SystemC Synthesizable Subset

1.3 draft

Described:
Synthesis Working Group of Open SystemC Initiative

History:
Draft 1.1.21 March 31, 2005
Draft 1 Febrary 16, 2005
Draft 1.1.22 May 10, 2007
Draft 1.1.22 August 31, 2007
Draft 1.1.22 September 19, 2007, amended by Andres Takach, Jos Verhaegh and Alan Su
Draft 1.1.23 Febuary 12, 2008, amended by Jos Verhaegh and Alan Su
Draft 1.1.23 May 7, 2008, cleanup by Andres Takach
Draft 1.1.23 May 8, 2008, cleanup and Section 1 editing by Alan Su
Draft 1.1.23 May 14, 2008, spellcheck/minor edits by Andres Takach
Draft 1.1.23 June 25, 2008, edits proposed by email 05-15-08 by Andres Takach
Draft 1.1.23 Oct 27, 2008. Minor edits to Sections 5.2.3 and 5.2.5.2 by Andres Takach
Draft 1.1.24 Dec 09, 2008. Accepted tracked changes.
Draft 1.2 Dec 10, 2008. Added language for sc_exports.
Draft 1.3 June 30, 2009. Eliminate discussion of process control extensions by Mike Meredith

Introduction

This standard describes a standard syntax and semantics for SystemC synthesis. It defines the subset of SystemC that is suitable for RTL/behavioral synthesis and defines the semantics of that subset for the synthesis domain. This standard is based on the C++ standard and the IEEE 1666 SystemC standard.

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 the Draft 2.0 effort:
Mike Meredith
Benjamin Carrion Schafer
Alan P. Su
Andres Takach, Chair
Jos Verhaegh

The following team members drove the Draft 1.0 effort:
Eike Grimpe
Rocco Jonack
Masamichi Kawarabayashi, past 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

SYSTEMC SYNTHESIZABLE SUBSET 1

1.3 DRAFT 1

INTRODUCTION I

1 OVERVIEW 1

1.1 SCOPE 1
1.2 PURPOSE 1
1.3 TERMINOLOGY 2
1.4 CONVENTIONS 3
1.5 ABSTRACTION LEVELS 4
1.6 ESL SYNTHESIS 12

2 TRANSLATION UNITS AND THEIR ANALYSIS 16

3 MODULES 17

3.1 MODULE DEFINITIONS 17
3.2 DERIVING MODULES 19

4 DATATYPES 21

4.1 FUNDAMENTAL TYPES 21
4.2 COMPOUND TYPES 26
4.3 CV-QUALIFIERS 27
4.4 SYSTEM C DATATYPES 27

5 DECLARATIONS 39

5.1 DECLARATIONS 39
5.2 DECLARATORS 41

6 EXPRESSIONS 42

7 FUNCTIONS 43

7.1 FUNCTION DEFINITIONS 43
7.2 FUNCTION BODY 43

8 STATEMENTS 45

8.1 LABELED STATEMENT 45
8.2 COMPOUND STATEMENT 45
8.3 WAIT STATEMENT 45
8.4 SIGNAL ASSIGNMENT STATEMENT 45
8.5 SELECTION STATEMENTS 45
8.6 ITERATION STATEMENTS 46
8.7 JUMP STATEMENTS 46
8.8 DECLARATION STATEMENT 46

9 PROCESSES 47

9.1 CLOCK 47
9.2 SC_METHOD 48
9.3 SC_CTHREAD 49

9.4 SC_THREAD 50
10 SUBMODULE INSTANTIATION 51
11 NAMESPACES 53
12 CLASSES 53
12.2 DERIVED CLASSES 55
12.3 MEMBER ACCESS CONTROL 55
12.4 SPECIAL MEMBER FUNCTIONS 55
13 OVERLOADING 58
14 TEMPLATES 61
15 PREPROCESSING DIRECTIVES 64
16 LEXICAL ELEMENTS 65
17 SCOPE AND VISIBILITY 66
18 MISCELLANEOUS 67
18.1 TRACING 67
18.2 OUTPUTTING MESSAGES TO STDOUT AND/OR COUT 67
18.3 EXCEPTION HANDLING 67
ANNEX A SYNTAX SUMMARY (INFORMATIVE) 68
A.1 KEYWORDS 68
A.2 LEXICAL CONVENTIONS 68
A.3 BASIC CONCEPTS 72
A.4 EXPRESSIONS 72
A.5 STATEMENTS 75
A.6 DECLARATIONS 76
A.6-1 SYSTEMC TYPE SPECIFIERS 78
A.7 DECLARATORS 79
A.8-1 MODULE DECLARATION 81
A.9 DERIVED CLASSES 84
A.10 SPECIAL MEMBER FUNCTIONS 84
A.11 OVERLOADING 85
A.12 TEMPLATES 85
A.14 PREPROCESSING DIRECTIVESB 86
ANNEX C GLOSSARY (INFORMATIVE) 88
REFERENCE 90

1 Overview

1.1 Scope

Synthesis Working Group (SWG) of Open SystemC Initiative(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 developers to develop SystemC compliant synthesis tools. SSC will be defined within C++ and IEEE 1666 SystemC 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, Functional Specifications, and the Language Reference Manual. Some books for SystemC modeling have also been published. 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, for Logic Synthesis and at the behavioral level, for High Level Synthesis, may be individually described to facilitate each design phase.
Figure 1.1, on the next page, shows an abstract view of a design flow involving both Logic and High Level synthesis

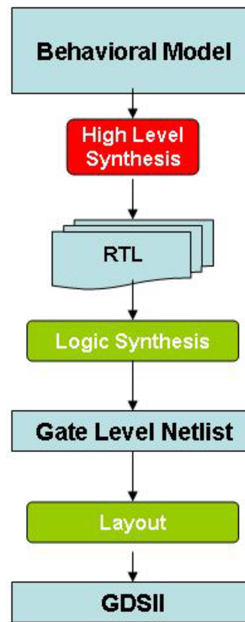


Figure 1.1 Abstract view of design flow involving High Level and Logic synthesis

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 a 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 document 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. text) refers to syntax that shall be ignored.

e) Syntax test that is shadowed (e.g. text) refers to syntax that shall be extended.

f) Lowercase words in roman font, some containing embedded hyphens, are used to denote syntactic categories, for example:
nested-namespace-specifier

g))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.

h) 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 (,).

i) Square braces [ ] enclose an 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

j) Any paragraph starting with "NOTE--" is informative and not part of the standard.

k) 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 Abstraction Levels

How abstraction levels support design activities

The goal of a system-level design methodology is to decrease design cost and design time Firstly, the complexity of modern systems does not allow us to describe implementations directly. Furthermore, it is difficult to create derivative implementations with different functions or different architectures, because functions and architectures cannot be extracted easily from implementations for re-use. Therefore a separation of function, architecture and implementation are necessary when designing a system. Design activities are needed to combine functions and architectures to implementations to decrease design time and design cost.

To be able to separate function, architecture and implementation in a System Design flow abstraction levels need to be defined. The idea is to gradually confront designers with implementation details such as timing and data representations. We distinguish three main levels of abstraction: function level, architecture level and implementation level. Each level has modeling views to be discussed in detail in respective sections. Figure 1.2 shows the abstraction levels in a System Design Flow.

