SystemC Synthesizable Subset Version 1.4.7

March 2016

Notices

Suggestions for improvements to the SystemC Synthesizable Subset are welcome. They should be sent to the Working Group’s email reflector:

swg@lists.accellera.org

The current Working Group web page is:

www.accellera.org/activities/committees/systemc-synthesis

Introduction

The growing popularity of SystemC for system and hardware design has spurred significant growth in the high-level synthesis (HLS) industry in the recent past. While there are multiple commercially available HLS tools that accept SystemC as an input language, the inherent difference between the description of a simulation model and that of a synthesis model presents the question of which constructs and semantics of SystemC should be supported by these tools and how.

This standard addresses this issue by defining a subset of SystemC that will be suitable for input to HLS tools. It is intended for use by hardware designers and HLS tool developers in a manner that allows hardware designers to create HLS models in SystemC that will be portable among all conforming HLS tools.

It should be noted that the intent of this version of the standard is to define a minimum subset of SystemC for synthesis, it is not meant to restrict HLS tools' support for syntax beyond this subset, including additional vendor-specific design/library components, e.g., point-to-point handshaking channels. This standard also does not define any set of tool-directives that may be required to instruct an HLS tool on how to perform synthesis on certain constructs.

This standard is defined within the existing ISO C++ and IEEE 1666 SystemC specifications. Hence, familiarity with these standards is presumed. Familiarity with various levels of abstraction and their relationship with ESL (Electronic System Level) synthesis may also be helpful in understanding this standard. A separate discussion on this topic is added as an Annex of this standard.

This standard was defined by the Synthesis Working Group of the Accellera Systems Initiative with participation from various HLS tool vendors and users. Below is the list of individuals who contributed to this standard.

Participants
The following team members drove the Draft 1.4 effort:
Ashfaq Khan
Jos Verhaegh
Tony Kirke
Mike Meredith
Benjamin Carrion Schafer
Bob Condon
Alan P. Su
Andres Takach, Chair
Stuart Swan
Lucien Murray-Pitts
Yusuke Iguchi
Masachika Hamabe
Mark Johnstone
Frederic Doucet

The following team members drove the Draft 1.1 ~ 1.3 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

Contents

1 Overview

1.1 Purpose

In this standard document, the Synthesis Working Group (SWG) of the Accellera Systems Initiative has defined a subset of SystemC that is appropriate for synthesis. This is intended to be useful for hardware designers to accelerate the modeling process with SystemC and for EDA tool developers to develop SystemC compliant high-level synthesis (HLS) tools. The synthesizable subset of SystemC will be defined within the existing ISO C++ and IEEE Std 1666 SystemC specifications. Hence, familiarity with these standards is presumed.

There are a wide variety of resources available to assist users in modeling with SystemC, including the IEEE Std 1666 Language Reference Manual (LRM), user guides of the HLS tools, and a number of books on SystemC modeling. This document is intended to fill a gap by defining a standard for creating synthesizable hardware description in SystemC, allowing a smooth transition between abstract modeling in SystemC and synthesizable description.

1.2 Scope

The synthesizable subset defines the syntactic elements in ISO C++, as described in ISO/IEC 14882:2003 [1] and SystemC, as described in IEEE Std 1666-2011 [2], that are appropriate for use in SystemC code intended for input to HLS tools. In some cases, there are references to ISO/IEC 14882:2011 [3] for very specific items that were widely available before they were standardized in 2011. For example, the type *long long* and the behavior that division truncates towards zero (`0`). The intent of this version of the standard 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.

The synthesizable subset of SystemC currently covers the register transfer level (RTL) and the behavioral level. More abstraction levels are also discussed in this document to provide context.

1.3 Terminology

1.3.1 Base Standards

The following standards contain provisions which, through reference in this text, are included in this standard. At the time of publication, the editions indicated were valid.

- ISO/IEC 14882:2003, Programming languages – C++, hereinafter called *ISOC++*.
- ISO/IEC 14882:2011, Programming languages – C++, hereinafter called *ISOC++11*.

- IEEE Std1666-2011, SystemC, hereinafter called the *SystemC LRM*.

1.3.2 Word usage

The word *shall* indicates mandatory requirements strictly to be followed to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*; *shall not* equals *is not permitted to*).

The word *should* indicates a certain course of action is preferred, but is not a mandatory requirement; or (in the negative form, *should not*) that a certain course of action is permitted, but such usage is discouraged (*should* equals *is recommended that*).

The word *may* indicates a course of action permissible within the limits of the standard (*may* equals *is permitted*).

The word *application* is a C++ program written by an end user that uses the SystemC, TLM-1, and TLM-2.0 class libraries, i.e., it uses classes, functions, or macros defined in the SystemC LRM.

The word *implementation* means any specific implementation of the full SystemC, TLM-1, and TLM-2.0 class libraries, as defined in the SystemC LRM, of which only the public interface need be exposed to the application.

The word *design* is used to mean a SystemC design written by an end user that describes hardware in conformance with this standard.

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) if it accepts that construct and produces a corresponding synthesis result.

The word *synthesis tool* is used to mean a high-level synthesis tool that *accepts* and *interprets* a *design* in conformance with this standard.

The term *call* is taken to mean call directly or indirectly. Call indirectly means call an intermediate function that in turn calls the function in question, where the chain of function calls *may* be extended indefinitely. Similarly, *called from* means called from directly or indirectly.

The term *class* is used to cover the C++ keywords *class* or *struct*.

Except where explicitly qualified, the term *derived from* (or *inherited from*) is taken to mean derived directly or indirectly from a *class*. Derived indirectly from means derived from one or more intermediate base *classes*.

A *synthesis refinement* in this document imposes a restriction or alteration upon some other standard (e.g., the SystemC LRM or the ISOC++ standard) in order to subset the otherwise supported methods to describe behavior to a set that can be implemented in hardware.

The word *deprecated* is used to describe a feature that is superseded by a better, safer, easier to use alternative. Use of the *deprecated* feature is strongly discouraged and future standards *may* make such use illegal.

1.3.3 Construct Categories

The constructs in this standard *shall* be categorized as:

Supported: A *synthesis tool* *shall* interpret the construct.

Ignored: A *synthesis tool* *shall* accept the construct, but *may* choose not to interpret the construct. The mechanism, if any, by which a *synthesis tool* notifies (warns) the user of such constructs is not defined in this standard.

Not Supported: A *synthesis tool* *may* choose not to accept the construct. The behavior of the *synthesis tool* upon encountering such a construct is not defined in this standard. For example, a *synthesis tool* *may* choose to fail upon encountering such a construct; alternatively, it *may* choose to accept/interpret such a construct. It *should* be noted that even if a *synthesis tool* accepts/interprets some of the constructs that are ***Not Supported***, a *design* that uses such constructs runs the risk of losing portability. However, it is also possible that, in a future revision of this standard, some of the constructs that are ***Not Supported*** now will be ***Supported***.

Supported with Restrictions: A *synthesis tool* *shall* interpret the construct with certain restrictions. This means, instances of the construct which are within the restrictions, as set forth by this standard, are ***Supported***; while instances that violate these restrictions are either ***Ignored*** or ***Not Supported***. Unless explicitly categorized as ***Ignored***, the violating instances are ***Not Supported***. A construct that is ***Supported with Restrictions*** is also said to have ***Restricted Support***.

1.4 Conventions

This document uses the following conventions:

a) The body of the text of this standard uses *italics* to denote SystemC or C++ reserved words (e.g., *sensitive* and *SC_MODULE*).
b) The body of the text of this standard also uses *italics underlined* to highlight definitions (e.g., *application* or *design*) or to visually reinforce key terms (e.g., *shall*, *should not*, and *synthesis refinement*).
c) The body of the text of this standard uses ***bold italics*** to visually reinforce construct categories (e.g., ***Supported*** and ***Ignored***).
d) The text of the SystemC examples and code fragments is represented in a `fixed-width font`.
e) An outlining box with the title “NOTE” provides an informative expansion of certain key concepts. They are intended to assist in understanding of a construct, but are not intended as a restriction or enhancement of a *synthesis tool*
f) 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 ISOC++ Implementation Compliance (ISOC++ 1.4)

The ISOC++ Section 1.4 Implementation Compliance applies to the synthesis subset.

1.5.1 Implementation-defined behavior (ISOC++ 1.3.5)

Simulation of the *design* is based on C++ compilers on specific computer platforms. In most cases, such platforms have converged on certain default implementation-defined behaviors

and *synthesis tools* are required to either adhere to those behaviors or provide warnings to the effect that a different implementation-defined behavior is being followed.
An example of an implementation-defined behavior is the bit-width of fundamental integer types.

1.5.2 Undefined behavior (ISOC++ 1.3.12)

The presence of undefined behavior *may* lead to differences between simulation and synthesis and differences in results from different *synthesis tools*. Unless this standard provides a *synthesis refinement* that provides a definition, such undefined behavior is ***Not Supported***
In some cases, undefined behavior is assumed to be not present in the *design*. One specific example is division by zero (`0`). *Synthesis tools* *may* assume that such a condition will not be present during simulation of the *design* and treat such a condition as a don't care.

1.5.3 Unspecified behavior (ISOC++ 1.3.13)

The presence of unspecified behavior *may* lead to differences between simulation and synthesis results and differences in results from different *synthesis tools*. An example is the order of evaluation of arguments to a function is not specified. In ISOC++, the implementation is not required to document which behavior occurs.

1.6 SystemC LRM Compliance

This section describes some general terms that are used in the SystemC LRM that have some similarities with terms used in ISOC++ as described in Section 1.5.

1.6.1 Implementation-defined (SystemC LRM 3.2.1)

The description in Section 1.5.1 applies. In general, most of the implementation-defined items in the SystemC LRM do not affect synthesis. One item that does affect synthesis is the type for limited precision (SystemC LRM 7.2.2). In that case, this standard only supports one implementation as specified in Section 6.3.

1.6.2 Undefined

The description in Section 1.5.2 applies when the word *undefined* is used in the context of the behavior not being defined.

1.6.3 Error

The word error is sometimes used interchangeably with the word undefined in the SystemC LRM. In many cases, an *error* refers to a compilation or simulation error. *Synthesis tools* assume that such conditions will not be present and treat such condition as a don't care, unless the *synthesis tools* are able to prove that the error is present, in which case they *ma*y issue an error appropriately.

2 Translation units

2.1 Translation units and their analysis

The text of a design is, as described by ISOC++ Section 2, kept in units called source files. A source file together with all included sources, less any lines skipped through preprocessor macros, is known as a translation unit.

A translation unit shall be specified using the basic source character set, as described by ISOC++ Section 2. However, the trigraph sequences, as described by ISOC++ section 2.3, and alternative tokens are ***Not Supported***. Furthermore, instance names using Universal-Character-Name as described in ISOC++ Section 2 and ISOC++ Annex E are synthesis tool-dependent and, therefore, ***Not Supported***.

Using multiple translation units to describe a design is ***Supported***. This standard, however, does not specify how a user will provide the translation-unit information to the synthesis tool.

The use of the keyword *extern* to refer to a declaration in a different scope within the same translation unit or in a different translation unit is ***Supported***. However, the use of libraries (pre-compiled binary files) is ***Not Supported***, and a design shall contain all the source files required for synthesis (with the exception of SystemC implementation and C/C++ native library files).

The use of *extern* string-literals for linkage specification (e.g., `extern "C"`, `extern "C++"`) is ***Not Supported*** (ISOC++ 7.5).

The functions *main* and *sc_main* are ***Ignored***.

2.2 Pre-processing 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.

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 version of the synthesis subset, in a year and month format such as 201601L.
4. `__SYNTHESIS__`: The value is implementation-dependent.

NOTE
The expected use of the `__SYNTHESIS__` and `SC_SYNTHESIS` macros is to guard non-synthesizable code. The version contained in `SC_SYNTHESIS` is meant to select different versions of the same behavior that depend on updates to the synthesis subset.

3 Modules

This section specifies the core SystemC language subset used in a _design_ for modeling a hardware element. In this document *sc_module* and module are used interchangeably. A single hardware element is represented by a module and a system is described by creating a hierarchy of modules stemming from a parent module.

This section does not limit the optimizations that can be performed on the model, the scope of which depends on the _synthesis tool_ itself. Irrespective of the level of optimizations performed, the result of the synthesis _should_ have the same functionality as the input model, but _may_ have different sequential timed behavior.

