

Intel Compiler for SystemC User Guide

version 1.3

MIKHAIL MOISEEV, Intel Corporation

Author's address: Mikhail Moiseev, Intel Corporation.

CONTENTS

1	Preface	3
2	Terminology and abbreviations	3
3	Overview	4
3.1	Hardware design with SystemC	4
3.2	Main ICSC features	5
3.2.1	SystemC and C++ support	5
3.2.2	Human-readable generated Verilog	5
3.2.3	Advanced verification features	5
3.2.4	Reusable module library	6
3.2.5	Simple project maintaining	6
3.2.6	Fast design checking and translation	6
4	Installation and run	7
4.1	Prepare to installation	7
4.2	Installation with install.sh script	8
4.3	Manual installation	8
4.3.1	Building and installing Protobuf	8
4.3.2	Building and installing LLVM/Clang	8
4.3.3	Building and installing ICSC	9
4.4	Folders and files after installation	9
4.5	Run tool for examples and tests	10
4.5.1	Generate SV code for one specific example or design	10
4.5.2	Generate SV code for all examples, tests and designs	10
4.5.3	Run SystemC simulation	11
4.6	Run tool for custom design	11
4.7	Tool options and defines	12
5	Preparing SystemC design	13
5.1	Module hierarchy	13
5.2	Module interconnect	13
5.3	Module constructor code	16
5.4	Method process	17
5.5	Clocked thread process	18
5.6	Clock and reset	20
5.7	Data types	22
5.8	Pointers and references	23
5.9	Array and SC vector types	24
5.10	Record and union types	26
5.11	Type cast	26
5.12	Dynamic memory allocation	27

5.13	Control flow operators	28
5.13.1	if	28
5.13.2	switch	29
5.13.3	for	30
5.13.4	while	30
5.13.5	do...while	31
5.13.6	break	31
5.13.7	continue	31
5.13.8	goto	31
5.14	Function calls	31
5.15	Naming restrictions	33
6	SystemVerilog generation rules	34
6.1	Module generation	34
6.2	Variables generation	36
6.3	Constants generation	38
6.4	Method process generation	39
6.5	Method process with empty sensitivity	40
6.6	Method process with latch(es)	40
6.7	Thread process generation	41
6.8	Thread process without reset	43
6.9	wait(N) conversion	44
6.10	Loops with wait generation	45
6.11	Switch generation	47
6.12	Records generation	49
6.13	Arithmetical operations	50
6.13.1	Signed and unsigned literals	50
6.13.2	Signed and unsigned types	50
6.13.3	Type cast in assignment	51
6.13.4	Operator comma	52
6.14	Name uniqueness	52
7	Extensions	54
7.1	Advanced FIFO	54
7.2	SystemVerilog intrinsic insertion	55
7.3	Memory module name	56
8	Advanced verification features	57
8.1	Immediate assertions	57
8.2	Temporal assertions	58
8.2.1	Temporal assertions in module scope	59
8.2.2	Temporal assertions in clocked thread process	60
8.2.3	Temporal assertions in loop inside of clocked thread	62

4 • Mikhail Moiseev

8.3	Special assertions	63
9	Error messages	64
9.1	Design correctness checking	64
9.2	Error messages	64

1 PREFACE

Intel®Compiler for SystemC* (ICSC) is open source tool distributed under [Apache License v2.0 with LLVM Exceptions](#). The source codes are available at github.com/intel/systemc-compiler.

This ICSC User Guide document intended to help user to install and run the tool, prepare SystemC design to be translated into SystemVerilog and understand the result SystemVerilog code. The User Guide document also describes some common cases in hardware design, ICSC extensions and advanced verification features. There are two main documents referenced in this User Guide:

- SystemC LRM – IEEE Standard for Standard SystemC Language Reference Manual, IEEE Std 1666 2011,
- SystemC Synthesizable Subset – SystemC Synthesizable Subset Version 1.4.7.

2 TERMINOLOGY AND ABBREVIATIONS

In this document the following terminology is used:

- Module - SystemC module, a C++ class or structure inherits `sc_module`;
- Modular interface - SystemC module which inherits `sc_interface`;
- Record - C++ class or structure which is not module nor modular interface;
- Signal - SystemC `sc_signal`;
- Port - SystemC `sc_in` and `sc_out`;
- Channel - signal or port;
- Port interface - SystemC `sc_port<IF>`;
- SC vector - SystemC `sc_vector<T>`, container with objects of type T.