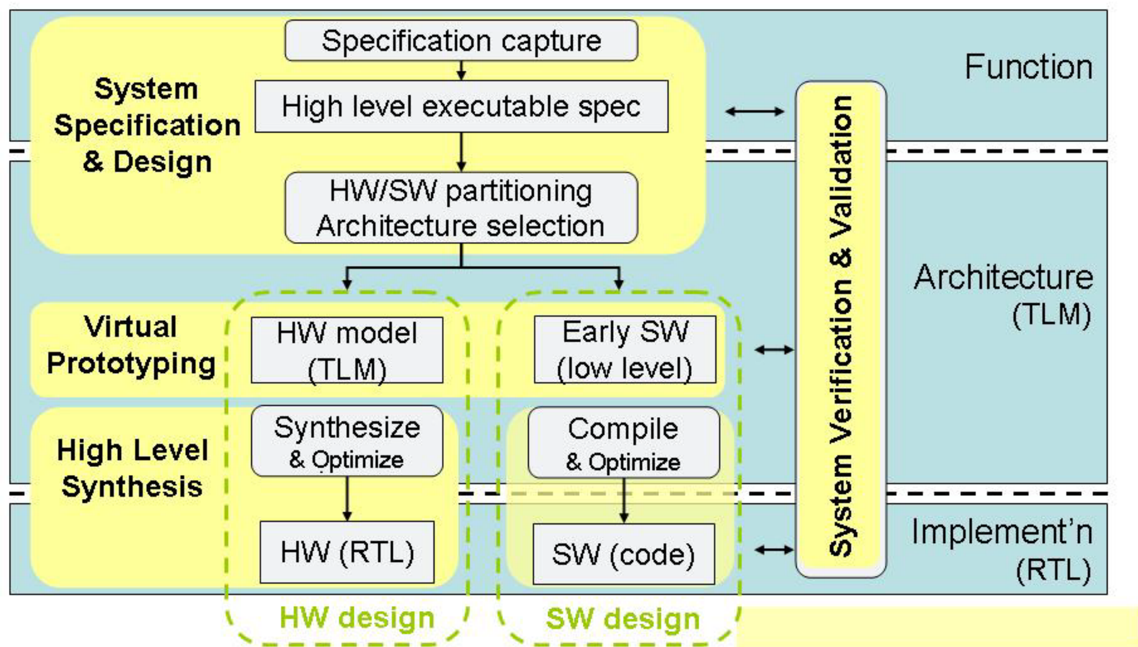


Figure 1.2 Abstraction levels in a System Design Flow

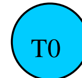
1.5.1 Function Level

The motivation for introducing this level of abstraction is to quickly obtain a function to determine what the system is supposed to do, without making architecture assumptions.

Hence there is the potential to re-use functions either to create derivative functions, or to synthesize different implementations with different architectures.
An example of a view at this level is a function modelled as a process network in YAPI which can be analyzed through simulation.
At the Function Level of abstraction two design steps can be identified:

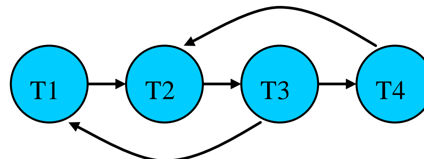
- Algorithm Specification
- Partitioning into communicating tasks

1.5.1.1 Function Level: Algorithm Specification



In this design step an executable functional specification of the algorithm is created. (e.g. in C/C++/Matlab code). This executable specification is used to check the validity of the algorithm. The simulation in this design step is sequential, it has no timing information and it has a single thread of control. The simulation speed is high due to lack of timing and architecture details
Profiling techniques are used to obtain an initial estimate of the computational load of the different functions and the amount of data transfer between them.
Code inspection is used to estimate the amount of flexibility required for each of the functions. The results of both, code inspection and profiling, are used as input for task partitioning and, in a later stage as input for Hardware/Software partitioning.
Next to algorithm verification the executable functional specification generated in this step is also used as a golden reference model throughout the whole flow.

1.5.1.2 Function Level: Partitioning into communicating tasks



With the design constraints and requirements and a suitable architecture template in mind and the results from the previous algorithm design step the system is partitioned into tasks that perform processing functions and channels through which data are communicated between these tasks.
The processes in such a network are concurrent and are connected by communication channels. Processes produce data elements and send them along a unidirectional communication channel where they are stored in a first-in-first-out order until the destination process consumes them. A network, as described above, is also referred to as a Kahn Process Network (KPN) [3].
In KPN, parallelism and communication are explicitly modeled, which is essential for the mapping onto multi-processor systems. Another property of KPN is that an application designer can combine processes into networks without specifying their order of execution. This property stimulates the modular construction and reuse of applications (functional IP), since it is easier to compose new applications using existing ones.
Using a multi-threaded simulation the communication load on the channels and the computation load on the tasks are analyzed. If necessary the system can be repartitioned to

meet the constraints and requirements. Also the functional correctness of the partitioning is checked during the multi-threaded simulation.
To model signal processing applications as Kahn Processing Networks YAPI [4,5] can be used. The purpose of YAPI is to enable the reuse of signal processing applications and the mapping of signal processing applications onto heterogeneous systems that contain hardware and software components.
YAPI has also been embedded in SystemC. YAPI embedded in SystemC is developed as a SystemC class library with a set of rules that can be used to model stream processing applications as a process network. As mentioned above the model of computation in YAPI is based on KPN.

1.5.2 Architecture Level

The motivation for introducing this level of abstraction is to quickly find an efficient implementation. Efficiency can be defined in terms of power, timing, area, etc. To be able to quickly evaluate the efficiency of alternative implementations, we want to avoid the effort of making them in detail. For example the decision to base an implementation on a message passing or a shared memory architecture leads to two alternative implementations.

Transaction Level Modeling (TLM) is developed for abstract modeling of (SoC) systems at architecture level allowing efficient system exploration. Literally a transaction is *the exchange of goods, services or funds*; or *a communicative action or activity involving two parties or things that reciprocally affect or influence each other* (Merriam-Webster Online Dictionary). Both meanings have two ingredients, exchange/communication and goods/influence. In an electronic system the goods or influence can be considered as the computation (goods) or the effect of the computation (influence). There are many discussions regarding TLM over the years, here we use only definitions, terminologies and libraries developed by the OSCI TLM Working Group (TLM WG).

1.5.2.1 Transaction Level Modeling

Although TLM includes computation and communication, OSCI TLM 1.0 and 2.0 discuss only the communication part. In the physical form, a transaction is a payload, the data structure that passed between modules. By definitions from TLM WG, we have following modeling views (or called modelling styles):

1. Un-Timed (UT): A modeling style in which there is no explicit mention of time or cycles, but which includes concurrency and sequencing of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes and blocking FIFOs. Some users adopt the practice of inserting random delays into untimed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.
2. Loosely Timed (LT): A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely timed model may include timer models and a notional arbitration interval or execution slot length. Some users adopt the practice of inserting random delays into loosely timed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.

3. Approximately Timed (AT): A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined.
4. Cycle Accurate (CA): A modeling style in which it is possible to predict the state of the model in any given cycle at the external boundary of the model and thus to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model in each cycle, but which is not required to explicitly re-evaluate the state of the entire model in every cycle or to explicitly represent the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles.

UT modules are synthesizable. LT modules use absolute time for timing information and thus not synthesizable. For example we cannot synthesize sc_time(10, SC_NS) which is used to represent the latency to execute a certain function. AT modules are not synthesizable as well because states and state transitions modelled using AT are not precise in timing. It is still uncertain if cycle accurate modules are synthesizable because TLM WG has not developed how to model cycle accurate in TLM 2.0. For further details please refer to TLM 2.0.