3.1 Module definitions

A SystemC module, as defined in the SystemC LRM definition, _may_ contain the port-level interfaces, any required internal storage elements, and any required behavior for that module.

The ***Supported*** possibilities for module definition are as follows.

- Use of the *SC_MODULE* macro;
- Derivation of a _class_ from *sc_module*.

In addition to the above, templated module definition has ***Restricted Support***. The restriction is that, for the top-level module, only modules that have been instantiated (as defined in ISOC++ 14.7.1 and 14.7.2) or have been explicitly specialized (as defined in ISOC++ 14.7.3) are ***Supported***. For non-top-level modules, templated module definition is ***Supported***.

Template specialization and partial-specialization of modules is also ***Supported***.

NOTE
In C++, template classes and functions are not instantiated until they are implicitly instantiated (as defined in ISOC++14.7.1) or are explicitly instantiated (as defined in ISOC++ 14.7.2). In C++, template classes and functions can be explicitly specialized (as defined in ISOC++ 14.7.3) using the "`template <>`" declaration (i.e., the class is fully specialized). Examples of implicit and explicit instantiations and explicit specialization are shown below. The most common form for synthesis is the use of implicit instantiation for submodules and implicit instantiation of the template top-level module in the testbench (from `main` or `sc_main`). If the testbench is not present or is excluded from analysis, the use of an explicit instantiation would be indicated. Note that explicit specializations are less common and only indicated when there is an advantage in providing a specialized version of the module (the example below is not an illustration of such a case).

```
template <int N>
SC_MODULE (design) {
  sc_in<int> a;
  sc_out<int> c;
```

```
    void add() { c = a + N;}

    SC_CTOR(design) : a("a"), c("c") {
      SC_METHOD(add);
      sensitive << a;
    }
  };

  template class design<5>;     // EXPLICIT INSTANTIATION

  int main() {
    design<7> x;        // IMPLICIT INSTANTIATION

  }

  template<> SC_MODULE(design<3>) {   // EXPLICIT SPECIALIZATION
    sc_in<int> a;
    sc_out<int> c;

    void add() { c = a + 3; }

    SC_CTOR(design) : a("a"), c("c") {
      SC_METHOD(add);
      sensitive << a;
    }
  };
```

3.1.1 Selecting the top of a design hierarchy

This standard does not define any method for specifying the top-level module(s) of a design. The mechanisms for specifying the top-level module(s) of a design are synthesis tool dependent.

3.1.2 Module member specification

The module member specification contains a set of member declarations and definitions. Any valid and legal SystemC macros are ***Supported***.

3.1.3 Module declarative items

3.1.3.1 Module special functions

Module constructors are ***Supported***. Module destructors are ***Ignored***. The other two special member functions, namely the *Copy Constructor* and the *Assignment Operator*, as defined in ISO C++ Section 12, are disabled for *sc_module* by the SystemC LRM 5.2.2, hence, they are disabled for *sc_module*.

3.1.3.2 Communication between processes through module member variables

Within a module, processes *shall* only communicate with each other through member variables which are of type *sc_signal*. Non-const variables which are not of *sc_signal* type *shall* be read and written by only one process.

3.1.3.3 Communication between modules

Communication between modules <u>shall</u> only be through *sc_in* and *sc_out*, which are to be bound to *sc_signal*s.

Calling the binding function for a port that is a data member of a module from the constructor of the parent to the module is ***Supported***.

Other cases of access to module non-const data members from outside the module is ***Not Supported***.

3.1.3.4 sc_port, sc_export, sc_signal, and other channels

- Instantiation of a *sc_port* or directly deriving from *sc_port* in a <u>design</u> is ***Not Supported***.
- Instantiation of a *sc_export* or directly deriving from *sc_export* in a <u>design</u> is ***Not Supported***.

The specialized ports and channels are described in Section 5 of this standard.

NOTE
Ports represent the externally visible interface to a module and are used to transfer data into and out of the module. Specialized ports using the *sc_in* and *sc_out* construcst are used to describe this pin level description at the module boundary. Signals can be used to interface between processes and modules. Signals are declared using the *sc_signal* construct.

3.1.3.5 Module constructor

Module constructor declaration through the use of the *SC_CTOR* macro or explicit declaration of a constructor special function either with no argument or a single argument for the module name are ***Supported***. Explicit declaration of a constructor special function with more than one argument or an argument that is not a module name is ***Not Supported***.

Every module declaration <u>shall</u> contain one declaration or definition of a constructor member function. Multiple constructor declarations or definitions are ***Not Supported***.

Within a constructor of an *SC_MODULE* and functions called from the constructor, the following operations are ***Supported*** to construct module hierarchy:

- Constructor calls of *sc_in*/*sc_out*/*sc_signal*/*SC_MODULE*.
- Initialization of pointers to *SC_MODULE*s with *SC_MODULE* objects allocated using the *new* operator.
- Port bindings between *sc_signal*/*sc_in*/*sc_out* using *bind()* and *operator()*.
- Creation of processes using *SC_THREAD*/*SC_CTHREAD*/*SC_METHOD* and sensitivity specification as described in Section 4.

In addition, initialization of references and constant data members in the constructor initializer list is ***Supported***.

In contrast to normal C++ practice, writing to data members other than initialization of *SC_MODULE* pointers, signals, or ports of a module from inside the *SC_MODULE* constructor are ***Not Supported***. Overwriting a variable of type pointer to *sc_module* after initialization is ***Not Supported***.

NOTE
Initialization of an *SC_MODULE* data member should be performed within the reset portion of an *SC_THREAD*, *SC_CTHREAD*, or sequential *SC_METHOD* in order to have the initialization be performed at reset time

3.2 Deriving modules

Deriving modules, as defined in SystemC LRM 5.2.3, is ***Supported***.

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() );
              reset_signal_is( reset, true ) ;
            }
          };
```

3.3 Module hierarchy

Module hierarchy, as defined in SystemC LRM 5.3.4, is ***Supported***.

Port binding, as described in SystemC LRM 4.1.3 is ***Supported***. Note that positional binding (as described in SystemC LRM Annex C as a deprecated feature) is ***Not Supported***.

4 Processes

The use of the *SC_THREAD*, *SC_CTHREAD*, and *SC_METHOD* constructs to create processes is ***Supported with Restrictions***. The use of *sc_spawn()* to create processes is ***Not Supported***.

None of the uses of *sc_process_handle* are synthesizable, hence such usage is ***Not Supported***. Consequently, the use of *suspend(), resume(), enable(), disable(), kill(), reset(),* and *throw_it()* is also ***Not Supported***.

4.1 SC_METHOD

In a SystemC *design*, the body of an *SC_METHOD* is executed whenever its sensitivity condition is met. The logic inferred by a *synthesis tool* for an *SC_METHOD* depends on both the sensitivity condition and the form of the *SC_METHOD* body.

As a *synthesis refinement*, all paths through an *SC_METHOD* *shall* either be combinational or sequential. An *SC_METHOD* with a mix of combinational and sequential paths is ***Not Supported***.

For *SC_METHOD*, *reset_signal_is* and *async_reset_signal_is* are ***Not Supported***. Modeling of reset behavior is done using the coding styles described below.

4.1.1 Combinational SC_METHOD

An *SC_METHOD* can used to describe combinational logic, with the *SC_METHOD* sensitive to any change in the signals in its sensitivity list.

As a *synthesis refinement*, the sensitivity list here *shall* be static and include all signals that are read in the body of the method. This is done to avoid accidental latching, which would lead to synthesis and simulation mismatches.

```
// Example of combinational SC_METHOD

SC_METHOD(comb); sensitive << a << b << c;

void comb() { f.write(a.read()+b.read()+c.read()); }
```

4.1.2 Sequential SC_METHOD

The body of a sequential *SC_METHOD* *shall* consists of two mutually exclusive branches based on the clock and reset conditions.

4.1.2.1 Sequential SC_METHOD with synchronous reset

A sequential *SC_METHOD* with synchronous reset *shall* be sensitive to only a positive or negative edge of a clock signal.

```
// Example of sequential SC_METHOD with synchronous reset:

SC_METHOD(dff); sensitive << clk.pos();
```

```
void dff() {
   if (rst == 0) {
      // reset signals
   } else {
      // assign signals
   }
}
```

4.1.2.2 Sequential SC_METHOD with asynchronous reset

A sequential *SC_METHOD* with asynchronous reset *shall* be sensitive to only a positive or negative edge of a clock signal and a positive or negative edge of a reset signal.

```
// Example of sequential SC_METHOD with asynchronous reset

SC_METHOD(dff); sensitive << clk.pos() << rst.neg();

void dff() {
   if (rst == 0) {
      // reset signals
   } else {
      // assign signals
   }
}
```

4.2 SC_THREAD and SC_CTHREAD

SystemC allows the usage of the *SC_THREAD* and *SC_CTHREAD* macros to create unspawned processes which run from the start of simulation until the end of simulation.

SC_CTHREAD and *SC_THREAD* are ***Supported with Restrictions***. They are ***Supported***, as specified in the SystemC LRM, when the following restrictions regarding the process sensitivity and process body are met (in addition to the other restrictions mentioned throughout this standard, e.g., regarding break/continue/goto statements).

NOTE
In earlier versions of SystemC, *SC_THREAD* and *SC_CTHREAD* differed in their reset behavior. Currently, there are minor syntax differences between the two, but for synthesis purposes, they have the same expressiveness.

4.2.1 Clock and Reset

The sensitivity of the process *shall* be specified in the constructor of the *SC_MODULE* enclosing the process. A process *shall* be statically sensitive to exactly one clock edge and *shall* have at least one reset specification. A process *may* have at most one synchronous reset specification and at most one asynchronous reset specification. Different *SC_(C)THREAD*s *may* be sensitive to different clocks and resets, i.e., an *SC_MODULE* *may* contain multiple clocks.

The use of *reset_signal_is* and *async_reset_signal_is* to specify reset sensitivity is **Supported**.

SC_THREAD example:

```
SC_THREAD(thread_process);
sensitive << clk.pos();
async_reset_signal_is( rst, false);// active low asynchronous reset
```

SC_CTHREAD example:

```
SC_CTHREAD(thread_process, clk.pos());
async_reset_signal_is( rst, false);// active low asynchronous reset
```

4.2.2 Thread process body

In a SystemC *application*, it is a common coding idiom to include an infinite loop containing a call to the *wait()* function within a thread process in order to prevent the process from terminating prematurely, at the same time allowing co-operative pre-emption during simulation by suspending the process. The thread process body of a *design* *shall* also follow this structure. Among the available constructs to suspend a process, a *design* *shall* only use a call to *wait()*, where the wait condition is the clock edge to which the thread process is sensitive, as specified in the module constructor. That is, the only ***Supported*** waits are the following.

- *wait()*
- *wait(int)*: where the integer argument is statically determinable.

Other forms of suspending a thread process are ***Not Supported***.

The behavior of a thread process consists of reset behavior, which executes on reset, and operational behavior, which executes after reset.

The body of the thread process in a *design* *shall* follow the form:

> [<optional reset behavior>]
> [<optional operational behavior>]
> <infinite loop> subject to the restrictions below

The reset and operational behavior *shall* be separated by a call to the *wait()* function. Any behavior encountered in the thread process prior to encountering the first call to the *wait()* function *shall* be considered by the *synthesis tool* as reset behavior. Any behavior encountered in the thread process after encountering the first call to the *wait()* function *shall* be considered by the *synthesis tool* as operational behavior.

The first call to the *wait()* function *may* occur before the infinite loop, or it *may* occur within the infinite loop. In the case where the first call to the *wait()* function occurs within the infinite loop, the behavior within the loop body prior to the first call to the *wait()* function *shall* be considered by the *synthesis tool* to be both reset behavior and operational behavior.

Multiple calls to the *wait()* function *may* occur within the body of the thread process, both before the infinite loop, and within the body of the infinite loop. The behavior encountered before the first call to the *wait()* function *shall* be the reset behavior; any behavior encountered after the first call to the *wait()* function *shall* be the operational behavior.

Within the body of the thread process, the first call to the *wait()* function shall be statically determinable. The first call to the *wait()* function may be within conditional constructs only when the condition is statically determinable (e.g., in the case where the condition is based on a template parameter). Subsequent calls to the *wait()* function (in the operational behavior) may occur within conditional constructs.

Reset values shall be statically determinable constants.

The following forms of infinite loop are ***Supported***, others are ***Not Supported***.

while(1) { }
while(true) { }
do { } while (1);
do { } while (true);
for (; ;) { }

NOTE
Example of SC_THREAD/SC_CTHREAD body:

```
void dut::thread_process() {
   // Reset behavior goes here
   wait(); // This wait separates reset behavior from
           // operational behavior.
   // You can  operational behavior here
   while(1) { // Repeated operational behavior.
              // The infinite loop prevents the operational behavior
              // from terminating.
      wait(); // This wait allows suspension of the process.
   }
}

void dut::thread_process() {
   // Reset behavior goes here
   // You cannot have operational behavior here, because there
   // is no wait() to separate them from reset behavior.
   while(1) {
      // The infinite loop prevents the process from terminating
      // Behavior here is both reset behavior and operational
      // behavior.
      wait(); // This wait separates reset behavior from
              // operational behavior and also allows suspension
              // of the process.
      //operational behavior
   }
}
```

NOTE
Reset behavior is behavior that is executed when reset is asserted. As Section 3.1.3.5 indicates, writing to data members, signals, or ports of a module from inside the *SC_MODULE* constructor is not reset behavior. Default constructors of data members of the *SC_MODULE* that initialize the data members (e.g., the default constructor for *sc_int* initializes the value to `0`), are not part of the reset behavior. If the behavior of the design depends on those non-reset initial values, then synthesis results might differ from 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! Synthesis will Not invoke default ctor for X.
     xChildSig("xChildSig") //Warning! Synthesis will Not invoke
                            //default constructor for XChild.
  {
     SC_CTHREAD(proc, clk.pos());
     reset_signal_is(rst, true);
  }

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

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

5 Predefined channels, interface proper and ports

5.1 Predefined Channels

5.1.1 sc_signal

sc_signal has ***Restricted Support***. Use of a *sc_signal* with the *WRITER_POLICY* defaulted or explicitly set to *SC_ONE_WRITER* is ***Supported***. Use of a *sc_signal* with the *SC_MANY_WRITERS* policy is ***Not Supported***.

Furthermore, only the following member functions are ***Supported***.

void write(const T&)
const T& read( )
sc_signal(), *sc_signal(string)*, and *sc_signal(signal)*
operator= (const T&)
operator= (const sc_signal<T,WRITER_POLICY>&)

Arrays of *sc_signal* are ***Supported***.

5.1.2 Resolved Channels

Resolved types are ***Not Supported***. This includes *sc_signal_resolved*, *sc_in_resolved*, *sc_inout_resolved*, *sc_out_resolved*, *sc_signal_rv*, *sc_in_rv*, *sc_inout_rv*, and *sc_out_rv*.

5.1.3 Other Channels

The following list of pre-defined SystemC channels are also ***Not Supported***:
sc_buffer, *sc_clock*, *sc_mutex*, *sc_semaphore*, *sc_fifo*, and *sc_event_queue*.

NOTE
While *sc_clock* channel is not supported, clock ports are supported as specified in Section 5.2.1.

5.2 Ports

The pre-defined specialized port classes are ***Supported*** so blocks in a SystemC hierarchy can communicate through convenient access to member functions of the pre-defined SystemC primitive channels.

5.2.1 sc_in, sc_out, and sc_inout

sc_in<T> and *sc_out<T>* for *T* being any synthesizable type are ***Supported***.
sc_in<bool>, and *sc_in_clk* are all ***Supported***.
sc_in<sc_dt::sc_logic> is ***Not Supported***.
*sc_inout<T> is **Not Supported***.

Furthermore, for *sc_in<T>*, only the following member functions are ***Supported***.

sc_in() and *sc_in(const char*)*
const T& read( ) const
operator const T& () const
void bind(const sc_signal_in_if<T>&)

void operator() (const sc_signal_in_if<T> &)
void bind(sc_port< sc_signal_in_if<T>, 1> &)
void operator() (sc_port< sc_signal_in_if<T>, 1> &)
void bind(sc_port< sc_signal_inout_if<T>, 1> &)
void operator() (sc_port< sc_signal_inout_if<T>, 1> &)

For *sc_in<bool>*, in addition to the above, the following further member functions are ***Supported***.

sc_event_finder& pos() const
sc_event_finder& neg() const

Furthermore, for *sc_out<T>*, only the following member functions are ***Supported***.

sc_out() and *sc_out(const char*)*
const T& read( ) const
operator const T& () const
void write(const T&)
operator= (const T&)
operator= (const sc_signal_in_if<T >&)
operator= (const sc_port< sc_signal_in_if<T >, 1> &)
operator= (const sc_port< sc_signal_inout_if<T >, 1> &)
operator= (const sc_out< T> &)
void bind(const sc_signal_inout_if<T>&)
void operator() (const sc_signal_inout_if<T> &)

For *sc_out<bool>*, in addition to the above, the following further member functions are ***Supported***.

sc_event_finder& pos() const
sc_event_finder& neg() const

Arrays of ports are ***Supported***.

Inheriting from the specialized port types is ***Not Supported***.

5.3 **sc_event**

sc_event in a <u>*design*</u> is ***Not Supported***.

6 Types

SystemC types are comprised of both the native C++ types and the additional SystemC types.

There are two kinds of native C++ types: fundamental types and compound types. Types describe objects, references, or functions.

NOTE
Alignment requirements mentioned in ISOC++ 3.9 are not relevant for synthesis. Synthesis may choose alternative data representations for internal objects (not part of the interface of the *design*) provided the I/O behavior of the *design* is unchanged. For example, the bit-width of an integer variable could be reduced based on the range of the variable or its representation could be changed from two's complement to sign-magnitude.

6.1 Fundamental Types

Fundamental types are comprised of integer types, floating-point types, and *void*.

6.1.1 Integer Types

The following integer types, as specified in ISOC++ and ISOC++11 3.9.1, are ***Supported***:

- *bool*
- *unsigned char, signed char, char*
- *unsigned short, signed short*
- *unsigned int, signed int*
- *unsigned long, signed long*
- *unsigned long long, signed long long* (ISOC++ 11)

The integer type *wchar_t* is ***Not Supported***.

NOTE
ISOC++ 3.9.1 specifies that it is implementation defined whether a *char* object can hold negative values; GNU G++ can support either mode through use of the "-*fsigned_char*/*unsigned_char*" switch. The *synthesis tool* vendor is likely to choose the mode that best meets the architecture for which their simulation model was built in order to achieve consistent results between simulation and synthesis models. ISOC++ 3.9.1 also specifies that plain *char, signed char,* and *unsigned char* are three distinct types. Even if a particular implementation allows a *char* to hold negative values, it is not the same type as a *signed char*.

6.1.1.1 Literals

Integer and Boolean literals, as specified in ISOC++ 2.13.1 and 2.13.5, are ***Supported***.

Character literals as described in ISOC++ 2.13.2 have ***Restricted Support***. The *L* prefix denoting wide character support is ***Not Supported***.

For synthesis, if the numerical value of the *char* literal has an effect on functionality (the exception being comparing *char*s for equality), characters *shall* be assumed to be encoded in the ASCII character set. This is a *synthesis refinement* over ISOC++ 2.13.2 and 2.13.4, which allows alternative execution character sets (ISOC++ Section 2.2, Paragraph 3).

6.1.1.2 **Representation and Bit Sizes**

Two's complement integer representations are ***Supported***. One's complement and sign magnitude integer representations are ***Not Supported***.

A *synthesis tool* *shall* have mechanisms to support the I/O observable behavior implied by bit-widths and representation for the computer platforms as indicated in the last column of the table below. It *shall* warn in case the choice of bit-width or the representation is not consistent with the definition for the computer platform.

Table 1: Bit Sizes for Integer Types

Integer Type	Relative Requirement	Current Compilers	
		Signed Representation	**Bit Width**
(un)signed char, char		two's complement	8
(un)signed short	bits(short) ≥ bits(char)	two's complement	16
(un)signed int	bits(int) ≥ bits(short)	two's complement	32
(un)signed long	bits(long) ≥ bits(int)	two's complement	32/64
(un)signed long long	bits(long long) ≥ bits(long)	two's complement	64

NOTE

1: The representation and the bit-width of an integer type determines its numerical range and its overflow behavior.

2: The ISOC++ and ISOC++11 standards set minimum requirements for the bit widths of integer types, but leave bit widths and the representation implementation-dependent.

3: ISOC++ 3.9.1 specifies unsigned integers *shall* obey the laws of arithmetic module 2^n, where *n* is the number of bits. Signed integers are of a pure binary numeration system and the representations allowed are two's complement, one's complement, and sign magnitude.

4: Table 1 provides an overview of the ISOC++ 3.9.1 requirements and bits sizes for integer types used on most compilers for popular computer platforms. It constrains the relative sizes of the different integer types, and also requires the signed and unsigned (and plain in the case of characters) versions of integer types to have the same storage. The last column in the table shows the bit widths for current platforms. As the table indicates, there is only a difference for the (un)signed long types in current platforms.

5: ISOC++11 provides typedefs for "exact-width" integer types. These are *intN_t* and *uintN_t*, where *N* can be `8`, `16`, `32`, or `64`. The typedefs are defined in the *std* namespace in the include <*cstdint*> header. For example, `int64_t` is defined as *long* on platforms where *long* is 64-bits wide and as *long long* on platforms where *long* is 32-bits wide.

6.1.2 Type Conversions

Type conversions are ***Supported*** as specified in the sections below.

NOTE
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*). The 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. An example of a usual arithmetic conversion is when an 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*.

6.1.2.1 **Integer Promotions**

Integer (Integral) Promotions (as defined in ISOC++ 4.5) are ***Supported***.

6.1.2.2 **Usual Arithmetic Conversions**

Usual Arithmetic Conversions (as defined in ISOC++ Section 5 and ISOC++11 4.5 corresponding to the addition of the *long long* types) are ***Supported***.

6.1.3 Operators

The following operators are ***Supported*** for the integer types:

1. Unary operators (`+`, `-`, `~`, and `!`).
2. Arithmetic binary operators (`+`, `-` , `*`, `/`, and `%`) and the corresponding assign operators (`+=`, `-=`, `*=`, `/=`, and `%=`). The result of division is truncated towards zero (`0`) as specified in ISOC++11 5.6.
3. Relational and Equality operators (`>`, `>=`, `<`, `<=`, `==`, and `!=`).
4. Bitwise binary operators (`&`, `|`, and `^`) and the corresponding assign operators (`&=`, `|=`, and `^=`).
5. The Conditional Operator (`?:`).

NOTE

The ~ operator on a *bool* argument x first promotes it to an *int* and then computes the one’s complement (value is -x-1) of the promoted value (ISOC++ 5.3.1).
 	
`bool x = true;`
`int y = ~x;   // y is -2`
`bool z = ~x;  // z is true`
 	
If the intended behavior is to get the logical complement, the logical negation operator *!* (ISOC++ 5.3.1) should be used.

The following operators have ***Restricted Support***.

1. Shift and shift assign operators (<<, >>, <<=, and >>=). Considering `E1` and `E2` as the two operands (as in "`E1` *shift_op* `E2`"), the support is as follows.
    a. ISOC++ 5.8 specifies that valid ranges for `E2` is `0` to *length*(*promoted_type*(`E1`))`-1` and that otherwise the behavior is undefined (Section 1.5.2). Shifts are ***Supported*** for valid ranges and ***Not Supported*** otherwise.
    b. For right shifts, if `E1` has a signed type, the sign bit shall be shifted in. This is a synthesis refinement on ISOC++ and ISOC++11, since these standards leave the behavior implementation-defined when `E1` is negative.
    c. For left shifts, the behavior specified in ISOC++ with the natural implications due to the addition of the *long long* types are presumed in this standard.
2. The prefix and postfix increment and decrement operators (`++x`, `--x`, `x++`, and `x--`) have ***Restricted Support***. The restriction is that the prefix and postfix increment operators on a *bool* operand are ***Not Supported*** as they are deprecated by ISOC++.

6.1.4 Floating Point Types

Floating literals (as specified in ISOC++ 2.13.3) are ***Supported*** for initializing synthesizable datatypes. If the floating literal is used to initialize an integer type (a floating-integral conversion is involved as specified in ISOC++ 4.9) and the truncated value is not representable in the integer type, then the behavior is undefined (see Section 1.5.2).

Otherwise, floating-point types (specified in ISOC++ 3.9.1 as *float*, *double*, and *long double*) are ***Not Supported***.

NOTE
One of the challenges for providing general synthesis support for floating-point datatypes is the fact that the bit accurate behavior of floating point arithmetic is dependent on implementations and compiler options used [4].

6.1.5 The void type

The *void* type (ISOC++ 3.9.1) is ***Supported***.

6.2 Compound Types

Compound types in C++, as described in ISOC++ 3.9.2, have ***Restricted Support*** determined by the support restrictions of the constituent types.

6.3 SystemC Datatypes

SystemC provides a number of datatypes that are useful for hardware design. These datatypes are implemented as C++ classes.

The following SystemC types have ***Restricted Supported*** as described in the sections below:

- Limited Precision Integer Types: *sc_int* and *sc_uint* (Section 6.3.1)
- Finite Precision Integer Types: *sc_bigint* and *sc_biguint* (Section 6.3.2)
- Finite Precision Fixed-point Types: *sc_fixed* and *sc_ufixed* (Section 6.3.3)
- Finite Word-Length Bit Vector Type: *sc_bv* (Section 6.3.4.1)
- Finite Word-Length Logic Vector Types (4-valued): *sc_lv* (Section 6.3.4.2)
- Single Bit Logic (4-valued): *sc_logic* (Section 6.3.4.2)

The explicit use of SystemC datatypes that are not in the list above is ***Not Supported***. The implicit use of other related types that arise as return types from operators on the types listed above has ***Restricted Support***. The restriction is that the bitwidth of a return type needs to be statically determinable.

NOTE
1: 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. 2: Underlying classes, such as *sc_signed* and *sc_unsigned*, can appear as the result of expression on supported types, but are not directly synthesizable. Their length/precision is dynamic. Some base and helper classes are specifically denoted in SystemC LRM 3.2.4 as classes that should not be used explicitly. The SystemC LRM annotates those classes with a superscript dagger (†).

6.3.1 Limited Precision Integer Types

SystemC LRM 7.5.2 and 7.5.3 state that the finite precision *sc_int/sc_uint* *shall* be held in an implementation-dependent native C++ integer which *shall* have a minimum representation size of 64 bits. For synthesis, representation sizes greater than 64-bits are ***Not Supported***.

In summary, the following types have ***Restricted Supported***.

- *sc_int<W>*: limited precision signed integer ($W \leq 64$).
- *sc_uint<W>*: limited precision unsigned integer ($W \leq 64$).

Support restrictions on operators and functions are covered in Section 6.3.5. Many operators are available through implicit conversions to *int_type* and *uint_type* and their support is determined by the support of those native C++ types.

6.3.2 Finite Precision Integer Types

The following Finite Precision Integer types have ***Restricted Support***.

- *sc_bigint<W>*: finite precision signed integer.
- *sc_biguint<W>*: finite precision unsigned integer.

The support restrictions of common operators and functions are covered in Section 6.3.5.

6.3.3 Finite Precision Fixed-point Types

The fixed-point types *sc_fixed* and *sc_ufixed* have ***Restricted Support*** as listed below.

1. Overflow modes have ***Restricted Support*** as specified in Table 2 below as a function of the two template parameters that determine the overflow mode: *o_mode* and *n_bits*.
2. All Quantization modes are ***Supported*** as specified in Table 3 below as a function of the template parameter that determines the quantization mode: *q_mode*.
3. Common operators/functions: see Section 6.3.5.
4. Specific member functions:
    a. Query of parameters: *wl*, *iwl*, *q_mode, o_mode*, and *n_bits* are ***Supported***.
    b. Query of value: *is_neg, is_zero,* and *value* are ***Not Supported***.

   c. Other member functions such as *overflow_flag*, *quantization_flag*, *type_params*, and *cast_switch* are ***Not Supported***.
5. Simulation specific functionality such as *sc_fxtype_param*, *sc_fxcast_switch*, and *SC_OFF* are ***Not Supported***.

Table 2: Overflow Modes

Overflow Mode	Parameters		Support
	o_mode	**n_bits**	
Wrap-around Basic (*default*)	*SC_WRAP*	0	***Supported***
Saturation	*SC_SAT*	-	***Supported***
Symmetrical Saturation	*SC_SAT_SYM*	-	***Supported***
Saturation to Zero	*SC_SAT_ZERO*	-	***Supported***
Wrap-around Advanced	*SC_WRAP*	> 0	***Not Supported***
Sign Magnitude Wrap-Around	*SC_WRAP_SM*	≥ 0	***Not Supported***

Table 3: Quantization Modes

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

6.3.4 Logic and Vector Types

6.3.4.1 Finite Word-Length Bit Vectors (sc_bv)

The finite word-length bit vector *sc_bv* has ***Restricted Support***. The restrictions on common operators/functions are outlined in Section 6.3.5.

6.3.4.2 Single-Bit Logic (sc_logic) and Finite Word-Length Logic Vectors (sc_lv)

The single-bit logic *sc_logic* and the finite world-length logic vector *sc_lv* have ***Restricted Support***. The restriction on common operators/functions are outlined in Section 6.3.5.
In addition, the following restrictions apply.

- The unknown logic constant (*sc_logic ("X"), SC_LOGIC_X*) is ***Not Supported***.
- The high-impedance logic constant (*sc_logic ("Z"), SC_LOGIC_Z*) is ***Not Supported***.

6.3.5 Common Operators and Functions

6.3.5.1 Bit Select Operator

The bit select operators as specified by SystemC LRM 7.2.5 have ***Restricted Support***. The restriction is that the index is within the bounds of the object being accessed.

NOTE
An out-of-bound access will be treated as an error by synthesis as defined in Section 1.6.3.

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

As defined by SystemC LRM 7.5.4.6, *sc_int*/*sc_uint* temporary values cannot have bit-select applied.

As defined by SystemC LRM 7.7 and 7.5.7, the bit select on concatenations or subreferences also cannot be performed.

Example:
```
  sc_int< 8 > x ;
  sc_bigint< 8 > y ;
  x[3] = y[2]     // Legal
  (x+x)[3] = 0 ;  // Illegal, as x+x is promoted to a native C++ type
  (y+y)[3] = 0 ;  // Legal as y+y is still a sc_bigint
  (y,y)[3] = 0 ;  // Illegal as concatenation doesn’t support bitref
```

6.3.5.2 **Part Select Operator**

The part select operators (*operator(l,r)* and *range(l,r)*), as specified by SystemC LRM 7.2.6, have ***Restricted Support***. The first restriction is both the left and the right index positions lie within the bounds of the object. The second restriction is the range of the part select has a statically determinable length. The third restriction is the left hand index is greater or equal to the right hand index.

Out-of-bound access *shall* be treated as an error (Section 1.6.3), as specified in the SystemC LRM.

NOTE

The third restriction implies bit-reversal using part selects is not supported. Given the second restriction, the third restriction becomes trivial to check.

The bit-reversal behavior is specified for fixed-point types, as specified in SystemC LRM 7.10.5. It is also implied for vector types in the specification of *reversed* in SystemC LRM 7.9.8.7. Part select is not allowed to reverse bit-order for limited-precision integers, as stated in NOTE1 in SystemC LRM 7.2.6. The SystemC LRM does not explicitly mention whether part selects are allowed to reverse bit-order for finite-precision integers.

Example of statically determinable range length:
```
      x(i+5,i+3) = y(k+4,k+2);  // range length = 3
```

NOTE

As defined by SystemC LRM 7.5.4.6, *sc_int*/*sc_uint* temporary values cannot have part-select applied.

Part select is not available for concatenations and subrefs of integer types. Part select is available for concatenations and subrefs of vector types (*sc_bv* and *sc_lv*).

Example:

```
  sc_int< 8 > x ;
  sc_bigint< 8 > y ;
  x(5,3) = y(4,2);    // Legal
  (x+x)(5,3) = 0 ;    // Illegal: x+x is promoted to native C++ type
  (y+y)(5,3) = 0 ;    // Legal as y+y is still a sc_bigint
  (y,y)(5,3) = 0 ;    // Illegal: concatenation of bitref not allowed
```

The result of a part select cannot be directly assigned to a fixed point variable, but it can be assigned to a range.

6.3.5.3 Function concat(C1,C2) and operator,(C1,C2)

The concatenation function and operator are ***Supported***.

NOTE

The SystemC datatype package defines concatenation functionality via a template specialized function *concat(C1,C2)* and *operator,(C1,C2)*.

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

Example:

```
     (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 not recommended. For example, using the Accellera Open Source simulator:

```
     sc_int<4> t = 1;     sc_bigint<4> tb = 1;
     sc_int<4> x = 2;    sc_bigint<4> xb = 2;

     cout << (t, t*x)  << endl;      // result = 3
     cout << (t, tb*xb) << endl;   // result = 258

     cout << (t, x >> 1) << endl;  // result = 3
     cout << (t, xb >> 1) << endl;  // result = 17
```

6.3.5.4 Reduction Operators

The reduction operators specified in SystemC LRM 7.2.8 are ***Supported***.

NOTE

The reduce operators are: *and_reduce*, *or_reduce*, *xor_reduce, nand_reduce*, *nor_reduce*, and *xnor_reduce*.

The reduction operators are not available for the fixed-point datatypes.

6.3.5.5 Arithmetic Operators

The arithmetic operators for limited and finite precision integers, as defined in the SystemC LRM, are ***Supported***.

The arithmetic operators for the finite precision fixed-point types, as defined in the SystemC LRM, have ***Restricted Support***. The implementation-dependent behavior described in SystemC LRM 7.10.6 (variable-precision fixed-point value limits) is ***Not Supported***. A consequence is that the division operator is ***Not Supported***.

NOTE
Examples of implementation-dependent behavior described the SystemC LRM 7.10.6 can be found in the Accellera open source simulator in the form of the following compiler flags: • *SC_FIXDIV_WL:* bounds the number of bits that are computed for division. Division can require an infinite number of bits to represent. • *SC_FXMAX_WL*: is the maximum width of a fixed-point type. It is presumed precision is set high enough so it does not change the behavior of the *design*. Synthesis presumes these limits are not present. The behavior of operators `++` and `–` for fixed-point datatypes, while using the standard increment/decrement by `1`, is in effect a NOP whenever *iwl* > *wl* and in some cases of *iwl* < `1`. The arithmetic operators are defined for the numeric types (integer and fixed-point types). The arithmetic operators include: • Unary operators `+` and `–` • Binary operators `+`, `-`, `*`, `/`, and `%`; assign operators `+=`, `-=`, `*=`, `/=`, and `%=`. The operators `%` and `%=` are not available for fixed-point types. • Prefix and Postfix increment and decrement operators `++` and `--`.

6.3.5.6 Bitwise Complement Operator

The unary bitwise complement operator ~ specified in the SystemC LRM is ***Supported***.

NOTE

The unary ~ operator complements its argument bitwise. For vector types, the width of the return type is identical to the width of the operand. This is consistent with the vector types not having an arithmetic view (no signed/unsigned treatment). The integer and fixed-point types (these are referred to in SystemC LRM 8.1 as *numeric types*) are intended to have an arithmetic treatment. In an arithmetic context, `~x` is equal to `(-x-1)`. However, the unary operators ~ for the SystemC integer and fixed-point types are not consistent with an arithmetic treatment and are in fact inconsistent amongst themselves:

```
    cout << ~((sc_uint<8>) 128) << endl;
        // 18446744073709551487
    cout << ~((sc_biguint<8>) 128) << endl;     //   383
    cout << ~((sc_ufixed<8,8>) 128) << endl;    //  127
    cout << ~((unsigned char) 128) << endl;
        //  -129, correct  ~x = (-x-1)
```

6.3.5.7 Bitwise Logical Operators

The binary bitwise logical operators (&, `|`, and `^`) and bitwise assignment operators (&=, `|=`, and `^=`) specified in the SystemC LRM are ***Supported***.

NOTE
The binary operations &, |, ^ compute the bitwise and, or and xor operation respectively. The mixing of signed an unsigned operands is not allowed for fixed-point types (a difference compared to SystemC integer types).