In this document the following abbreviations are used:

- ICSC - Intel®Compiler for SystemC*;
- SC - SystemC language;
- SV - SystemVerilog language;
- SVA - SystemVerilog assertions;

3 OVERVIEW

Intel®Compiler for SystemC* (ICSC) is a tool which translates hardware design in SystemC language into equivalent design in synthesizable SystemVerilog.

ICSC supports SystemC synthesizable subset standard IEEE Std 1666. In addition it supports arbitrary C++ code in module constructors and everywhere at elaboration phase. Synthesizable SystemVerilog means a subset of SystemVerilog language IEEE Std 1800 which meets requirements of the most logic synthesis and simulation tools.

ICSC is focused on improving productivity of design and verification engineers. That provided with SystemC language high level abstractions and extensions implemented in ICSC itself. The rest of this section devoted to general SystemC design approaches and main features of ICSC.

ICSC works under Linux OS. There is a script to automatically build and install the tool at Ubuntu 20.04. ICSC has minimalistic set of options which have appropriate default values. To run ICSC it needs to point input SystemC files and top modules name. Installation procedure and run ICSC tool described in Section 4.

SystemC to SystemVerilog translation with ICSC has multiples stages, including input design checks to detect non-synthesizable code, common coding mistakes and typos. The general rules are specified by SystemC synthesizable subset standard. Some good practices and common cases in synthesizable SystemC designs are discussed in Section 5.

ICSC produces human-readable SystemVerilog code which looks very similar to the input SystemC design. SystemC thread and method processes are translated into `always_comb` and `always_ff` blocks. Control flow of generated always blocks is similar to control flow of the SystemC processes. There are almost the same variable declarations, control statements (if, switch, loops) and code blocks. More details of generated SystemVerilog code structure are given in Section 6.

ICSC has multiple extensions including Advanced FIFO collection, support of SystemVerilog intrinsics and vendor memory as it is described in Section 7. Advanced verification features based on immediate and temporal SystemC assertions are discussed in Section 8.

Input SystemC design has to be passed through C++ compiler and comply to SystemC synthesizable standard rules. To simplify error detection and understanding, ICSC provides precise and meaningful error reporting which is given in Section 9.

3.1 Hardware design with SystemC

SystemC language and design methodology intended to use single language for architecture exploration, executable specification creating, hardware/software partitioning, hardware modules implementation, verification environment setup and testbench development.

Hardware design with SystemC has several advantages over conventional SystemVerilog/VHDL flow:

- Efficient FSM design in clocked threads, with implicit states inferred from `wait()` calls;
- Full power of C++ language including

- Object-oriented programming (inheritance, polymorphism),
- Template-based meta-programming,
- Abstract types and specific data types.
- More abilities to write reusable code with parameterization based on templates, class constructor parameters, and other design patterns;
- Huge number of open source C++ libraries which can be used for hardware design verification.

3.2 Main ICSC features

ICSC is intended to improve productivity of design and verification engineers.

ICSC main advantages over existing HLS tools:

- Better SystemC and C++ support;
- Human-readable generated Verilog, closely matching SystemC sources;
- Advanced verification features with automatic SVA generation;
- Library with memory, channels and other reusable modules;
- Simpler project maintaining and verification environment setup;
- Fast design checking and code translation.

3.2.1 *SystemC and C++ support.* ICSC supports modern C++ standards (C++11, C++14, C++17). That allows in-class initialization, for-each loops, lambdas, and other features. As ICSC uses dynamic code elaboration, there is no limitations on elaboration stage programming. That means full C++ is supported in module constructors and everywhere executed at elaboration phase.

ICSC operates with the latest SystemC version 2.3.3, so it includes all modern SystemC features.

3.2.2 *Human-readable generated Verilog.* ICSC generates SystemVerilog code which looks like SystemC input code. That is possible as ICSC does no optimizations, leaving them to logic synthesis tools next in the flow.