1.5.3 Implementation Level

This abstraction level captures the details of the interfaces and the IO functionality including a full or partial specification/modelling of their timing. The communication among of blocks is carried out at the signal-level. The specification of the interface is pin-accurate and should be preserved by synthesis for the top-level module. Implementation levels include Register-Transfer Level (RTL), Gate Level and Behavioral Level. RTL and Gatel Level are widely used and have traditionally written in HDL languages such as verilog/VHDL and SystemVerilog. The abstraction level below gate level is expressed in the GDSII format. SystemC is not suitable for this abstraction level.

The Gate Level consists of interconnection of instantiations of technology leaf cells. The specification is structural. The behaviour of each cell is written for simulation and in general is quite simple. For example, combinational gates are typically written in the form of concurrent statements. Sequential gates include registers and memories are usually written in the same form as it is done at the RTL level.

The RTL level allows the specification of both structure and more behavioral constructs:

- In addition to bit-wise logic, word-level arithmetic, such as a * b can be specified and synthesized.
- Loops with constant number of iterations can be specified. Such loops are fully unrolled.
- A finite-state machine (FSM) can be specified in a way where synthesis can recognize it as an FSM and perform optimizations such as state encoding etc. The computation of the next-state and the output is done with behavioral constructs such as if-then-else and case statements.
- A finite-state machine, where states and transitions may contain complex logic and arithmetic behaviour (not just simple constant assignments to outputs). This is called an explicit-state machine. Registers may be specified either inside or outside the explicit-state machine.

The interface of RTL sub-blocks may be changed by synthesis (boundary optimization), but the top-level interface is preserved. Clock, reset and enable behavior is explicitly specified. Internal cycle timing of operations maybe changed in limited ways (retiming) under user control.

The verification methodology of the output from RTL synthesis against the reference RTL specification is well defined for both combinational and sequential hardware. For instance IEEE Standard 1076-2004 defines this for VHDL and the same methodology is applicable for SystemC RTL specifications.

The behavioral-level introduces some freedom in how operations and IO are scheduled by only partially constraining the cycle-by-cycle behaviour of the IO. Registers are not explicitly defined, but instead are determined by synthesis. Storage requirements are dependent on how operations are scheduled: registers are used to store values that are used one or more cycles after the cycle in which they are generated. Storage of arrays may be mapped to memories or to registers. The specification of behaviour is in the form of an implicit-state machine rather than the explicit-state machine generally used for RTL. In an implicit-state machine, there is no explicit state variable that is used to select what behaviour is executed next. Instead, the behaviour consists of a process that is sensitive to the clock and possibly a reset signal (for asynchronous resets). The process uses language constructs such as loops, constructs to continue and exit loops and constructs to specify conditional behaviour (if-then-else and case constructs) and wait statements that specify cycle timing among sets of output assignments.

The output from behavioral synthesis is a synthesizable RTL description and/or a Gate-Level description. The verification methodology of the generated specifications against the behavioral (source) specification is more complex (than the RTL level vs. Gate-Level specification) since the cycle-by-cycle behaviour may be changed by synthesis.

To devise the discussion in implementation level, let equation $Y = P(X)$ denotes the target process where P is the function of the target process. $X = \{x_1, x_2, \dots, x_n, v_1, v_2, \dots, v_m\}$ be the set of input signals where x_i, $1 \leqq i \leqq n$, are signals in the sensitivity list and v_j, $1 \leqq j \leqq m$, are signals not in the sensitivity list. And $Y = \{y_1, y_2, \dots, y_l\}$ be the set of output signals.

. *Example*:

```
SC_MODULE( AddMul_1 ) {
      sc_in< sc_clock > clk;
      sc_in< sc_uint<16> > a, b, c;
      sc_out< sc_uint<32> > result;

      void addmul_1() {
            result = a + (b * c);
      }

      SC_CTOR( AddMul_1 ) {
            SC_METHOD( addmul_1 );
            sensitive << clk.pos();

      }
};
```

In the above example P is the function `void addmul_1();` $X = \{x_1, v_1, v_2, v_3\}$ where $x_1 =$ `clk`, $v_1 =$ `a`, $v_2 =$ `b` and $v_3 =$ `c`; $Y = \{y_1\}$ where $y_1 =$ `result`.

1.5.3.1 Behavioral Level

Behavioral level is also known as functional level or behavioral architectural-level as called in [De Micheli 94, Knapp 96]. At behavioral level, the equation $Y = P(X)$ both X and Y contain no time. As soon as any x_i of X changes value then P computes Y at exactly the same instance. Different from UT, only a non-empty subset of X is in the sensitivity list. A virtual clock can be the only one or one of the x_i of X. When the virtual clock triggers the process, P computes Y based on X at exactly the same instance.

. *Example*:

```
SC_MODULE( AddMul_2 ) {
      sc_in< sc_uint<16> > a, b, c;
      sc_out< sc_uint<32> > result;

      void addmul_2() {
            result = a + (b * c);
      }

      SC_CTOR( AddMul_2 ) {
            SC_METHOD( addmul_2 );
            sensitive << a << b << c;
      }
};
```

In the above example the process function P is `addmul_2()`; the input set is $X = \{x_1, x_2, x_3\}$ where x_1 = a, x_2 = b and x_3 = c; the output set is $Y = \{y_1\}$ where y_1 = `result`.

. *Example*:

```
SC_MODULE( AddMul_3 ) {
      sc_in< sc_clock > clk;
      sc_in< bool > rst;
      sc_in< sc_uint<16> > a, b, c;
      sc_out< sc_uint<32> > result;

      void addmul_3() {
            result = 0;
            wait();
            while (1) {
                  result = a + (b * c);
                  wait();
            }
      }

      SC_CTOR( AddMul_3 ) {
            SC_CTHREAD( addmul_3, clk.pos() );
            reset_signal_is(rst, false);

      }
};
```

In the above example the process function P is the combination of `void addmul_3()` and semantics of `SC_CTHREAD` and `reset_signal_is()`. Furthermore `clk` and `rst` ports are identified as sensitive events and `clk` is the clock port and `rst` the reset port. The input set is $X = \{x_1, x_2, v_1, v_2, v_3\}$ where x_1 = `clk`, x_2 = `rst`, v_1 = a, v_2 = b and v_3 = c. The output set

is $Y = \{y_1\}$ where y_1 = `result`. The semantics of SC_CTHREAD and reset_signal_is() will be described later in Section 9.4.

1.5.3.2 Register Transfer Level

Register Transfer Level (RTL), as the name suggested, describes functions and signals from registers to registers. The basic elements of this level are combinational and sequential functional/logic units, registers and signals. Notice that a physical circuit contains no registers cannot be modelled at RTL. It can only be modelled at gate level and below.
At RTL:

1. Time does not necessarily in the unit of cycles. In some cases it is in a time unit like picosecond or nanosecond.
2. Time is an implicit factor of $Y = P(X)$. Time is not considered as input nor output signals. Instead time is embedded in P to compute Y. P implicitly describes how much time each atomic operation, i.e. addition, multiplication, subtraction, and, or, etc., will take to evaluation how much time in total to compute Y.
3. The physical clock port must be in the sensitivity list.

