested-name-specifier [**template**] unqualified-id **;**
| using-declaration
| template-declaration

member-declarator-list ::=
member-declarator
| member-declarator-list [**,**] member-declarator

member-declarator ::=
declarator [pure-specifier]
| declarator [constant-initializer]
| [identifier] **:** constant-expression

pure-specifier ::=
= 0

constant-initializer ::=

= constant-expression

A.8-1 Module Declaration

sc-module-specifier ::=
sc-module-head { [module-member-specification] }

sc-module-head ::=
SC_MODULE(identifier **)**
| class-key [nested-name-specifier] identifier **:** [**public**] **sc_module**

sc-module-member-specification ::=
sc-module-member-declaration [sc-module-member-specification]
| access-specifier **:** [sc-module-member-specification]

sc-module-member-declaration ::=
member-declaration
| sc-signal-dclaration
| sc-sub-module-declaration
| sc-module-constructor-definition
| sc-module-constructor-declaration
| sc-has-process-declaration

sc-signal-declaration ::=
sc-signal-key **<** type-specifier > signal-declarator-list **;**
| sc-resolved-key signal -declarator-list **;**
| sc-resolved-vector-key < constant-expression > signal -declarator-list **;**

signal-declarator-list ::=
identifier
| signal-declarator-list **,** identifier

| **sc_in_clk**
| **sc_out_clk**
| **sc_inout_clk**

sc-resolved-key ::=
sc_signal_resolved
| **sc_in_resolved**
| **sc_out_resolved**
| **sc_inout_resolved**

sc-resolved-vector-key ::=
sc_signal_rv
| **sc_in_rv**
| **sc_out_rv**
| **sc_inout_rv**

sc-sub-module-declaration ::=
id-expression [*****] identifier **;**
sc-module-constractor-declaration ::=
SC_CTOR(identifier **) ;**
| identifier **(sc_module_name** [identifier] [**,** parameter-declaration-list] **) ;**

sc-module-constructor-definition ::=
SC_CTOR(identifier **)** [ctor-initializer] sc-module-constructor-body

| identifier **(** **sc_module_name** identifier [**,** parameter-declaration-list] **)** **:** **sc_module** **(** identifier **)** [**,** mem-initializer-list] sc-module-constructor-body

sc-module-constractor-body ::=
{ [sc-module-constractor-element-seq] **}**

sc-module-constractor-element-seq ::=
sc-module-constractor-element
| sc-module-constractor-element-seq sc-module-constractor-element

sc-module-constractor-element ::=
sc-module-instantiation-statement
| sc-port-binding-statement
| sc-process-statement

sc-module-instantiation-statement ::=
identifier **= new** [**::**] [nested-name-specifier] class-name **(** string_literal **) ;**

sc-port-binding-statement ::=
sc-named-port-binding-statement **;**
| sc-positional-port-binding-statement **;**

sc-named-port-binding-statement ::=
identifier -> id-expression **(** id-expression **) ;**
identifier **.** id-expression **(** id-expression **) ;**

sc-positional-port-binding ::=
[*****] identifier **(** identifier-list **)**

identifier-list ::=
id-expression
| identifier-list id-expression

sc-process-statement ::=
SC_METHOD (identifier **) ;** sensitivity-list| **SC_CTHREAD (** identifier **,** sc-event **) ;** [sc-watching-statement]

sc-process-definition ::=
void sc-process-id **( )** sc-process-body

sc-process-id ::=
identifier
| template-id
| [**::**] nested-name-specifier [**template**] identifier
| [**::**] nested-name-specifier [**template**] template-id

sc-process-body ::=
sc-method-body
| sc-cthread-body

sc-method-body ::=
compound-statement

sc-sensitivity-list ::=
sc-sensitivity-clause
| sc-sensitivity-list sc-sensitivity-clause

sc-sensitivity-clause ::=

sensitive (sc-event **) ;**
| **sensitive_pos (** identifier **) ;**
| **sensitive_neg (** identifier **) ;**
| **sensitive** sc-event-stream **;**
| **sensitive_pos** sc-event-stream **;**
| **sensitive_neg** sc-event-stream **;**

sc-event-stream ::=
<< sc-event
| sc-event-stream << sc-event

sc-identifier-stream ::=
<< identifier
| sc-identifier-stream << identifier

sc-event ::=
identifier
| identifier **. pos ( )**
| identifier **. neg ( )**

sc-watching-satement ::=
watching (identifier **. delayed ( ) ==** sc-watching-condition **) ;**

sc-watching-condition ::=
boolean-literal
| logic-literal

sc-cthread-body ::=
compound-statement **wait ( ) ; while (true) {** compound-statement **}**

sc-has-process-declaration ::=
SC_HAS_PROCESS(identifier **) ;**

A.9 Derived classes

base-clause ::=
: base-specifier-list

base-specifier-list ::=
base-specifier
| base-specifier-list **,** base-specifier

base-specifier ::=
[**::**] [nested-name-specifier] class-name
| virtual [access-specifier] [**::**] [nested-name-specifier] class-name
| access-specifier [virtual] [**::**] [nested-name-specifier] class-name

access-specifier ::=
private
| **protected**
| **public**

A.10 Special member functions

conversion-function-id ::=
operator conversion-type-id

converion-type-id ::=
    type-specifier-seq [conversion-declarator]

conversion-declarator ::=
    ptr-operator [conversion-declarator]

ctor-initializer ::=
    : mem-initializer-list

mem-initializer-list ::=
    mem-initializer
    mem-initializer **,** mem-initializer-list

mem-initializer ::=
    mem-initializer-id **(** [expression-list] **)**

mem-initializer-id ::=
    [**::**] [nested-name-specifier] class-name
    | identifier