Human-readable generated code gives productivity advantages over the HLS tools:

- DRC and CDC bugs in generated Verilog can be quickly identified in input SystemC;
- Timing violation paths can be easily mapped to input SystemC;
- ECO fixes have little impact on generated SystemVerilog.

3.2.3 *Advanced verification features.* SystemC includes `sc_assert` macro for checking assertion expression during simulation. ICSC provides automatic translation `sc_assert` to equivalent SystemVerilog Assertion (SVA).

In addition to `sc_assert` SVA provides assertions with temporal conditions which looks similar to SVA. These assertions implemented in the ICSC library and have the same semantic

in SystemC and SystemVerilog simulation. More details about verification support described in Section 8.

3.2.4 Reusable module library. ICSC module library includes the following:

- Advanced FIFO for modules and processes interconnect with zero size option;
- Zero delay channel which is base for memory modules with functional interface accessible from all process types;
- Clock gate, clock synchronizer and other standard modules

Detailed description of the library is given in Section 7.

3.2.5 Simple project maintaining. ICSC supports CMake build system and provides simple run with input files specified in CMake file. ICSC does not use any project scripts, tool-specific macros or pragmas.

ICSC supports SystemVerilog intrinsic module insertion into SystemC design. The SystemVerilog code can be written or included into SystemC module. More details of intrinsic support given in Section 7.2.

3.2.6 Fast design checking and translation. ICSC does design check and error reporting in a few second for average size designs (100K of unique SystemVerilog lines-of-code). SystemVerilog code generation of such complexity designs typically takes a few minutes.

4 INSTALLATION AND RUN

This section explains building and installation of ICSC on any Linux OS. In the following description Ubuntu 20.04 is used, all operations are given for bash terminal.

ICSC is based on Clang/LLVM tool chain. It uses Google Protobuf library and patched SystemC library. Patched SystemC sources are uploaded into ICSC repository.

There are two ways to install ICSC:

- Installation with install.sh script
- Manual installation

4.1 Prepare to installation

ICSC can be installed on Linux OS with:

- C++ compiler supports C++17 (for gcc it is version 8.0.0 or later)
- CMake version 3.12 or later
- git to clone ICSC repository

Initial step before installation is to setup some folder as \$ICSC_HOME and clone ICSC source repository to \$ICSC_HOME/icsc:

```
$ export ICSC_HOME=/home/user/my_iscs_folder
$ git clone https://github.com/intel/systemc-compiler $ICSC_HOME/icsc
```

After clone before installation there is the following folder structure:

```
$ICSC_HOME
* icsc
  * cmake          -- CMake files
  * components     -- assertions, fifo and other library components
  * designs        -- folder for user designs with an design template
  * doc            -- user guide latex and pdf files
  * examples       -- a few illustrative examples
  * sc_elab        -- elaborator sources
  * sc_tool        -- ICSC sources
  * systemc        -- patched SystemC 2.3.3 sources
  * tests          -- unit tests
  * .gitmodules    -- not intended to be used here, can be removed
  * CMakeLists.txt -- Cmake file for ICSC tool
  * LICENSE.txt    -- Apache 2.0 WITH LLVM exceptions license
  * README.md      -- Tool description
  * install.sh     -- Installation script
```

4.2 Installation with install.sh script

The `install.sh` script contains all the stages of manual installation, that includes generating SystemVerilog code for examples.

Open bash terminal and run `icsc/install.sh` from `$ICSC_HOME` folder:

```
$ cd $ICSC_HOME
$ icsc/install.sh          # download and install all required components
$ cd $ICSC_HOME
$ source setenv.sh        # setup PATH and LD_LIBRARY_PATH
```

Before using the installed tool in a new terminal it needs to run `setenv.sh`:

```
$ export ICSC_HOME=/home/user/my_icsc_folder
$ cd $ICSC_HOME
$ source setenv.sh        # setup PATH and LD_LIBRARY_PATH
```

4.3 Manual installation

4.3.1 Building and installing Protobuf. Download Protobuf version 3.6.1 or later from <https://github.com/protocolbuffers/protobuf/releases> into `$ICSC_HOME` folder.

```
$ cd $ICSC_HOME/protobuf-3.13.0
$ mkdir build && cd build
$ cmake ../cmake/ -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$ICSC_HOME
  -DBUILD_SHARED_LIBS=ON
$ make -j8
$ make install
```