An RTL module has a Finite State Machine (FSM) which describes cycle-by-cycle behaviour of the target module. The functional behaviour of each state can be described inside the FSM. We call this kind of FSM the FSM with Datapath (FSMD). Or the functional behaviour is described using a separate datapath which is then controlled by the FSM.
. *Example*:

```
SC_MODULE( AddMul_4 ) {
      sc_in< sc_clock > clk;
      sc_in< bool > rst;
      sc_in< sc_uint<16> > a, b, c;
      sc_out< sc_uint<32> > result;

      void addmul_4() {
            sc_signal<sc_uint<32> > tmp1;

            tmp1 = 0;
            result = 0;
            wait();
            while (1) {
                  tmp1 = b * c;
                  wait();
                  result = a + tmp1;
                  wait();
            }
      }

      SC_CTOR( AddMul_4 ) {
            SC_CTHREAD( addmul_4, clk.pos() );
            reset_signal_is(rst, false);

      }
};
```

In the above example the process function P = `void addmul_4()` is a cycle-by-cycle FSMD with reset state, i.e. the `if(rst == false){}` block, and a 2-state, register-to-register computation unit, i.e. the `while(1){}` block, and registers are inherited state

registers and `tmp1`. The cycle time is implicitly described in the combined semantics of `SC_CTHREAD`, `reset_signal_is()` and `wait()` statements. The clock port, `clk`, is specified as a sensitive input using `SC_CTHREAD()`.

1.5.3.3 Gate Level

At this level the basic elements are logic gates, i.e. `AND`, `OR`, `NOT`, `XOR`, etc., and signals that connect gates. To model at this level we must first have a logic gate library so one can instantiate gates needed then declare proper signals to glue logic gates together.
. *Example*:

```
SC_MODULE( AND2 ) {
      sc_in< sc_uint<1> > a, b;
      sc_out< sc_uint<1> > c;

      void and2() {
            c = a & b;
      }

      SC_CTOR( AND2 ) {
            SC_METHOD( and2 );
            sensitive << a << b;
      }
};

SC_MODULE( OR2 ) {
      sc_in< sc_uint<1> > a, b;
      sc_out< sc_uint<1> > c;

      void or2() {
            c = a | b;
      }

      SC_CTOR( OR2 ) {
            SC_METHOD( or2 );
            Sensitive << a << b;
      }
};

SC_MODULE( ANDOR ) {
      sc_in< sc_uint<1> > a, b, c;
      sc_out< sc_uint<1> > d;

      void andor() {
            AND2 *andgate;
            R2   *orgate;
            sc_signal<sc_uint<1> > wire1, wire2, wire3, wire4;

            andgate = new AND2("andgate");
            orgate = new OR2("orgate");
            a(wire1);
            b(wire2);
            andgate->a(wire1);
            andgate->b(wire2);
            andgate->c(wire3);
```

```
            orgate->a(wire3);
            orgate->b(wire4);
            d(wire4);
            while (1) {
                  wait();     // forever loop
            }
      }

      SC_CTOR( ANDOR ) {
            SC_THREAD( andor );
            sensitive << a << b << c;

      }
};
```

In above example we first build `AND2` and `OR2` gates to be used as library components in the ANDOR module. It is interesting to note that the modeling style of `ANDOR` is different from that of `AND2` and `OR2`. In `ANDOR` gates and wires are instantiated then connected. There is no any computational or controlling description in `void addor()`. However in `AND2` and `OR2` the modeling style fits the description of the behavioral level. For such primitive components one do can argue that their behavioral descriptions are actually at gate level.

1.6 ESL Synthesis

1.6.1 Introduction

The conventions described in this chapter are focused on synthesis solutions that generate non-programmable RTL cores.

High Level Synthesis, also known as *Behavioral Synthesis*, the second level of the ESL Synthesis, allows design at higher levels of abstraction by automating the translation and optimization of a behavioral description, or high-level model, into an RTL implementation. It transforms un-timed or partially timed functional models into fully timed RTL implementations.

Because a micro-architecture is generated automatically, designers can focus on designing and verifying the module functionality. Design teams create and verify their designs in less time because it eliminates the need to fully schedule and allocate design resources, as done with existing RTL methods. This behavioral design flow increases design productivity, reduces errors, and speeds up verification. Figure 1.3 shows an abstract view of a design flow involving both Logic and High Level synthesis

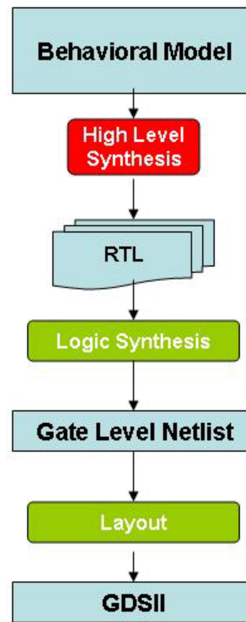


Figure 1.3 Abstract view of design flow involving High Level and Logic synthesis

A typical high level synthesis process incorporates a number of complex stages. This process starts with a high-level language description of a module's behavior, including I/O actions and computational functionality. Several algorithmic optimizations are performed to reduce the complexity of a result and then the description is analyzed to determine the essential operations and the dataflow dependencies between them.
The other inputs to the high level synthesis process typically include a target technology library, characterized at a target frequency for the selected fabrication process, and a set of directives that will influence the resulting architecture. The directives include, for example, timing constraints used by the algorithms of the tool, as they create a cycle-by-cycle schedule of the required operations. The characterized library is used when allocation and binding occurs, in order to assign these operations to specific functional units such as adders, multipliers, comparators, etc.
Finally, a state machine is generated that will control the resulting datapath implementation of the desired functionality. The datapath and state machine outputs are in RTL code, optimized for use with conventional logic synthesis or physical synthesis tools.

1.6.2 Vision

Easier management of system complexity, accelerated design verification and implementation, increased opportunity for design reuse, and wider selection of implementation options, these are just a few reasons why project teams are moving to ESL design. However, by moving to the higher levels of abstraction a design gap between ESL and RTL has come into existence. For each algorithm modelled at abstract level, there are numerous ways it can be realized in hardware. However, with tight schedules and increasing complexity, there simply is not enough time to create more than one RTL implementation by hand after an algorithm and architecture choice is made. This way alternative hardware implementation options that could significantly impact performance, area, or power are seldom created or evaluated.

With the availability of a general-purpose language based on C++, like SystemC, and the maturity of high level synthesis and verification tools, high-level models can be leveraged to

help evaluate trade-offs in architectures and algorithms in a way not previously available. Summarizing, high level synthesis enables the transition to high-level design by closing the gap between ESL and RTL flows.