6.3.5.8 **Logical Negation**

The logical negation operator `!` specified in the SystemC LRM is ***Supported***.

NOTE
The operator `!` is available for *sc_int_bitref*, *sc_uint_bitref*, *sc_signed_bitref_r*, and *sc_unsigned_bitref_r*. Using the operator `!` on fixed-point types leads to implicit conversions, so there *may* be unexpected truncation involved. The operator `!` is available on limited precision types through implicit conversions to *int_type* (C++ native type).

6.3.5.9 **Relational Operators**

Relational Operators, as defined in the SystemC LRM, are ***Supported***.

NOTE
The relational operators compare the two operands as in C++ and return a value of type *bool*. The comparison is done arithmetically for integer and fixed-point types. The relational operators `==` and `!=` operators are available for vector types. Mixing signed and unsigned limited precision integers can lead to unexpected results. For example: `(sc_uint<8>) 1 > (sc_int<8>) -1  // incorrectly returns false`

6.3.5.10 **Shift Operators**

6.3.5.10.1 Limited Precision Integers

The shift operators (<< and >>) and shift assign operators (<<= and >>=) have ***Restricted Support***. The restriction is the second operand needs to be in the range `0` to `63`.

NOTE
The restriction is inherited from the implicit conversion to the native C++ 64-bit integers in its implementation (see Section 6.1.3 and SystemC LRM 7.5.4.6). This restriction is also tied to the restriction in Section 6.3.2 on the maximum bitwidth for limited precision integers. Outside the range `0` – `63`, the behavior is undefined (see Section 1.5.2).

6.3.5.10.2 Finite Precision Integers

The shift operators (<< and >>) and shift assign operators (<<= and >>=) have ***Restricted Support***. The first restriction is the second operand has to have a non-negative value. The support for the left shift operator (<<) is also restricted to have a statically determinable return bit-width, which implies the value of second operand is statically determinable.

The restrictions above also apply to the said operators applied on the base types (*sc_value_base*, *sc_signed*, and *sc_unsigned*), and the types resulting from part selects (*sc_signed_subref_r*, *sc_signed_subref*, *sc_unsigned_subref_r*, and *sc_unsigned_subref*) and concatenations (*sc_concatref*).

NOTE
SystemC LRM 7.6.3.7 specifies the behavior is undefined (see Section 1.6.2) if the second operand is negative.

6.3.5.10.3 Finite Precision Fixed-Point Types

The shift operators (<<= and >>=) have ***Restricted Support***. The restriction is the length and integer length of the return type have to be statically determinable, which implies the value of the second operand is statically determinable.

The shift assign operators (<<= and >>=) have ***Restricted Support***. The restriction is the value of the second operand is statically determinable for all cases other than when the first operand has *qmode=SC_TRN*, *omode=SC_WRAP*, and *n_bits=0*. This restriction guarantees no hardware is required for handling rounding and/or saturation.

The restrictions above also apply to the said operators applied on the base types (*sc_fxval*, *sc_fixnum*, *sc_fix*, and *sc_ufix*) and types resulting from part selects (*sc_fxnum_subref*).

NOTE
Fixed-point shifts are bidirectional (though this is not explicitly stated in the SystemC LRM) as the second argument is a C++ *int*. The hardware cost can be minimized if the range of the second argument can be reduced. For example, if the range analysis of the *synthesis tool* can determine that the second argument is non-negative, then a unidirectional shift suffices.

6.3.5.10.4 Finite Word-Length Vector Types

The shift operators (<< and >>) and shift assign operators (<<= and >>=) have ***Restricted Support***. The first restriction is the second operand has to have a non-negative value. The support for the left shift operator (<<) is also restricted to have a statically determinable return bit-width, which implies the value of second operand is statically determinable.

The restrictions above also apply to the said operators applied on the base types (*sc_lv_base* and *sc_bv_base*), and the types resulting from part selects (*sc_subref_r* and *sc_subref*) and concatenations (*sc_concref_r* and *sc_concref*).

NOTE
SystemC LRM 7.9.3.7, 7.9.4.7, and 7.9.8.6 specify it is an error (Section 1.6.3) if the second operand is negative. The SystemC LRM is silent on what happens when the second operand is negative for the classes resulting from concatenation (*sc_concref_r* and *sc_concref*).

6.3.5.11 Rotate Operators

The rotate operators *lrotate* and *rrotate* have ***Restricted Support***. The first restriction is the argument (amount what which to rotate) is not negative. The second restriction is the value of the argument is statically determinable.

NOTE
The rotate operators are available for the vector types (*sc_lv* and *sc_bv*), their base classes (*sc_lv_base* and *sc_bv_base*), and the types resulting from part select (*sc_subref*) and

concatenations (*sc_concref*) of vector classes.

The SystemC LRM does not explicitly define the behavior for negative values.

6.3.5.12 **Explicit Conversions**

Explicit conversions have ***Restricted Support*** as outlined below.

- Conversions *to_int*, *to_uint*, *to_long*, *to_ulong*, *to_int64,* and *to_uint64* are ***Supported***.
- The following conversions are ***Not Supported***:
    - *to_float* and *to_double*;
    - *to_char* (conversion to character for *sc_logic* and bit select for vector types);
    - *to_string* (and its shortcut variants *to_dec*, *to_hex*, *to_oct*, and *to_bin*).

- Conversion *to_bool* are
    - ***Not Supported*** for types *sc_logic* and vector types;
    - ***Supported*** for bit-select for the limited and finite precision integers.
- Member function *is_01* (the SystemC LRM labels this as a explicit conversion function) is ***Not Supported***.
- Member function *value* (for *sc_logic* and vector types) is ***Not Supported***.

6.3.5.13 **Conversion Operators**

The conversion operators have ***Restricted Support***. The restriction is the implicit conversion operator is to a supported type.

NOTE

The following conversion operators are defined in the SystemC LRM, along with the classes that define them.

- operator *int_type()*: limited precision integers.
- operator *uint_type()*: limited precision integers.
- operator *uint64()*: *sc_int_bitref_r*, *sc_uint_bitref_r*, *sc_signed_bitref_r*, *sc_unsigned_bitref_r,* and *sc_concatref.*
- operator *sc_unsigned()*: *sc_signed_subref_r*, *sc_unsigned_subref_r,* and *sc_concatref*.
- operator *sc_logic()*: *sc_bitref_r*.
- operator *double()*: *sc_fxnum* and *sc_fxval*.
- operator *bool()*: *sc_fxnum_bitref*.
- operator *sc_bv_base()*: *sc_fxnum_subref*.

6.3.5.14 **Assignment Operators and Constructors**

Assignment operators and constructors have ***Restricted Supported*** as described below.

Initialization through String Literals, as specified in SystemC LRM 7.3, has ***Restricted Support***. Initialization of numeric or vector type object with a string literal as a parameter of the copy constructor is ***Supported***, but an initialization through a variable of *char* or *std::string* type is ***Not Supported***. The *sc_numrep* of *SC_NOBASE* (implementation-defined prefix or missing prefix) is ***Not Supported***.

The SystemC LRM introduces the concept of case insensitivity to the prefix and magnitude representations. These are **Supported**.

NOTE
In accordance with the SystemC definition of string literals, SystemC LRM 7.3, a literal representation may be used as the value of a SystemC numeric or vector type object. It consists of a standard prefix, followed by a magnitude expressed as one or more digits.

sc_numrep	**Prefix (case insensitive)**	**Magnitude format**
SC_NOBASE	Implementation-defined	Implementation-defined
SC_DEC	0d	Decimal Number, digits 0-9
SC_BIN	0b	Binary Number, digits 0-1
SC_BIN_US	0bus	Binary Unsigned
SC_BIN_SM	0bsm	Binary Sign & Magnitude
SC_OCT	0o	Octal Number, digits 0-7
SC_OCT_US	0ous	Octal Unsigned
SC_OCT_SM	0osm	Octal Signed & Magnitude
SC_HEX	0x	Hexadecimal Number, digits 0-9, a-f, A-F
SC_HEX_US	0xus	Hexadecimal Unsigned
SC_HEX_SM	0xsm	Hexadecimal Sign & Magnitude
SC_CSD	0csd	Canonical Signed Digit

6.3.5.15 **String input and output**

- ***Not Supported*** for *dump( )*, *print( )*, and *scan( )*. See SystemC LRM 7.2.10 and 7.2.11.
- ***Not Supported*** for *to_string( )*. See SystemC LRM 7.3.
- ***Not Supported*** for *String Shortcut Methods* (*to_dec( )*, *to_bin()*, and so on). See SystemC LRM 7.10.8.1.

6.3.5.16 **Length**

The member function *length* is ***Supported***.

6.3.5.17 **Reversed**

The member function *reversed* is ***Not Supported***.

NOTE
This is consistent with the restriction on bit-reversal in Section 6.3.5.2.

7 Declarations

Declarations, as defined for ISOC++ Section 7, have ***Restricted Support*** and those specific restrictions and coding guidelines are listed in the subsequent sections and chapters.

7.1 Specifiers

Specifiers (ISOC++ 7.1) have ***Restricted Support***.

7.1.1 Storage class specifiers

Storage class specifiers (ISOC++ 7.1.1) have ***Restricted Support***.

1. The *auto* and *register* specifiers are hints to a C++ compiler and are ***Ignored.***
2. The *mutable* specifier is ***Supported***.
3. The *extern* specifier has ***Restricted Support***, as specified in Section 2.1.
4. The *static* specifier has ***Restricted Support***.
   a. Static <u>*class*</u> definitions are ***Supported***.
   b. Static variables have ***Restricted Support***. Only *const* static variables are supported.
   c. Static functions and static member functions are ***Supported***.
   d. Static data members of <u>*class*</u> have ***Restricted Support***. Only *const* static data members of <u>*class*</u> are ***Supported***.

Examples:

```
static class my_class c ;             // Static class definition,
supported
static void my_func( void ) { }   // Static function definition,
supported
static int var;    // Non-const, not supported
static const int const_var = 5;        // Const, supported
struct my_struct {
    static int member;             // Non-const, not supported
    static const int const_member; // Const, supported
    static void my_member_func( void ) { }
                                   // Static member function
};
```

7.1.2 Function specifiers

The function specifiers, as defined in ISOC++ 7.1.2, have ***Restricted Support***.

- 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 12.10.3.

7.1.3 The typedef specifier

Supported as defined in ISOC++ 7.1.3.

7.1.4 The friend specifier

Supported as defined in ISOC++ 7.1.4.

7.1.5 Type specifiers

Type specifiers (ISOC++ 7.1.5) have ***Restricted Support***.

7.1.5.1 **cv-qualifiers**

The cv-qualifiers (ISOC++ 7.1.5.1) have ***Restricted Supported***.

1. The cv-qualifier *const* is ***Supported***
2. The cv-qualifier *volatile* is ***Not Supported***.

7.1.5.2 **Simple type specifiers**

Simple type specifiers (ISOC++ 7.1.5.2) have ***Restricted Support***. They are ***Supported*** for types that are ***Supported*** as specified in Section 6.

7.1.5.3 **Elaborated type specifiers**

Elaborated type specifiers (ISOC++ 7.1.5.3) are ***Supported***.

7.1.6 Enumerations

Supported as defined in ISOC++ 7.2.

7.1.7 The asm declaration

Not supported.

7.1.8 Linkage specifications

External linkage (as described in ISOC++7.5) is ***Not Supported***. See Section 2.

8 Declarators

Declarators, as defined in ISOC++ Section 8, have ***Restricted Support***.

8.1 Type names

Supported as defined in ISOC++ 8.1.

8.2 Ambiguity resolution

Supported as defined in ISOC++ 8.2.

8.3 Kinds of declarators

8.3.1 Pointers

Pointers have ***Restricted Support***. Pointers that are statically determinable are ***Supported***. Otherwise, they are ***Not Supported***. Statically determinable implies the *synthesis tool* 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 *shall* also be statically determinable.

Using the value of a pointer as data is ***Not Supported***. This includes, for instance, testing that a pointer is zero (`0`) or hashing on a pointer.

8.3.2 References

Supported as defined in ISOC++ 8.3.2.