4.3.2 Building and installing LLVM/Clang. Download LLVM 7.0.0 and Clang 7.0.0 from <https://releases.llvm.org/download.html#7.0.0> into `$ICSC_HOME` folder.

```
$ mv $ICSC_HOME/cfe-7.0.0.src $ICSC_HOME/llvm-7.0.0.src/tools/clang
$ cd $ICSC_HOME/llvm-7.0.0.src
$ mkdir build && cd build
$ cmake ../ -DLLVM_ENABLE_ASSERTIONS=ON -DCMAKE_BUILD_TYPE=Release
  -DCMAKE_INSTALL_PREFIX=$ICSC_HOME
$ make -j8
$ make install
```

4.3.3 Building and installing ICSC.

```
$ cd $ICSC_HOME/icsc
$ mkdir build && cd build
$ cmake ../ -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$ICSC_HOME
$ make -j8
$ make install
```

4.4 Folders and files after installation

After installation there is the following folder structure:

```
$ICSC_HOME
* bin          -- binary utilities, can be removed
build          -- build folder, can be removed
* icsc         -- ICSC sources
  * build      -- build folder, can be removed
  * cmake      -- CMake files
  * components -- assertions, fifo and other library components
  * designs    -- folder for user designs with an design template
  * doc        -- user guide latex and pdf files
  * examples   -- a few illustrative examples
  * sc_elab    -- elaborator sources
  * sc_tool    -- ISCS sources
  * systemc    -- patched SystemC 2.3.3 sources
  * tests      -- unit tests
  * .gitmodules -- not intended to be used here, can be removed
  * CMakeLists.txt -- Cmake file for ICSC tool
  * LICENSE.txt  -- Apache 2.0 WITH LLVM exceptions license
  * README.md    -- Tool description
  * install.sh   -- Installation script
* include      -- LLVM/Clang, SystemC and other headers
* lib          -- tool compiled libraries
* libexec      -- can be removed
* lib64        -- tool compiled libraries
* share        -- can be removed
* CMakeLists.txt -- CMake file for examples, tests and user designs
* README       -- build and run examples, tests and used designs description
* setenv.sh    -- set environment script for bash terminal
```

4.5 Run tool for examples and tests

There are number of examples in examples sub-folder:

- asserts – immediate and temporal SystemC assertions with SVA generation
- counter – simple counter with SC_METHOD and SC_CTHREAD processes
- decoder – configurable FIFO example
- dvcon20 – assertion performance evaluation examples
- fsm – finite state machine coding
- intrinsic – Verilog code intrinsic example
- int_error – error reporting example, dangling pointer de-reference inside
- latch_ff – simple latch and flip flop with asynchronous reset

There are number of unit tests in tests sub-folder:

- const_prop – constant propagation analysis tests
- cthread – general tests in SC_CTHREAD
- elab_only – dynamic elaborator tests
- method – general tests in SC_METHOD
- mif – modular interface tests
- misc – extra tests
- record – local and members of struct and class type
- state – state tests
- uniquify – module uniquification tests

4.5.1 Generate SV code for one specific example or design.

```
$ cd $ICSC_HOME
$ source setenv.sh           # setup PATH and LD_LIBRARY_PATH
$ cd build
$ cmake ../                  # prepare Makefiles
$ ctest -R counter           # run SV generation for counter example
```

4.5.2 Generate SV code for all examples, tests and designs.

```
$ cd $ICSC_HOME
$ source setenv.sh           # setup PATH and LD_LIBRARY_PATH
$ cd build
$ cmake ../                  # prepare Makefiles
$ ctest -j8                  # compile and run SV generation
```

Generated SystemVerilog files are put into sv_out folders. For counter example:

```
$ cd examples/counter        # go to counter example folder
$ cat sv_out/counter.sv      # see generated SystemVerilog file
```

4.5.3 *Run SystemC simulation.* SystemC simulation for examples and tests can be run with:

```
$ cd $ICSC_HOME
$ mkdir build && cd build
$ cmake ../                # prepare Makefiles
$ make counter             # compile SystemC simulation for counter example
$ cd examples/counter      # go to counter example folder
$ ./counter                # run SystemC simulation
```

4.6 Run tool for custom design

To run ICSC for custom design it needs to create a CMakeList.txt file for the project. SystemVerilog generation is performed with `svc_target` function call. `svc_target` is CMake function defined in `$ICSC_HOME/lib64/cmake/SVC/svc_target.cmake`.