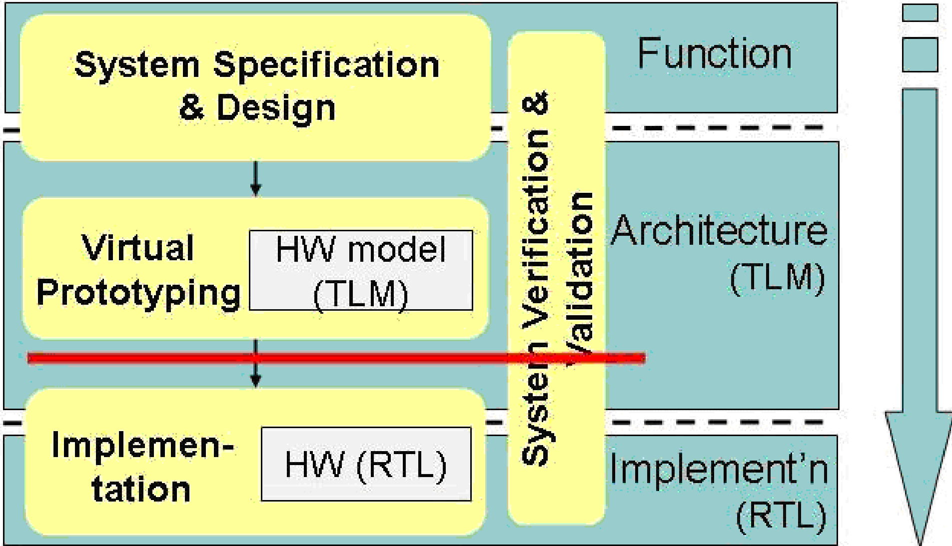


Figure 1.4 The vision from architecture (TLM) to implementation (RTL)

Figure 1.4 shows the vision to synthesize from architecture to RTL. Components in the architecture are described in synthesizable SystemC and interfaced and connected using the TLM library. As mentioned earlier HLS is the second layer of ESL Synthesis, the first layer is architecture synthesis. Architecture synthesis reads in architectures described using TLM library and decide memory infrastructure, cache structure, DMA structure, bus layers, I/O devices, CPU selections, etc. Architecture synthesis and HLS together comprise the ESL Synthesis. While synthesizing TLM semantic is the first step toward architecture synthesis, it is a future task of Synthesis WG to define the synthesizable subset of the TLM library.

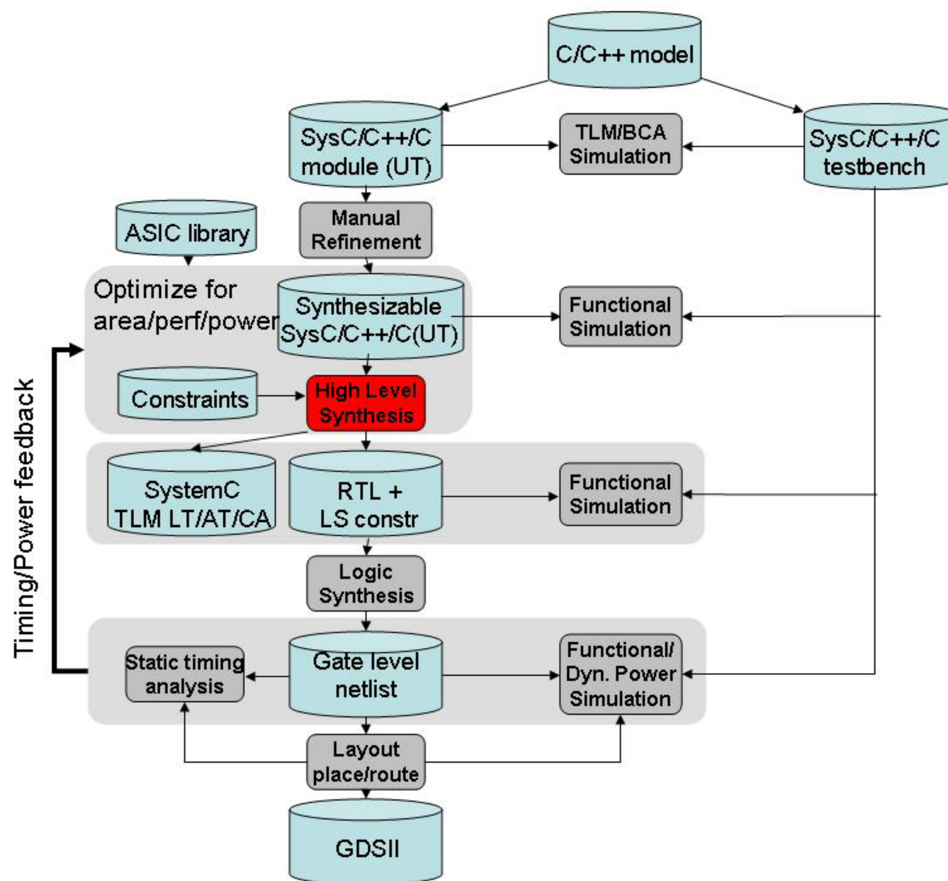


Figure 1.5 Abstract view of possible implementation flow from ESL down to GDSII

Figure 1.5 shows a possible implementation flow from ESL down to GDSII incorporating ESL Synthesis. Based on the diagram a wish list of requirements for ESL Synthesis solutions, based on C/C++/SystemC can be defined:

- automate the RTL implementation from behavioral C/C++/SystemC (TLM) models
- support a fast design time (implementation & Verification)
  - At least a 2x improvement compared to hand coded design
- support optimization for Performance, Area and Power
  - On par or better area figures for a required performance compared to hand coded
  - Automatic timing closure based on back-annotated static timing analysis
  - Automatic power optimization based on dynamic power simulation
- support generation of behavioral SystemC TLM models with timing annotation (LT/AT) and SystemC CA models
- automate reuse of high level test environment for verification of RTL implementation, netlist and generated SystemC TLM views
  - Support of assertions
  - Support for functional equivalence checking
- support synthesis from and generation of TLM 2.0 SystemC compliant models
- Full integration of high level synthesis methodology in System Level Design Environment

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 or **sc_main** function.

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

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

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.

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

Ports represent the externally visible interface to a module and are used to transfer data into and out of the module. Ports can be declared using sc_in, sc_out and sc_inout constructs.

3.1.2.2 Signals

Signals can be used to keep values and interface between processes.

3.1.2.3 Exports

Exports represent the externally visible interface to a module and are used to transfer data into and out of the module. Exorts can be declared using sc_export construct.

3.1.2.4 Module constructor

Every module declaration must contain exactly one declaration or definition of a constructor method. Sub module 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, through the use of the SC_CTOR macro, or through the explicit derivation of a module class from sc_module. Both ways are supported for synthesis.

Where the 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.

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**'.

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 initialization:

```
    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**() statement in its body.

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

4.1.1 Integer Types

The following integer 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 integer types with the exception of 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 integer 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

Integer 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 integer 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 integer types provided the I/O behavior of the design is unchanged.

4.1.1.2 Bit sizes

The bit sizes for the different integer 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 integer 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 integer types to have the same storage. ISOC++ Section 3.9.1 also constrains the relative sizes of the different integer types. The minimum sizes are derived from the minimum required numerical limits as specified by ISOC Section 5.2.4.2.

Integer 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 integer 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 integer 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 integer types that are applied in the evaluation of expressions: integer promotions, and usual arithmetic conversions. An example of an integer 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 (integer promotion that is performed as part of the usual arithmetic conversion) and then converted to long long.

4.1.2.1 Integer Promotions

- The rules for integer promotions are identical to those defined for C++ (ISOC++ Section 4.5).

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++ (ISOC++ Section 5) 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 for synthesis are defined as follows:

- First, Integer 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 unsigned int
- Otherwise, no conversion 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 integer 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)
```

4.1.3.2 Operator: +a (unary plus)

The type of the result is that of the integer 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 integer 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 a **synthesis refinement** with respect to ISOC++ (Section 5.6). The synthesis behavior is consistent with the behavior specified in ISOC (Section 6.5.5).