A.11 Overloading

operator_function_id ::=
    operator operator

operator ::= *one of*

~~**new**~~	~~**delete**~~	~~**new[]**~~	~~**delete[]**~~					
+	-	*	/	**%**	^	**&**	|	~
!	=	<	>	+=	-=	*=	/=	**%=**
^=	**&=**	|=	<<	>>	>>=	<<=	==	**!=**
<=	>=	**&&**	||	++	--	**,**	->*	->
()	**[]**							

A.12 Templates

template-declaration ::=
    [**export**] **template** < template-parameter-list > declaration

template-parameter-list ::=
    template-parameter
    | template-parameter-list **,** template-parameter

template-parameter ::=
    type-parameter
    | parameter-declaration

type-parameter ::=
    class [identifier]
    | **class** [identifier] = type-id
    | **typename** [identifier]
    | **typename** [identifier] = type-id
    | **template** < template-parameter-list > **class** [identifier]
    | **template** < template-parameter-list > **class** [identifier] = id-expression

template-id ::=
    template-name < [template-argument-list] >

template-name ::=

identifier

template-argument-list ::=
  template-argument
  | template-argument-list **,** template-argument

template-argument ::=
  assignment-expression
  | type-id
  | id-expression

explicit-instantiation ::=
  template declaration

explicit-specification ::=
  template **< >** declaration

A.13 Exception handling

Exception handling cannot be used for synthesis coding.

~~try_blick ::=~~
  ~~**try** compound_statement handler_seq~~

~~function_try_block ::=~~
  ~~**try** [ctor_initializer] function_body handler_seq~~

~~handler_seq ::=~~
  ~~handler [handler_seq]~~

~~handler ::=~~
  ~~**catch {** exception_declaration **}** compound_statement~~

~~exception_declaration ::=~~
  ~~type_specifier_seq declarator~~
  ~~| type_specifier_seq abstract_declarator~~
  ~~| type_specifier_seq~~
  ~~| **...**~~

~~throw_expression ::=~~
  ~~**throw** [assignment_expression]~~

~~exception_specification ::=~~
  ~~**throw (** [type_id_list] **)**~~

~~type_id_list ::=~~
  ~~type_id~~
  ~~| type_id_list **,** type_id~~

A.14 Preprocessing directivesb

All preprocessing directives of C++ are acceptable for synthesis coding.

preprocessing-file ::=
  [group]

group ::=
  group-part
  | group group-part

group-part ::=
    [pp-token] new-line
  | if-section
  | control-line

if-section ::=
    if-group [elif-groups] [else-group] endif-line

if-group ::=
    # **if** constant-expression new-line [group]
  | # **ifdef** identifier new-line [group]
  | # **ifndef** identifier new-line [group]

elif-groups ::=
    elif-group
  | elif-groups elif-group

elif-group ::=
    # **elif** constant-expression new-line [group]

else-group ::=
    # **else** new-line [group]

endif-line ::=
    # **endif** new-line

control-line ::=
    # **include** pp-tokens new-line
  | # **define** identifier replacement-list new-line
  | # **define** identifier lparen [identifier-list] replacement-list new-line
  | # **undef** identifier new-line
  | # **line** pp-tokens new-line
  | # **error** [pp-tokens] new-line
  | # **pragma** [pp-tokens] new-line
  | # new-line

lparen ::=
  the left_parenthesis character without preceding white_space

replacement-list ::=
  [pp-tokens]

pp-tokens ::=
    preprocessing-token
  | pp-tokens preprocessing-token

new-line ::=
    the new_line character

Annex B Glossary (informative)

behavioral level: A design level which has no detail of hardware-resource and operating-schedule. Today, we expect that there is one clock signal for event trigger. We have to describe an algorithm and interface between outside and inside of module as ports.

behavioral synthesis: A synthesis from a behavioral level design to RTL level design or gate level design. A behavioral synthesis tool works for resource sharing and scheduling. Resource means hardware which are memories, registers, combinational circuits and so on. Scheduling means resource assignment of each operation to each clock cycle and hardware.

class: The same term as the one used in C++. In SystemC, the class mechanism is used for module definition and the object types defined for SystemC. The class except module is limited the usage for synthesis.

constructor: The same term as the one used in C++.

clock: A basic signal which triggers hardware events which occurs every fixed period.

cycle: A period of clock.

datatype: A type of signal.

function: The same term as the one used in C++.

initializer: The same term as the one used in C++.

macro: A keyword which is defined in preprocessor

member: The same term as the one used in C++.

method: It is a term of object oriented design. C++ realized it as a member function.

module: A capsulated block which has ports for interface.

named mapping: A way that all ports of a module are binding to signals by their names.

namespace: The same term as the one used in C++.

operator: The same term as the one used in C++.

overload: The same term as the one used in C++.

pointer: The same term as the one used in C++.

port: A interface signal which connects between inside and outside of module.

positional mapping: A way that all ports of a module are binding to signals by their describing positions.

reset: A signal which indicates that registers have to become initial value.

process: A special function which is triggered by sensitivity signals in module. There are three kind of process in SystemC, which are SC_METHOD, SC_THREAD and SC_CTHREAD.

register transfer level(RTL): A design level which has the description of register and combination logic. The design should be cleared the schedule of each cycle operation and register recouces.

RTL synthesis: A synthesis from a RTL level design to a gate level design. A RTL synthesis tool works for logic synthesis which is mainly solving of boolean algebra .

signal: A object which is declared as sc_signal.

submodule: A module which is called in a module. The submodule works as the part of the calling module.

template: The same term as the one used in C++.

Annex C   Index