The custom design can be placed into `$ICSC_HOME/icsc/designs` folder. There is an empty design template `$ICSC_HOME/icsc/designs/template`. This design template contains `example.cpp` and `dut.h` files. In the design template top module is specified as variable name `dut_inst` which is instantiated in module `tb`, so full SystemC name `tb.dut_inst` is provided.

Listing 1. CMakeList.txt file for design template

```
# Design template CMakeList.txt file
project(mydesign)

# All synthesizable source files must be listed here (not in libraries)
add_executable(mydesign example.cpp)

# Test source directory
target_include_directories(mydesign PUBLIC $ENV{ICSC_HOME}/examples/template)

# Add compilation options
# target_compile_definitions(mydesign PUBLIC -DMYOPTION)
# target_compile_options(mydesign PUBLIC -Wall)

# Add optional library, do not add SystemC library (it added by svc_target)
#target_link_libraries(mydesign sometestbenchlibrary)

# svc_target will create @mydesign_sctool executable that runs code generation
# and @mydesign that runs general SystemC simulation
# ELAB_TOP parameter accepts hierarchical name of DUT
# (that is SystemC name, returned by sc_object::name() method)
svc_target(mydesign ELAB_TOP tb.dut_inst)
```

4.7 Tool options and defines

To run ICSC tool for the custom project it needs to create CMakeList.txt file. SystemVerilog code generation is done with `svc_target` function. `svc_target` is CMake function defined in `$ICSC_HOME/lib64/cmake/SVC/svc_target.cmake`.

ICSC has several options, which can be specified as `svc_target` parameters:

- `ELAB_TOP` – design top module name, it needs to be specified if top module is instantiated outside of `sc_main()` or if there are more than one modules in `sc_main()`;
- `MODULE_PREFIX` – module prefix string, no prefix if not specified, prefix applied for every module excluding Verilog intrinsic (module with `__SC_TOOL_VERILOG_MOD__`) and memory modules (modules with `__SC_TOOL_MEMORY_NAME__`);
- `REPLACE_CONST_VALUE` – replace constant with its evaluated value if possible, by default constant variable is used;
- `INIT_LOCAL_VARS` – initialize non-initialized process local variables with zero to avoid latches, that related to CPP data types only, SC data types always initialized with 0;
- `NO_SVA_GENERATE` – do not generate SVA from immediate and temporal SystemC assertions, normally SVA are generated;
- `NO_REMOVE_EXTRA_CODE` – do not remove unused variable and unused code in generated SV, normally such code is removed to improve readability.

ICSC tool provides `__SC_TOOL__` define for input SystemC project translation. This define used in temporal assertions and other ICSC library modules to have different behavior for simulation and SV generation. `__SC_TOOL__` can also be used in project code to hide pieces of code which is not targeted for translation to SystemVerilog.

To completely disable SystemC temporal assertion macro `SCT_ASSERT_OFF` can be defined. That allows to hide all assertion specific code to meet SystemC synthesizable standard requirements. `SCT_ASSERT_OFF` is required if the SystemC design is passed through a tool which includes its own (not patched) SystemC library.

5 PREPARING SYSTEMC DESIGN

5.1 Module hierarchy

SystemC design consists of module and modular interface instances which are organized into a module hierarchy. *Module* is a C++ class or structure which inherits `sc_module` class. *Modular interface* is a C++ class or structure which inherits `sc_interface` and `sc_module` classes. If a class inherits `sc_interface`, but does not inherit `sc_module` we call it *pure interface*. Pure interface is not a part of module hierarchy. Each interface and module, except top module, is instantiated inside of *parent module*. Modules and modular interfaces can be instantiated at stack or be dynamically allocated in heap with operator `new`.

The difference between module and modular interface from ICSC viewpoint is that modular interface is flattened into parent module where it is instantiated. That means there is no modular interface instance in generated SystemVerilog, but its fields and processes are instantiated into the parent module. In other words, modular interface is a kind of module which needs to be flattened in its parent module.