4.1.3.7 Operator ~a (bitwise complement of a)

The type of the result is that of the integer 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 integer 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.
Synthesis of the floating-point types used in C++ presents verification challenges. The simulation of the generated hardware has to be compared to the C++ behavior running in a processor. While most platforms conform to the IEEE 754 stardard, this is not sufficient as the IEEE 754 standard allows different implementations to produce different results [10]. The main reason for such differences is that intermediate results may be placed in destinations that are beyond the user's control. For instance a destination of an intermediate floating-point computation may be a 64-bit double in memory or an internal 80-bit register in a processor. Differences in processor architectures and in compiler optimizations may lead to differences in results. If the program uses floating-point comparisons to select what branch of code to execute, the differences may be indeed be very difficult to account for by a verification methodology.

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 ISOC++ standard as element type, excluding pointers, and which are supported for synthesis, or any SystemC data type which is supported for synthesis, and which conforms to the requirements on element types stated in the ISOC++ standard. In addition, the synthesis subset supports declaration of 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 as defined in ISOC++.

4.2.4 Enumerations

Supported as defined in ISOC++.

4.2.5 Pointers to Nonstatic Class Members

Supported subject to the limitations described in section 4.2.2 on pointers.

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 type of any supported type is also supported.

4.4 System C Datatypes

System-C provides a number of datatypes that are useful for hardware design. These datatypes are implemented as 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 and 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 ($W \leq 64$)
- sc_uint<W>: finite precision unsigned integer ($W \leq 64$)

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:

<table>
<tr><th>Op Category</th><th colspan="6">Operators/Methods</th></tr>
<tr><td>Arithmetic</td><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td></td></tr>
<tr><td>Assign</td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td></td></tr>
<tr><td>Unary</td><td>+</td><td>-</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Auto Incr/dec</td><td>++ prefix</td><td>++ postfix</td><td>-- prefix</td><td>-- postfix</td><td></td><td></td></tr>
<tr><td>Bitwise</td><td>&</td><td>|</td><td>^</td><td></td><td></td><td></td></tr>
<tr><td>Assign</td><td>&=</td><td>|=</td><td>^=</td><td></td><td></td><td></td></tr>
<tr><td>Unary</td><td>~</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Relational</td><td>==</td><td>!=</td><td><</td><td><=</td><td>></td><td>>=</td></tr>
<tr><td>Shift</td><td>>></td><td><<</td><td>>>=</td><td><<=</td><td></td><td></td></tr>
<tr><td>Bit Select</td><td>[x]</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Part Select</td><td>(i,j)</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Concatenation</td><td>(,)</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Conv to C integer</td><td>to_int</td><td>to_long</td><td>to_int64</td><td>to_uint</td><td>to_uint64</td><td>to_ulong</td></tr>
<tr><td>Assignment</td><td>=</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Arithmetic, shift and bitwise operations generate intermediate values that do not lose 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 integer 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 integer 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 hardware 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 numbers 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, **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. (**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:

- 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 integer 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 integer 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 arguments 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 integer	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 a signed value 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 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 and 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 a 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 the appropriate sign extension is done.
Note: It is advisable to make the second operand be of unsigned type to guarantee that the inferred shifter is unidirectional. Otherwise, whether a bidirectional or unidirectional shifter 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.,

```
bool x = y[2] & z[5];
x[3] = 1;
```

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)
Index values outside the range [0, W-1] are invalid. Synthesis may assume that all index values are in their valid ranges.

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. Index values outside the range [0, W-1] are invalid (**this behavior is inconsistent with sc_bigint/sc_biguint**). Synthesis may assume that all index values are in their valid ranges. 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 integer	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];
```

Index values outside the range [0, W-1] are invalid. Synthesis may assume that all index values are in their valid ranges.

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. Range bounds outside the range [0, W-1] are invalid. Synthesis may assume that all range bounds are in their valid ranges.

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. Use of linkage specification and inline assemply language constructs are not supported.

Deprecated items

Embedded assembler routines are not supported.

5.1.1 Specifiers

Supported as defined in ISOC++.

5.1.1.1 Storage class specifiers

Restricted support.

1 The specifiers **auto** and **register** are hints to a C++ compiler and may be ignored by synthesis.

2 The specifier **mutable** is supported as defined in ISOC++.

3 The support of the specifier **extern** is limited as described in Section 5.1.4.

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

5.1.1.2 Function specifiers

The **inline** specifier is supported as defined in ISOC++.
The **explicit** specifier is supported as defined in ISOC++.
The **virtual** specifier is supported. Virtual functions are supported with the limitations described in Section 12.2.3

5.1.1.3 The `typedef` specifier

Supported as defined in ISOC++.

5.1.1.4 The `friend` specifier

Supported as defined in ISOC++.

5.1.1.5 Type specifiers

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

5.1.1.6 The cv-qualifiers

Supported as defined in ISOC++.

5.1.1.7 Simple type specifiers

Restricted support.

Deprecated items

Types **w_char_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.

The following SystemC types are supported:

sc_int
sc_uint
sc_bigint
sc_biguint
sc_logic
sc_lv
sc_bit
sc_bv
sc_fixed
sc_ufixed

(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.)

For sc_fixed and sc_ufixed, the following rounding and overflow modes are supported :

SC_RND
SC_RD_ZERO
SC_RND_MIN_INF
SC_RND_INF
SC_RND_CONV
SC_TRN
SC_TRN_ZERO

SC_SAT
SC_SAT_ZERO
SC_SAT_SYM
SC_WRAP
SC_WRAP_SM

5.1.2 Enumeration declarations

Supported.

5.1.3 The `asm` declaration

Not supported.

5.1.4 Linkage specifications

External linkage is not supported.

5.2 Declarators

Restricted support.
Declarators are supported as defined in ISOC++ with the restriction that exception handling is not supported.

5.2.1 Type names

Restricted support.
Type names are supported as defined in ISOC++ with the restriction that exception handling is not supported.

5.2.2 Ambiguity resolution

Supported as defined in ISOC++.

5.2.3 Function parameters

Function parameters are supported as defined in ISOC++ with the restriction that ellipsis are not supported.

5.2.4 Default arguments

Supported as defined in ISOC++.

5.2.5 Initializers

Restricted support.

A non-const variable declaration located in the body of a module must not have an initializer and must not be initialized 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. For detail information and examples please refer to Section 9.

5.2.5.1 Aggregates

Initialization of aggregates is supported as defined in ISOC++.

5.2.5.2 Character arrays

Initialization of character arrays is supported as defined in ISOC++. Also see Section **Error! Reference source not found.** that covers support on type char. For synthesis, if the numerical value of the char has an effect on functionality (the exception being comparing chars for equality) characters are assumed to be encoded in the ASCII character set. This is a refinement over ISOC++ which allows alternative *execution character sets*.

5.2.5.3 References

Initialization of references is supported as defined in ISOC++.

6 Expressions

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

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

has a behavior that is not specified in ISOC++. Expressions that are legal in ISOC++ but whose behavior is unspecified due to order of evaluation or order of side effects are supported for synthesis. Synthesis tools are permitted to interpret such expressions in any way that is compliant with the ISOC++ standard. It is advised that user code avoid such expressions to avoid differences in results between simulation using any particular compiler and synthesis using any particular synthesis tool.

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 definitions are supported as defined in ISOC++ with the restriction that specification of an exception-handling **try** block is not supported.

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.

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 section describes the different forms of sequential statements to be used in the synthesizable 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. Statements as defined in ISOC++ are supported with the restriction that specification of an exception handling **try** block is not supported. In addition, the SystemC **wait** statement and signal assignment statements are supported.

8.1 Labeled statement

Labels are supported as defined in ISOC++.

8.2 Compound statement

The compound statement (also, and equivalently, called "block") is supported as defined in ISOC++ to group sets of statements together.

8.3 Wait statement

Only simple **wait** statements with integer arguments must be used (default integer argument is 1). The **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.

8.4 Signal assignment statement

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

Signal or port identifier must denote a variable of type **sc_signal** or of a port type that 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**) . Resolved types are supported with restrictions as described in 4.4.4.2
- Signals can only be assigned one value in between two events

8.5 Selection statements

Selection statements include if, if-else and switch statements.
Selection statements are supported as defined in ISOC++.

8.5.1 The if statement

Supported as defined in ISOC++.

8.5.2 The switch statement

Supported as defined in ISOC++;

8.6 Iteration statements

The iteration statements **while**, **do** and **for** are supported as defined in ISOC++.

8.7 Jump statements

The jump statements **break**, **continue**, **return**, and **goto** are supported as defined in ISOC++

8.7.1 The break statement

The break statement is supported as defined in ISOC++.

8.7.2 The continue statement

The continue statement is supported as defined in ISOC++.

8.7.3 The return statement

Restricted support.
The **return** statement is supported as defined in ISOC++ with the restriction that a **return** statement may not occur within the function defining an **SC_CTHREAD** or **SC_METHOD** process.

8.8 Declaration statement

Declaration statements are supported as defined in ISOC++.

9 Processes

9.1 Clock

9.1.1 SC_CTHREAD

SC_CTHREAD processes use the second parameter to address an sc_in< bool > type port declared in the sc_module as the clock port. And notice that one must specify this module to be positive or negative clock triggered by using the pos() or neg() specifier appended to the SC_CTHREAD second parameter. Notice that bool is a 2-value data type therefore only '0' and '1' values are allowed and no 'Z' and 'X' can be assigned to a clock port.

9.1.2 SC_METHOD

The second way to specify a clock is to declare sc_in_clk ports in a module. This usage is specifically for SC_METHOD type processes. For each sc_in_clk port it must be accompanied by a sensitive clause of exactly one process, where pos() or neg() specifiers must be used to indicate the clock edge triggering property.

9.1.3 SC_THREAD

SC_THREAD is not synthesizable, please read Section 9.5 for detail descriptions. And since SC_THREAD is not synthesizable so there is no need to discuss its clock property.

9.1.4 Multiple Clocks

It is possible to have multiple clock ports in one module. However a process can only be triggered by exactly one clock. A clock can trigger more than one process.

. *Example*:

```
SC_MODULE( MC ) {
      sc_in< sc_uint<16> > x, y;
      sc_out< sc_uint<16> > sum, diff;
      sc_out< sc_uint<32> > mul;
      sc_in< bool > clk_ct; //  The SC_CTHREAD clock
      sc_in_clk clk_m; //  Must be specified in a sensitive clause

      void add() {            // This process is clocked by clk_m
            sum = x + y;
      }
      void sub() {            // This process is clocked by clk_t
            diff = x - y;
      }
      void multiply() {       // This process is clocked by clk_ct
            mul = x * y;
      }

      SC_CTOR( MC ) {
            SC_METHOD( add );
            sensitive << clk_m.pos();   // Triggering edge must be specified

            SC_METHOD( sub );
            sensitive << clk_m.neg();   // Triggering edge must be specified
```

```
            SC_CTHREAD( multiply, clk_ct.neg() );
                                        // Triggering edge must be specified
        }
    };
```

9.2 SC_METHOD

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.

9.2.1 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.

            output.write( in_a.read() & in_b.read() );
      }

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

Example:

```
SC_MODULE( Counter ) {
      sc_in_clk 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 << rst_n.neg() << clk.pos();
    }
};
```

9.3 SC_CTHREAD

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

Reset port specification is only supported in SC_CTHREAD type processes using a reset_signal_is(sc_in<bool>&, bool) clause. The first parameter addresses the reset port and the second parameter specifies this reset to be active high (true) or active low (false). Notice that one can only specify synchronous resets in IEEE 1666.

Example:

```
SC_MODULE( IIR ) {
    sc_in< sc_uint<32> > x;
    sc_out< sc_uint<32> > y;
    sc_in< bool > clk;
    sc_in< bool > reset;

    void iir() {        // This process describes purely combinational logic.
        …
    }

    SC_CTOR( IIR ) {
        SC_CTHREAD( iir, clk.pos() );    // Pos. trig. clock
        reset_signal_is( reset, false); // active low reset
    }
};
```

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. Any behavior after the first wait is considered operational behavior. Since the behavior inside the unbounded loop is consider operational behavior, the first wait that separates reset behavior from operational behavior must be located before the unbounded loop or is the first statement of the unbounded loop. Also notice that there may exist multiple waits before the unbounded loop. The following structures are suggested for SC_CTHREAD's. A synthesis tool must at least support these structures. In addition, it may support any other structure which satisfies the above requirements.

Examples:

```
// 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
        }
    }
```

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

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

and

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

9.4 SC_THREAD

SC_THREAD is non-synthesizable. SC_THREAD is blocking and can contain wait() statements that makes the process to wait on any signals in the sensitivity list. This also means an SC_THREAD process can be broken into stages by wait() statements. The non-blocking property of SC_METHOD makes it possible for synthesizers to synchronize input, computation and output. However in SC_THREAD this synchronization relationship is broken because it is hard for the synthesizer to synchronize inputs specified in wait() statements at different stages. Such input signal could be a clock or a reset. Without the support of reset_signal_is() in SC_THREAD it is hard for a synthesizer to synthesize the behaviour properly.

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.

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.

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.

If a sub-module is declared using 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

Namespace definition is supported as defined in ISOC++.

11.1.2 Unnamed namespaces

Supported as defined in ISOC++.

11.1.3 Namespace member definitions

Supported as defined in ISOC++.

11.1.4 Namespace alias

Supported as defined in ISOC++.

11.1.5 The `using` declaration

Supported as defined in ISOC++.

11.1.6 Using directive

Supported as defined in ISOC++.

12 Classes

Restricted support.

Classes and structs have restricted support. Unions are not supported.

12.1.1 Class names

Supported as defined in ISOC++.

12.1.2 Class members
Supported as defined in ISOC++.

12.1.3 Member functions
Supported as defined in ISOC++.

Note: the term member function only refers 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 as defined in ISOC++.

12.1.3.2 The `this` pointer
Supported as defined in ISOC++.

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 as defined in ISOC++.

12.1.7 Nested class declarations
Supported as defined in ISOC++.

12.1.8 Local class declarations
Supported as defined in ISOC++.

12.1.9 Nested type names
Supported as defined in ISOC++.

12.2 Derived classes

Supported as defined in ISOC++.

12.2.1 Multiple base classes

Supported as defined in ISOC++.

12.2.2 Member name lookup

Supported as defined in ISOC++.

12.2.3 Virtual functions

Virtual functions are supported provided that when such functions are called the type of the "this" object can be statically determined.

12.2.4 Abstract classes

Supported as defined in ISOC++.

12.3 Member access control

Supported as defined in ISOC++.

12.3.1 Access specifiers

Supported as defined in ISOC++.

12.3.2 Accessibility of base classes and base class members

Supported as defined in ISOC++.

12.3.3 Access declarations

Supported as defined in ISOC++.

12.3.4 Friends

Supported as defined in ISOC++.

12.3.5 Protected member access

Supported as defined in ISOC++.

12.3.6 Access to virtual functions

Supported as defined in ISOC++.

12.3.7 Multiple access

Supported as defined in ISOC++.

12.3.8 Nested classes

Supported as defined in ISOC++.

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

Supported as defined in ISOC++.

12.4.3 Conversions

Supported as defined in ISOC++.

12.4.3.1 Conversion by constructor
Supported as defined in ISOC++.

12.4.3.2 Conversion functions
Supported as defined in ISOC++.

12.4.4 Destructors
Supported as defined in ISOC++.

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 as defined in ISOC++.

12.4.6.2 Initializing bases and members
Supported as defined in ISOC++.

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
Supported as defined in ISOC++.

13 Overloading

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

13.1.1 Overloadable declarations

Supported as defined in ISOC++.

13.1.2 Declaration matching

Supported as defined in ISOC++.

13.1.3 Overload resolution

Supported as defined in ISOC++.

13.1.3.1 Candidate functions and argument lists

Supported as defined in ISOC++.

Function call syntax

Supported as defined in ISOC++.

Call to named function

Supported as defined in ISOC++.

Call to object of class type

Supported as defined in ISOC++.

Operators in expressions

Supported as defined in ISOC++.

Initialization by constructor

Supported as defined in ISOC++.

Copy-initialization of class by user-defined conversion

Supported as defined in ISOC++.

Initialization by conversion function

Supported as defined in ISOC++.

Initialization by conversion function for direct reference binding

Supported as defined in ISOC++.

13.1.3.2 Viable functions

Supported as defined in ISOC++.

13.1.3.3 Best Viable Function

Supported as defined in ISOC++.

Implicit conversion sequences

Supported as defined in ISOC++.

Standard conversion sequences
Supported as defined in ISOC++.

User-defined conversion sequences
Supported as defined in ISOC++.

Ellipsis conversion sequences
Not supported.

Reference Binding
Supported as defined in ISOC++.

Ranking implicit conversion sequences
Supported as defined in ISOC++.

13.1.4 Address of overloaded function
Address operations are not supported.

13.1.5 Overloaded operators
Supported as defined in ISOC++.

The **new**, **delete**, **new[]** and **delete[]** operators are not supported.

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

Supported as defined in ISOC++.

13.1.5.2 Binary operators

Supported as defined in ISOC++.

13.1.5.3 Assignment

Supported as defined in ISOC++.

13.1.5.4 Function call

Supported as defined in ISOC++.

13.1.5.5 Subscripting

Supported as defined in ISOC++.

13.1.5.6 Class member access

Supported as defined in ISOC++.

13.1.5.7 Increment and decrement

Supported as defined in ISOC++.

13.1.6 Built-in operators

Supported as defined in ISOC++.

14 Templates

Supported as defined in ISOC++.

14.1.1 Template parameters

Supported with the restriction that template parameters must not include pointers to functions.

14.1.2 Names of template specializations

Supported as defined in ISOC++.

14.1.3 Template arguments

Supported as defined in ISOC++.

14.1.3.1 Template type arguments

Supported as defined in ISOC++.

14.1.3.2 Template non-type arguments

Supported as defined in ISOC++.

14.1.3.3 Template template arguments

Supported as defined in ISOC++.

14.1.4 Type equivalence

Supported as defined in ISOC++.

14.1.5 Template declarations

Supported as defined in ISOC++.

14.1.5.1 Class Templates

Supported as defined in ISOC++.

Member functions of class templates

Supported as defined in ISOC++.

Member classes of class templates

Supported as defined in ISOC++.

Static data members of class templates

Supported as defined in ISOC++.

14.1.5.2 Member templates

Supported as defined in ISOC++.

14.1.5.3 Friends

Supported as defined in ISOC++.

14.1.5.4 Class template partial specializations

Supported as defined in ISOC++.

Matching of class template partial specializations
Supported as defined in ISOC++.

Partial ordering of class template specializations
Supported as defined in ISOC++.

Members of class template specializations
Supported as defined in ISOC++.

14.1.5.5 Function templates
Supported as defined in ISOC++.

Function template overloading
Supported as defined in ISOC++.

Partial ordering of function templates
Supported as defined in ISOC++.

14.1.6 Name resolution
Supported as defined in ISOC++.

14.1.6.1 Locally declared names
Supported as defined in ISOC++.

14.1.6.2 Dependent names
Supported as defined in ISOC++.

Dependent types
Supported as defined in ISOC++.

Type-dependent expressions
Supported as defined in ISOC++.

Value-dependent expressions
Supported as defined in ISOC++.

Dependent template arguments
Supported as defined in ISOC++.

14.1.6.3 Non-dependent names
Supported as defined in ISOC++.

14.1.6.4 Dependent name resolution
Supported as defined in ISOC++.

Point of instantiation
Supported as defined in ISOC++.

Candidate functions

Supported as defined in ISOC++.

14.1.6.5 Friend names declared within a class template
Supported as defined in ISOC++.

14.1.7 Template instantiation and specialization
Supported as defined in ISOC++.

14.1.7.1 Implicit instantiation
Supported as defined in ISOC++.

14.1.7.2 Explicit instantiation

14.1.7.3 Explicit specialization
Supported as defined in ISOC++.

14.1.8 Function template specializations
Supported as defined in ISOC++.

14.1.8.1 Explicit template argument specification
Supported as defined in ISOC++.

14.1.8.2 Template argument deduction
Supported as defined in ISOC++.

Deducing template arguments from a function call
Supported as defined in ISOC++.

Deducing template arguments taking the address of a function template
Supported as defined in ISOC++.

Deducing conversion function template arguments
Supported as defined in ISOC++.

Deducing template arguments from a type
Supported as defined in ISOC++.

14.1.8.3 Overloaded resolution
Supported as defined in ISOC++.

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).

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**.

When those constructs contain side effects (e.g., changing values of variables), a synthesis tool may warn and ignore them.

```
printf("x = %d", x);  // Ignored
printf("y = %d", ++y);      // Ignored with warning
cout << "z = " << z << endl;      // Ignored
```

18.3 Exception handling

Not supported.

Annex A Syntax summary (informative)

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 ::=
a*ny 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
| 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 **;**

selection_statement ::=
if (condition **)** statement
| **if** (condition **)** statement **else** statement
| **switch** (condition **)** statement

condition ::=
expression
| type_specifier_seq declarator = assignment_expression

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++.

Reference

[1] OSCI Synthesis WG, Basic Requirements of SystemC Subset for Synthesis
[2] OSCI Synthesis WG, Advanced Requirements of SystemC Subset for Synthesis
[3] G. Kahn, The semantics of a simple language for parallel programming, *Information Prasing*, J.L. Rosenfeld, Ed. North-Holland Publishing Co., 1974.
[4] Erwin de Kock et. al, YAPI: Application Modeling for Silgnal Processing Systems, *Proc. of DAC'00*, 2000, pp. 402-405.
[5] Erwin de Kock et. al., Proposal for Modeling Khan Process Networks and Synchronous Dataflow in SystemC, NXP Lab white paper
[6] B. Bhattacharya, J. Rose and S. Swan, Language Extensions to SystemC: Process Control Constructs, *Proc. of DAC*'07, 2007, pp. 35-38.
[7] ISO/IEC 14882: Programming languages - C++, 1998
[8] ISO/IEC 9899: Programming languages – C, 1999
[9] IEEE 1666 SystemC standard Language Reference Manual, IEEE, 2006
[10] Numerical Computation Guide. Chapter D: What Every Computer Scientist Should Know About Floating-Point Arithmetic. Sun Microsystems Inc. 2004. ISBN:0595286534.