8.3.3 Pointers to Nonstatic Class Members

Pointers to members using `::*` and access via `->*` and `.*` have ***Restricted Support***. They are ***Supported*** described in ISOC++ 8.3.3 and 5.5, subject to the restrictions described in Section 8.3.1 on pointers.

8.3.4 Arrays

The element type of an array *shall* be any of the types which are permitted by ISOC++ as element types, 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 ISOC++.

Any declaration of an array *shall* include the specification of its bound, either explicitly, if no initializer is specified, or as an implication from the initializer, if such is specified.

8.3.5 Function parameters

Function parameters (ISOC++ 8.3.5) have ***Restricted Support***. The restriction is ellipsis (...) function parameters are ***Not Supported***.

8.3.6 Default arguments

Default arguments (ISOC++ 8.3.6) are ***Supported***.

8.4 Function definition

Function definitions (ISOC++ 8.4) are ***Supported***.

8.5 **Initializers**

Initializers (ISOC++ 8.5) are ***Supported***.

8.5.1.1 **Aggregates**

Initialization of aggregates (ISOC++ 8.5.1) is ***Supported***.

8.5.1.2 **Character arrays**

Initialization of character arrays (ISOC++ 8.5.2) is ***Supported***. Also see Section 6.1.1.1 for character literals that are supported.

8.5.1.3 **References**

Initialization of references is ***Supported***, as defined in ISOC++ 8.5.3.

9 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.

NOTE
User code should not use such expressions to avoid differences in the results between simulation using a particular compiler and synthesis using a particular synthesis tool.

9.1 Function call

Recursive functions (a function which includes a call to itself either directly or indirectly) are ***Not Supported***.

9.2 Explicit type conversion

An explicit type conversion (ISOC++ 5.4) can be expressed using a type conversion operator, a functional notation, or cast notation. An explicit type conversion of non-pointer type is ***Supported***. On the other hand, an explicit type conversion of pointer is ***Supported with Restrictions***.

9.2.1 Type conversion operators

9.2.1.1 Dynamic cast

The type conversion operator function *dynamic_cast<T>(v)* (ISOC++ 5.2.7) is ***Not Supported***.

9.2.1.2 Static cast

The type conversion operator function *static_cast<T>(v)* (ISOC++ 5.2.9) is ***Supported*** within the limitations above.

9.2.1.3 Reinterpret cast

The type conversion operator function *reinterpret_cast<T>(v)* (ISOC++ 5.2.10) is ***Not Supported***.

9.2.1.4 Const cast

The type conversion operator function *const_cast<T>(v)* (ISOC++ 5.2.11) is ***Supported*** within the limitations above.

9.2.2 Functional notation

An explicit type conversion of pointer type using functional notation is ***Supported with Restrictions***. Each type conversion of pointer type using functional notation can be mapped to any of type conversion operators described in Section 9.2.1. The restriction is the same as those of the type conversion operators.

9.2.3 Cast notation

An explicit type conversion using cast notation is ***Supported with Restrictions***. Each type conversion of pointer type using cast notation can be mapped to any of type conversion operators described in Section 9.2.1. The restriction is the same as those of the type conversion operators, which are equivalent to the given explicit type conversion using cast notation.

9.3 typeid

The type identification function *typeid* (ISOC++ 5.2.8) is ***Not Supported***.

9.4 Unary Expressions and Operators

9.4.1 Unary Operators

The unary operators (ISOC++ 5.3.1) of the form `*  &  +  -  !` and `~` are ***Supported***.

9.4.2 Increment and decrement

Supported (ISOC++ 5.3.2).

9.4.3 sizeof

The *sizeof* operator (ISOC++ 5.3.3) is ***Not Supported***.

9.4.4 New and delete

The *new* operator (ISOC++ 5.3.4) has ***Restricted Support***. It is ***Supported*** for the instantiation of variable arrays and *SC_MODULE*. It is ***Not Supported*** for the instantiation of types derived from the *sc_object* type, other than *SC_MODULE*. The *new* operator *may* appear in the constructor or constructor initializer list within *SC_MODULE* only and that use of *new* is ***Not Supported*** anywhere within the behavioral description, such as inside *SC_METHOD*, *SC_THREAD*, or *SC_CTHREAD*. When allocating an array of objects using *new*, the number of elements *shall* be statically determinable.

The *delete* operator (ISOC++ 5.3.5) is ***Not Supported***.

The use of *set_new_handler( )* *shall* be ***Ignored***.

Overloading the *new* or *delete* operators are ***Not Supported***. The *placement new* construct is ***Not Supported***.

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;
    unsigned int *mem ;

    SC_CTOR(MyModule): add("add"), mem(new unsigned[128])  {
      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);
     }
};
```

9.5 Pointer-to-member operators

The pointer-to-member operators `->*` and `.*` have ***Restricted Support*** (ISOC++ 5.5). The restrictions are based on the ***Restricted Support*** of pointers as defined in Section 8.3.1.

9.6 Multiplicative, Additive, Shift, Relational, Equality, and Assignment operators

Supported (ISOC++ 5.6, 5.7, 5.8, 5.9, 5.10, and 5.17).

9.7 Bitwise and Logical AND/OR/XOR operators

Supported (ISOC++ 5.11 thru 5.15).

9.8 Conditional Operator

Supported (ISOC++ 5.16).

9.9 Comma operator

Supported (ISOC++ 5.18).

10 Statements

Statements (as defined in ISOC++ Section 6) have ***Restricted Support***.

10.1 Labeled statement

Labels are ***Supported*** as defined in ISOC++ 6.1.

10.2 Compound statement

The *compound* statement (also, and equivalently, called *block*) is supported, as defined in ISOC++ 6.3, to group sets of statements together.

10.3 Selection statements

Selection statements include *if*, *if-else*, and *switch* statements.
Selection statements are ***Supported*** as defined in ISOC++ 6.4.

10.3.1 The if statement

Supported as defined in ISOC++ 6.4.1.

10.3.2 The switch statement

Supported as defined in ISOC++ 6.4.2;

10.4 Iteration statements

The iteration statements *while*, *do*, and *for* are ***Supported*** as defined in ISOC++ 6.5.

10.5 Jump statements

Jump Statements (ISOC++ 6.6) have ***Restricted Support*** as described below.

10.5.1 The break statement

The *break* statement (as defined in ISOC++ 6.6.1) has ***Restricted Support***.
The restriction is specified in Section 4: a *break* that exits the infinite loop of an *SC_CTHREAD* or *SC_THREAD* process is ***Not Supported***.

10.5.2 The continue statement

The *continue* statement (as defined in ISOC++ 6.6.2) is ***Supported***.

10.5.3 The return statement

The *return* statement (as defined in ISOC++ 6.6.3) has ***Restricted Support***.
The restriction is specified in Section 4: a *return* statement that occurs within the function defining an *SC_CTHREAD* or *SC_THREAD* process is ***Not Supported***.

10.5.4 The goto statement

The *goto* statement (as defined in ISOC++ 6.6.4) is ***Not Supported***.

10.6 Declaration statement

Declaration statements are ***Supported*** as defined in ISOC++ Section 7.

10.7 **Exception handling statements**

The exception handling mechanisms (as defined in ISOC++ Section 15) are ***Not Supported***. These include the *try*, *catch()*, and *throw* statements, and the special functions *terminate()*, *unexpected()*, and *uncaught_exception()*.

11 Namespaces

namespaces (as defined in ISOC++ 7.3) are ***Supported.*** Non-const global variables are ***Not Supported*** for synthesis. Within a function body, only those names of variables *shall* be used, which are declared previously in the function body or passed as parameters. Global constants are ***Supported*** for synthesis.

Examples:

```
// Example Namespace
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 Namespace definition

Supported as defined in ISOC++ 7.3.1.

11.1.1 Unnamed namespaces

Supported as defined in ISOC++ 7.3.1.1.

11.1.2 Namespace member definitions

Supported as defined in ISOC++ 7.3.1.2.

11.2 Namespace alias

Supported as defined in ISOC++ 7.3.2.

11.3 The using declaration

Supported as defined in ISOC++ 7.3.3.

11.4 Using directive

Supported as defined in ISOC++ 7.3.4.

12 Classes

Restrictions pertaining to classes defined in the SystemC LRM are specified in the respective Section in this standard.
This Section describes the support for user-defined classes.

Restricted Support as described below (ISOC++ Section 9).

12.1 Class names

Supported as defined in ISOC++ 9.1.

12.2 Class members

Supported as defined in ISOC++ 9.2.

12.3 Member functions

Supported as defined in ISOC++ 9.3.

12.3.1 Nonstatic member functions

Supported as defined in ISOC++ 9.3.1.

12.3.2 The `this` pointer

Supported as defined in ISOC++ 9.3.2.

12.4 Static members

Restricted Support.

12.4.1 Static member functions

Supported (ISOC++ 9.4.1) as specified in Section 7.1.1.

12.4.2 Static data members

Static data members (ISOC++ 9.4.2) have ***Restricted Support*** as specified in Section 7.1.1.

12.5 Unions

Not supported (ISOC++ 9.5).

12.6 Bit-fields

Bit-fields, as defined in ISOC++ 9.6, are ***Not Supported***.

12.7 Nested class declarations

Supported as defined in ISOC++ 9.7.

12.8 Local class declarations

Supported as defined in ISOC++ 9.8.

12.9 Nested type names

Supported as defined in ISOC++ 9.9.

12.10 **Derived classes**

Supported as defined in ISOC++ Section 10.

12.10.1 Multiple base classes

Supported as defined in ISOC++ 10.1.

12.10.2 Member name lookup

Supported as defined in ISOC++ 10.2.

12.10.3 Virtual functions

Virtual functions, as defined in ISOC++ 10.3, have ***Restricted Support*** provided that when such functions are called, the type of the "*this*" object can be statically determined.

12.10.4 Abstract classes

Supported as defined in ISOC++ 10.4.

12.11 **Member access control**

Supported as defined in ISOC++ Section 11.

12.11.1 Access specifiers

Supported as defined in ISOC++ 11.1.

12.11.2 Accessibility of base classes and base class members

Supported as defined in ISOC++ 11.2.

12.11.3 Access declarations

Supported as defined in ISOC++ 11.3.

12.11.4 Friends

Supported as defined in ISOC++ 11.4.

12.11.5 Protected member access

Supported as defined in ISOC++ 11.5.

12.11.6 Access to virtual functions

Supported as defined in ISOC++ 11.6.

12.11.7 Multiple access

Supported as defined in ISOC++ 11.7.

12.11.8 Nested classes

Supported as defined in ISOC++ 11.8.

12.12 **Special member functions**

Restricted Support as described below (ISOC++ Section 12).

12.12.1 Constructors

Constructors of user defined classes (as defined in ISOC++ 12.1) are ***Supported***.

12.12.2 Temporary objects
Supported as defined in ISOC++ 12.2.

12.12.3 Conversions
Supported as defined in ISOC++ 12.3.

12.12.3.1 **Conversion by constructor**
Supported as defined in ISOC++ 12.3.1.

12.12.3.2 **Conversion functions**
Supported as defined in ISOC++ 12.3.2.

12.12.4 Destructors
Supported as defined in ISOC++ 12.4.

12.12.5 Free store
Not supported (ISOC++ 12.5).

12.12.6 Initialization
Supported (ISOC++ 12.6).

12.12.6.1 **Explicit initialization**
Explicit initialization (as defined in ISOC++ 12.6.1) is ***Supported***.

12.12.6.2 **Initializing bases and members**
Supported as defined in ISOC++ 12.6.2.

12.12.7Copying class objects
Supported as defined in ISOC++ 12.8.

13 Overloading

Restricted Support as described below (ISOC++ Section 13).

13.1 Overloadable declarations

Supported as defined in ISOC++ 13.1.

13.2 Declaration matching

Supported as defined in ISOC++ 13.2.

13.3 Overload resolution

Supported as defined in ISOC++ 13.3.

13.3.1 Candidate functions and argument lists

Supported as defined in ISOC++ 13.3.1.

13.3.1.1 Function call syntax

Supported as defined in ISOC++ 13.3.1.1.

13.3.1.1.1 Call to named function

Supported as defined in ISOC++ 13.3.1.1.1.

13.3.1.1.2 Call to object of class type

Supported as defined in ISOC++ 13.3.1.1.2.

13.3.1.2 Operators in expressions

Supported as defined in ISOC++ 13.3.1.2.