The SystemC design may have one or several top module instances in `sc_main` function or another module, but only one of them will be translated into SystemVerilog. The top module to take by ICSC is specified in `ELAB_TOP` option of `svc_target`. If top module instantiated in `sc_main` function and there is no more modules instantiated, `ELAB_TOP` option could be omitted.

Top module contains child module(s). Every module may inherit another module(s), interface(s) or class(es) according with C++ rules. Multiple inheritance and virtual inheritance is supported by ICSC tool.

Module, interface, class or structure can be template type. Template types, template specialization and instantiation specified by C++ rules. All that is supported by ICSC tool.

In accordance with SystemC LRM, modules/modular interfaces and pointers to them cannot be function parameters, cannot be returned from function, and cannot be used as signal/port template type.

5.2 Module interconnect

For inter-module communication signals, ports and port interfaces (`sc_port<IF>`) can be used. Having explicit pointers/references to another module fields or methods considered as bad programming style and should be avoided.

Child module input/output ports could be directly connected to corresponded input/output ports of its parent module. Child ports, not connected to any signal/port, are promoted to its parent module and further up to top module. That practically means, unconnected port becomes the same type port of top module in generated SystemVerilog.

```
// SystemC child module port promotion to top module
```

```

SC_MODULE(Child) {
    sc_in_clk  clk{"clk"};
    sc_in<sc_int<16>> in{"in"};    // Not connected
    sc_out<sc_int<16>> out{"out"}; // Not connected

    SC_CTOR(Child) {}
};

SC_MODULE(Top) {
    sc_in_clk  clk{"clk"};
    Child child_inst{"child_inst"};

    SC_CTOR(Top) {
        child_inst.clk(clk);
    }
};

```

```

// SystemVerilog generated
module Top // "tb_inst.top_mod"
(
    input logic clk,
    input logic signed [15:0] child_instin, // Port promoted
    output logic signed [15:0] child_instout // Port promoted
);
...
endmodule

```

Two modules with the same parent module can be connected through:

- (1) triple of `sc_in<T>`, `sc_signal<T>`, `sc_out<T>`
- (2) pair of `sc_in<T>`, `sc_signal<T>`
- (3) pair of `sc_out<T>`, `sc_signal<T>`

In case (1) `sc_in<T>` port is in module A, `sc_out<T>` port in module B and `sc_signal<T>` in their parent module. These input and output ports bound to the signal in the parent module constructor. In case (2) `sc_in<T>` port is in module A, `sc_signal<T>` in module B. The input port connected to the signal in the parent module constructor. In case (3) `sc_signal<T>` is in module A, `sc_out<T>` port in module B. The output port connected to the signal in the parent module constructor.

```

// SystemC child modules connected to each other
SC_MODULE(Producer) {

```



```

    sc_signal<bool>      req_sig{"req_sig"};
    sc_out<bool>         resp{"resp"};
    sc_out<sc_int<16>>   data{"data"};

    SC_CTOR(Producer) {}
};

SC_MODULE(Consumer) {
    sc_in<bool>          req{"req"};
    sc_signal<bool>      resp_sig{"resp_sig"};
    sc_in<sc_int<16>>    data{"data"};

    SC_CTOR(Consumer) {}
};

SC_MODULE(Parent) {
    Producer prod{"prod"};
    Consumer cons{"cons"};

    sc_signal<sc_int<16>> data{"data"};

    SC_CTOR(Parent) {
        cons.req(prod.req_sig); // in-to-signal (2)
        prod.resp(cons.resp_sig); // out-to-signal (3)
        prod.data(data);          // in-to-signal-to-out (1)
        cons.data(data);
    }
};

```

```

// SystemVerilog generated
module Parent // "tb_inst.top_mod"
(
);
// SystemC signals
logic signed [15:0] data;
logic resp;
logic req;

Producer prod
(
    .resp(resp),
    .data(data),

```

```

    .req_sig(req)
);
Consumer cons
(
    .resp_sig(resp),
    .req(req),
    .data(data)
);
...
endmodule

```

A module process can access its child modular interface instance fields and call methods. That is possible as interface fields and methods are moved to the parent module in generated SystemVerilog. Direct accessing fields of another module considered as bad programming style and should be avoided.