13.3.1.3 Initialization by constructor

Supported as defined in ISOC++ 13.3.1.3.

13.3.1.4 Copy-initialization of class by user-defined conversion

Supported as defined in ISOC++ 13.3.1.4.

13.3.1.5 Initialization by conversion function

Supported as defined in ISOC++ 13.3.1.5.

13.3.1.6 Initialization by conversion function for direct reference binding

Supported as defined in ISOC++ 13.3.1.6.

13.3.2 Viable functions

Supported as defined in ISOC++ 13.3.2.

13.3.3 Best Viable Function

Supported as defined in ISOC++ 13.3.3.

13.3.3.1 Implicit conversion sequences

Supported as defined in ISOC++ 13.3.3.1.

13.3.3.1.1 Standard conversion sequences
Supported as defined in ISOC++ 13.3.3.1.1.

13.3.3.1.2 User-defined conversion sequences
Supported as defined in ISOC++ 13.3.3.1.2.

13.3.3.1.3 Ellipsis conversion sequences
Not Supported (ISOC++ 13.3.3.1.3).

13.3.3.1.4 Reference Binding
Supported as defined in ISOC++ 13.3.3.1.4.

13.3.3.2 Ranking implicit conversion sequences
Supported as defined in ISOC++ 13.3.3.2.

13.4 Address of overloaded function
Address operations are ***Not Supported*** (ISOC++ 13.4).

13.5 Overloaded operators
Supported as defined in ISOC++ 13.5.

13.5.1 Unary operators
Supported as defined in ISOC++ 13.5.1.

13.5.2 Binary operators
Supported as defined in ISOC++ 13.5.2.

13.5.3 Assignment
Supported as defined in ISOC++ 13.5.3.

13.5.4 Function call
Supported as defined in ISOC++ 13.5.4.

13.5.5 Subscripting
Supported as defined in ISOC++ 13.5.5.

13.5.6 Class member access
Supported as defined in ISOC++ 13.5.6.

13.5.7 Increment and decrement
Supported as defined in ISOC++ 13.5.7.

13.6 Built-in operators
Supported as defined in ISOC++ 13.6.

14 Templates

Supported as described below (ISOC++ Section 14).

14.1 **Template parameters**

Supported (ISOC++ 14.1).

14.2 **Names of template specializations**

Supported as defined in ISOC++ 14.2.

14.3 **Template arguments**

Supported as defined in ISOC++ 14.3.

14.3.1 Template type arguments

Supported as defined in ISOC++ 14.3.1.

14.3.2 Template non-type arguments

Supported as defined in ISOC++ 14.3.2.

14.3.3 Template template arguments

Supported as defined in ISOC++ 14.3.3.

14.4 **Type equivalence**

Supported as defined in ISOC++ 14.4.

14.5 **Template declarations**

Supported as defined in ISOC++ 14.5.

14.5.1 Class Templates

Supported as defined in ISOC++ 14.5.1.

14.5.1.1 **Member functions of class templates**

Supported as defined in ISOC++ 14.5.1.1.

14.5.1.2 **Member classes of class templates**

Supported as defined in ISOC++ 14.5.1.2.

14.5.1.3 **Static data members of class templates**

Restricted Support (ISOC++ 14.5.1.3). See Section 12.4.2 for static data members of classes and Section 7.1.1 for the static storage class specifier.

14.5.2 Member templates

Supported as defined in ISOC++ 14.5.2.

14.5.3 Friends

Supported as defined in ISOC++ 14.5.3.

14.5.4 Class template partial specializations
Supported as defined in ISOC++ 14.5.4.

14.5.4.1 **Matching of class template partial specializations**
Supported as defined in ISOC++ 14.5.4.1.

14.5.4.2 **Partial ordering of class template specializations**
Supported as defined in ISOC++ 14.5.4.2.

14.5.4.3 **Members of class template specializations**
Supported as defined in ISOC++ 14.5.4.3.

14.5.5 Function templates
Supported as defined in ISOC++ 14.5.5.

14.5.5.1 **Function template overloading**
Supported as defined in ISOC++ 14.5.5.1.

14.5.5.2 **Partial ordering of function templates**
Supported as defined in ISOC++ 14.5.5.2.

14.6 **Name resolution**
Supported as defined in ISOC++ 14.6.

14.6.1 Locally declared names
Supported as defined in ISOC++ 14.6.1.

14.6.2 Dependent names
Supported as defined in ISOC++ 14.6.2.

14.6.2.1 **Dependent types**
Supported as defined in ISOC++ 14.6.2.1.

14.6.2.2 **Type-dependent expressions**
Supported as defined in ISOC++ 14.6.2.2.

14.6.2.3 **Value-dependent expressions**
Supported as defined in ISOC++ 14.6.2.3.

14.6.2.4 **Dependent template arguments**
Supported as defined in ISOC++ 14.6.2.4.

14.6.3 Non-dependent names
Supported as defined in ISOC++ 14.6.3.

14.6.4 Dependent name resolution
Supported as defined in ISOC++ 14.6.4.

14.6.4.1 Point of instantiation
Supported as defined in ISOC++ 14.6.4.1.

14.6.4.2 Candidate functions
Supported as defined in ISOC++ 14.6.4.2.

14.6.5 Friend names declared within a class template
Supported as defined in ISOC++ 14.6.5.

14.7 Template instantiation and specialization
Supported as defined in ISOC++ 14.7.

14.7.1 Implicit instantiation
Supported as defined in ISOC++ 14.7.1.

14.7.2 Explicit instantiation
Supported as defined in ISOC++ 14.7.2.

14.7.3 Explicit specialization
Supported as defined in ISOC++ 14.7.3.

14.8 Function template specializations
Supported as defined in ISOC++ 14.8.

14.8.1 Explicit template argument specification
Supported as defined in ISOC++ 14.8.1.

14.8.2 Template argument deduction
Supported as defined in ISOC++ 14.8.2.

14.8.2.1 Deducing template arguments from a function call
Supported as defined in ISOC++ 14.8.2.1.

14.8.2.2 Deducing template arguments taking the address of a function template
Supported as defined in ISOC++ 14.8.2.2.

14.8.2.3 Deducing conversion function template arguments
Supported as defined in ISOC++ 14.8.2.3.

14.8.2.4 Deducing template arguments from a type
Supported as defined in ISOC++ 14.8.2.4.

14.8.3 Overloaded resolution
Supported as defined in ISOC++ 14.8.3.

15 Libraries

This section outlines support for external C++ or SystemC libraries.

15.1 Standard C and C++ Libraries

Library functions from C/C++ have ***Restricted Support***. Most of them are ***Not Supported***, while some are ***Ignored*** as described below. Note that side effects of calls to generate the arguments to an ignored function are ***Not Supported***.

15.1.1 Outputting messages to stdout, stderr, cout and/or cerr

The following constructs are ***Ignored***:

- *printf* and *fprintf* functions, and
- using *operator* << to *cout* or *cerr*.

NOTE
The functions/operators called to produce the arguments to an ignored function are not part of the ignored construct.

```
     printf(”x = %d”, x);            // Ignored
     printf(”y = %d”, ++y);
               // Side effect ++y Not Supported
     cout << ”z = ” << z << endl;    // Ignored
```

15.2 SystemC Functions and Types

15.2.1 Tracing

The following tracing constructs are ***Ignored***:

- 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*.

15.2.2 set_stack_size

Ignored.

15.2.3 sc_gen_unique_name

Ignored.

15.2.4 before_end_of_elaboration and end_of_elaboration

Not Supported.

15.2.5 start_of_simulation and end_of_simulation

Not Supported.

15.2.6 dont_initialize

Supported.

15.2.7 next_trigger

Not Supported.

15.2.8 print and dump

The SystemC functions *print* and *dump* are ***Ignored***.

15.2.9 kind

Ignored.

15.2.10 sc_report_handler

The *sc_report_handler* class has ***Restricted Support***. The member function *sc_report_handler::report* is ***Ignored***. All other member functions of *sc_report_handler* are ***Not Supported***.

Note
This implies that the macros sc_assert, SC_REPORT_FAIL, SC_REPORT_ERROR, SC_REPORT_WARNING and SC_REPORT_INFO are ignored.

Annex A Levels of Abstraction in SystemC Design and Introduction to High-level Synthesis (Informative)

A.1 Introduction to Abstraction Levels

How abstraction levels support design activities

The complexity of modern systems does not allow us to design such systems directly without modeling at a number of abstraction levels. 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.

System designers utilize a separation of function, architecture, and implementation to manage these issues, with design activities at each level of abstraction. It is possible to distinguish three main levels of abstraction: function level, architecture level, and implementation level. Each level has its own standards and methodologies. The SystemC synthesizable subset defines a standard set of language constructs for describing a System at the implementation level.

Figure A.1 shows one representation of the abstraction levels in a System Design Flow.

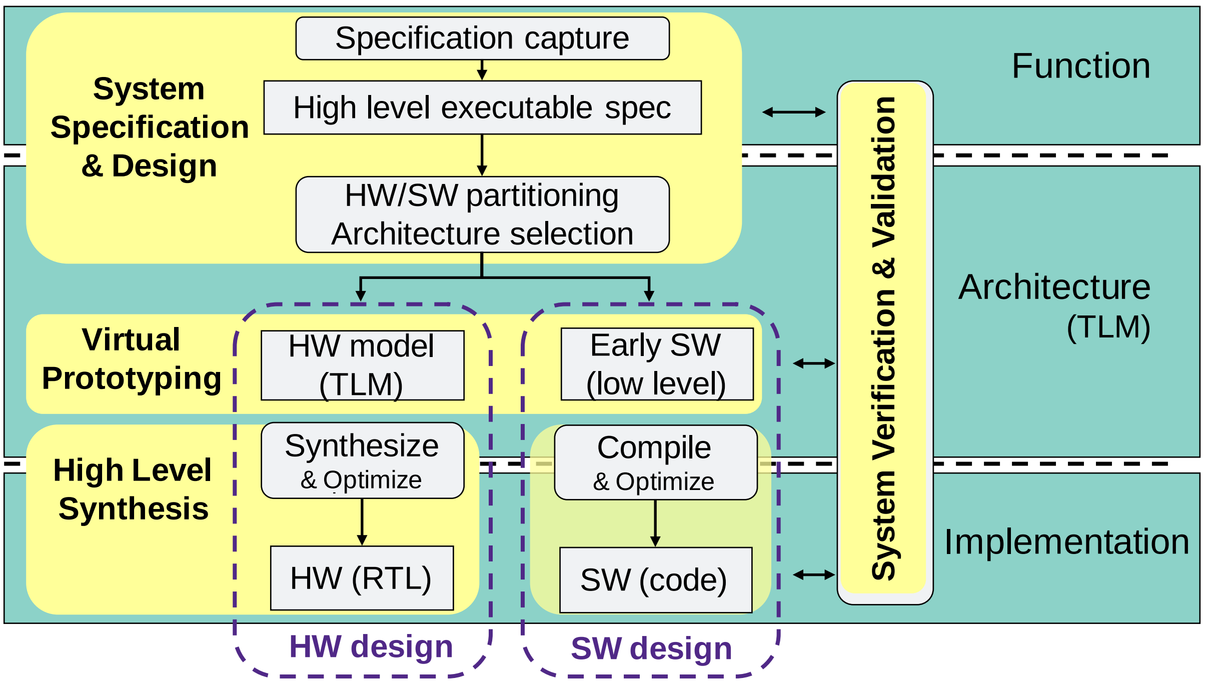


Figure A.1 Abstraction levels in a System Design Flow

The abstraction levels include:

1. The Function level in which the algorithms are defined and the system is partitioned into communicating tasks.
2. The Architecture level in which the tasks are assigned to execution units, communication mechanisms between the execution units are defined, and implementation metrics, such as performance, are modeled and estimated.

3. The Implementation level in which the precise method of implementation of the execution units is defined.

The Implementation level is further subdivided:

A. The Implementation Behavior level in which the implementation is specified in terms of an algorithm embodied in an implicit state machine.
B. The Implementation Register Transfer Level in which the implementation is specified in terms of a combination of combinatorial logic and an explicit state machine.
C. The Implementation Gate Level in which the implementation is specified in terms of technology leaf cells

A.2 Introduction to high-level synthesis

This section is focused on synthesis solutions that generate non-programmable register transfer level (RTL) cores from high-level descriptions.

High-Level Synthesis (HLS), also known as *Behavioral Synthesis*, allows designing at a level of abstraction higher than the register-transfer level by automating the translation and optimization of a behavioral description, or a high-level model, into an RTL implementation. It also 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 A.2 shows an abstract view of a representative design flow involving High-Level synthesis and Logic synthesis.

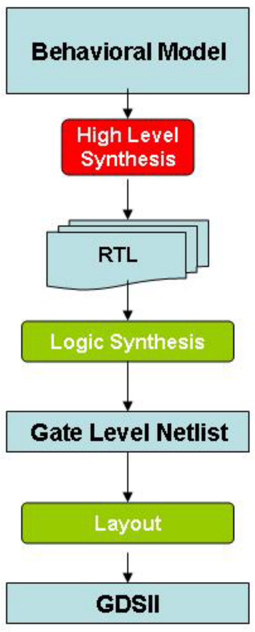


Figure A.2 Abstract view of design flow involving High-Level and Logic synthesis

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 and a set of directives that will influence the resulting architecture. These directives include, for example, timing constraints used by the algorithms of a *synthesis tool* as they create a cycle-by-cycle schedule of the required operations. In some flows, a set of low-level components is characterized at a target frequency for the selected fabrication process. This 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 to implement the desired functionality. The datapath and state machine outputs are RTL code optimized for use with conventional simulation and logic synthesis or physical synthesis tools.

To be useful to system designers, SystemC-based HLS typically needs to accommodate two different semantics of timing.

- Portions of a *design* are described in a time-independent manner - allowing the *synthesis tool* to schedule operations for specific clock cycles.
- Portions of a *design* are described in a timing accurate matter - allowing the user to specify the cycle-accurate protocol a portion of a system uses to communicate with the rest of the *design*.

These two competing requirements pose challenges for verification. The SystemC language has no constructs to distinguish where cycle-accuracy is intended and where a *synthesis tool* has degrees of freedom to re-order operations. The practical implication of this is there is no guarantee the pre-synthesis simulation of a *design* will match the post-synthesis simulation of the resulting RTL. The large body of existing techniques in testbench design and verification methodology which experienced designers use to validate functional correctness is beyond the scope of this document.

In practice, existing *synthesis tools* have adopted tool-specific methodologies for expressing when a segment of code is intended to be treated as cycle-accurate. Standardizing how these blocks of code should be specified is beyond the current scope of this standard, but is a fertile area for future standardization efforts.

A.3 Vision for high-level design

Easier management of system complexity, accelerated design verification and implementation, increased opportunity for design reuse, and a wider selection of implementation options, these are just a few reasons why project teams are moving to high-level design. However, by moving to the higher levels of abstraction, a design gap between HLS 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 ways not previously possible.

Figure A.3 shows the vision for synthesizing from architecture to RTL. Components in the architecture are described in synthesizable SystemC and interfaced and connected using the TLM library.

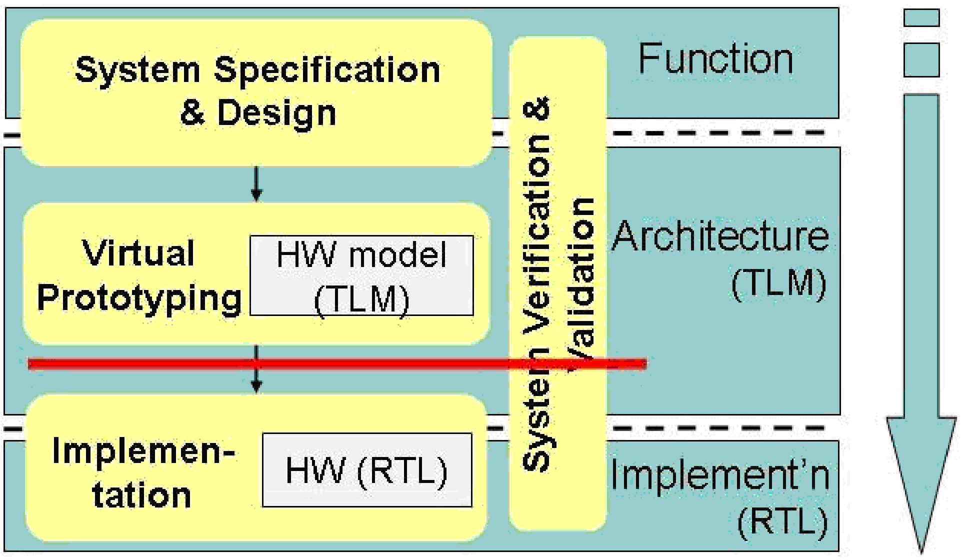


Figure A.3 The vision from architecture (TLM) to implementation (RTL)

Figure A.4 shows a possible implementation flow from Electronic System Level (ESL) down to GDSII. Based on this diagram, a wish list of requirements (beyond the scope of this standard) 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 results 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 the 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 a System Level Design Environment

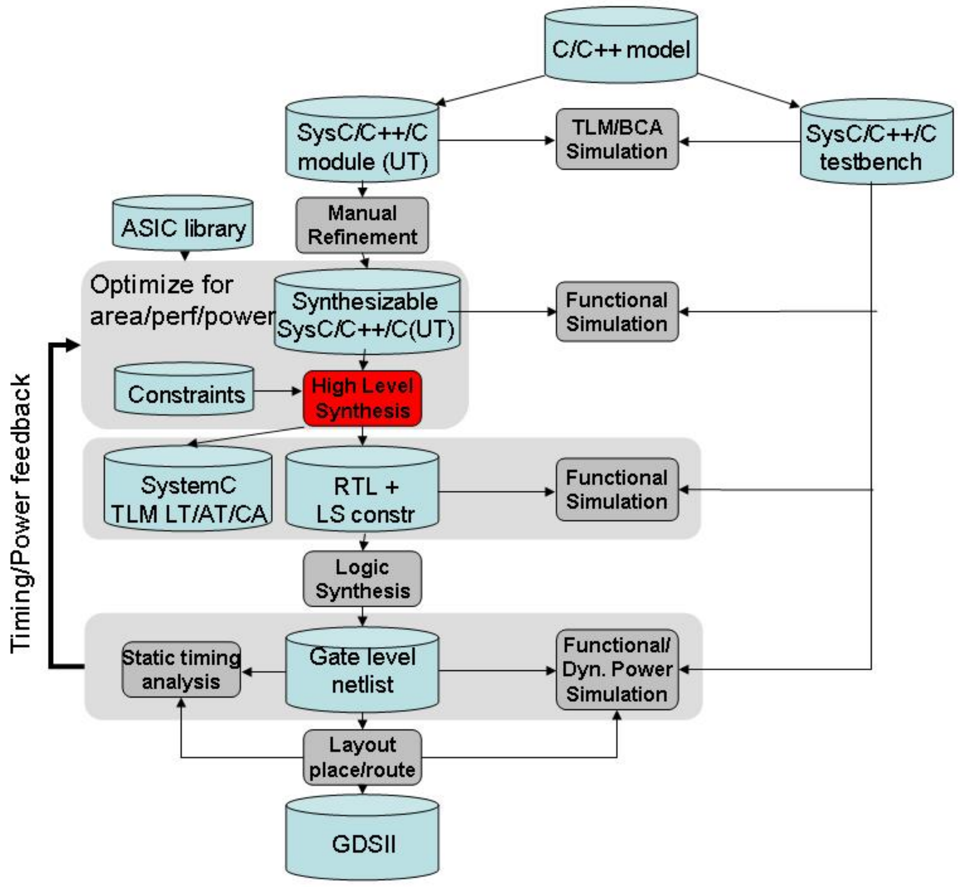


Figure A.4 Abstract view of possible implementation flow from ESL down to GDSII

A.4 Abstraction Level Details

A.4.1 Function Level

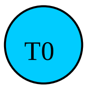
The motivation for introducing this level of abstraction is to quickly obtain a function to determine what the system is supposed to do, without making architecture assumptions. Hence, there is the potential to re-use functions either to create derivative functions or to synthesize different implementations with different architectures.

An example of a view at this level is a function modelled as a process network which can be analyzed through simulation.

At the Function Level of abstraction, two design steps can be identified:

- Algorithm Specification
- Partitioning into communicating tasks

A.4.1.1 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.

A.4.1.2 Function Level: Partitioning into communicating tasks

With the design constraints and requirements, 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 is 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. Such a network is also referred to as a Kahn Process Network (KPN) (Kahn, 1974).

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 [5], YAPI [6] [7] 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.

Function level is also known as behavioral architectural-level [8] [9]]. At this level, for 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. 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.

This can be modeled in SystemC using *SC_METHOD*s, *SC_THREAD*s, or *SC_CTHREAD*s.

Example 1 using SC_METHOD:

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

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

	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 2 using SC_CTHREAD:

```
SC_MODULE( AddMul_3 ) {
	sc_in< bool > 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.read() + (b.read() * c.read());
			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 the 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`.

A.4.2 Architecture Level

The motivation for introducing this level of abstraction into a design flow 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, it is desirable 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) was developed for abstract modeling of (SoC) systems at the 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 have been many discussions regarding TLM over the years; here, the definitions, terminologies, and libraries developed by the OSCI TLM Working Group (TLM WG) are used.

A.4.3 Transaction Level Modeling

Although TLM includes computation and communication, TLM-1 and TLM-2.0 in the SystemC LRM only describes the communication part. In TLM-1 and TLM-2.0, a transaction is a payload, the data structure that passed between modules. The SystemC LRM considers the following coding styles for transaction-level modelling:

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 shall be accomplished using synchronization primitives such as events, mutexes, and blocking FIFOs. Some users adopt the practice of inserting random delays into untimed descriptions 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 the features necessary to boot an operating system and 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 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, 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 explicitly represent the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles.

In this document, UT modules represent an abstraction that aligns well with the synthesizable subset described herein. LT modules use absolute time for timing information and, thus, are not synthesizable (e.g., it is infeasible to 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. The CA modeling style is not addressed in detail in the SystemC LRM.

A.4.4 Implementation Level

This abstraction level captures the details of the interfaces and the I/O functionality, including a full or partial specification/modelling of their timing. The communication among 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 Gate Level are widely used and have traditionally been written in hardware description 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.

A.4.4.1 Implementation Gate Level

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

SystemC is not generally used to represent gate-level constructs.

A.4.4.2 Implementation RT Level

Register Transfer Level (RTL), as the name suggests, describes functions and signals from registers to registers. The basic elements of this level are combinational and sequential functional/logic units, registers, and signals.

An RTL module has a Finite State Machine (FSM) which describes cycle-by-cycle behavior of the target module. The functional behavior of each state can be described inside the FSM. This kind of FSM can be called an FSM with Datapath (FSMD). Or the functional behavior can be described using a separate datapath which is then controlled by the FSM.

The RT level allows the specification of both structural 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.
- An FSM can be specified in such a way that 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.
- An FSM, where states and transitions *may* contain complex logic and arithmetic behavior (not just simple constant assignments to outputs), is constructed as 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 may be 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.

Example:

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

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

      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`.

A.4.4.3 Implementation Behavioral-Level

The behavioral-level introduces some freedom in how operations and I/O are scheduled by only partially constraining the cycle-by-cycle behavior of the I/O. 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 registers.

The specification of behavior 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 behavior is executed next. Instead, the behavior 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 behavior (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 behavior may be changed by synthesis.

Annex B References

[1] "ISO/IEC 14882:2003, Programming Languages - C++," 2003.

[2] IEEE, "IEEE 1666 SystemC Standard Language Reference Manual," 2011.

[3] "ISO/IEC 14882:2011, Programming languages - C++," 2011.

[4] Sun Microsystems Inc, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," in *Numerical Computation Guide*, 2004.

[5] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing, Proceedings of IFIP Congress*, Stockholm, 1974.

[6] e. a. Erwin de Kock, "YAPI: Application Modeling for Signal Processsing Systems," in *Proc. of DAC*, 2000.

[7] e. a. Erwin de Kock, "Proposal for Modeling Kahn Process Networks and Synchronous Dataflow in SystemC," NXP Lab white paper.

[8] G. D. Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill Higher Education, 1994.

[9] D. W. Knapp, Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler, Prentice Hall, 1996.