In general case a module can have pointer to another module or `sc_port<IF>` connected to another module. It can access field and call methods of the pointee module if both of these modules are flatten in the same module in SystemVerilog. That can be parent and its child instance of modular interface or two child instances of modular interface(s) in the same parent module.

`sc_port<IF>` is special case of port which provides pure interface IF. `sc_port<IF>` can be connected to a modular interface, which implements all abstract methods of IF. With `sc_port<IF>` it is possible to call methods of the IF that limits access to connected module. In that access via `sc_port<IF>` differs from access via modular interface pointer.

Other ways of call of another module methods or access another module fields are prohibited.

5.3 Module constructor code

Any module/modular interface must have at least one constructor. Module/modular interface constructor must have `sc_module_name` parameter which can be passed to `sc_module` inheritor or just not used. Module/modular interface constructor can contains arbitrary C++ code.

Constant fields can be initialized in module constructor initialization list and body, they are generated as `localparam` in SV. Constant of any integral type initialized at elaboration phase should be 64 bit or less (stored in `int64` field). Constant more than 64 bit has unknown value after elaboration.

```

// Constant fields initialized in constructor code
class MyModule : public sc_module {
    sc_uint<8> a = 0;
    const bool b = false;

```

```

int c[4] = {0,0,0,0};

MyModule(const sc_module_name& name, sc_uint<8> aa, int bb) :
    sc_module(name)
{
    const_cast<sc_uint<8>&>(a) = (bb) ? aa - 1 : aa + 1;
    const_cast<bool&>(b) = bb;
    for (int i = 0; i < N; i++) {
        const_cast<int&>(c[i]) = i;
    }
}
}
MyModule mod("mod", 12, true);

```

Non-constant module fields cannot be initialized in constructor or in place. After elaboration phase non-constant fields have unknown value. They need to be initialized in reset section of a process.

5.4 Method process

Method process created with SC_METHOD is used to describe combinational logic, so it is *combinational method* in terms of SystemC synthesizable standard.

Method process can be sensitive to one or more signal change event or be no sensitive to any. The method sensitivity list should be static and should include all signals/ports that are read in the method function to avoid unintentional latches. ICSC tool supports latches as described in Section 6.6.

```

// SystemC method process example
SC_MODULE(MyModule) {
    sc_in<bool>    in{"in"};
    sc_signal<int> sig{"sig"};
    sc_out<bool>  out{"out"};

    SC_CTOR(MyModule) {
        SC_METHOD(methodProc);
        sensitive << in << sig;
    }
    void methodProc() {
        bool b = in;        // Use in, it need to be in sensitive list
        if (sig != 0) {     // Use sig, it need to be in sensitive list
            out = b;
        } else {

```

```

        out = 0;
    }
}
};

```

For method process ICSC generates always_comb block as described in 6.4.

Method process with empty sensitivity are typically used to assign constant value signal/port initialization. In the SystemVerilog code one or more assign statements are generated for such process, see 6.5.

In such method it is possible to use local variables to store intermediate results. Module variable initialization can be done in such process, but not recommended as can lead to concurrent assignment to the variable. Ternary operator with arbitrary condition is supported here. if statement with statically evaluated condition is also supported. Loops and other control flow statements cannot be used here.

Read port/signal in such method is supported, but not recommended as can lead to different behavior in Verilog vs SystemC (in SystemC process not activated if signal is changed). In the Verilog one or more assign statements are generated for method without sensitivity.

```

// SystemC method process with empty sensitivity
static const bool COND = true;
void emptySens()
{
    a = 0;
    if (COND) {
        b = 1;
    } else {
        c = 2;
    }
    int i = 1;
    d = (!COND) ? i : i + 1;
}

```

5.5 Clocked thread process

Clocked thread process created with SC_THREAD and SC_CTHREAD macros are supported. Clocked thread is activated by one edge of clock signal which is specified in constructor: for SC_CTHREAD as second macro parameter, for SC_THREAD in sensitivity list.

```

// Clock thread process with clock and reset
class MyModule : public sc_module
{

```

```

sc_clk_in    clk{"clk"};
sc_in<bool>  rst{"rst"};

    CTOR(MyModule) {
        SC_CTHREAD(threadProc, clk.pos());
        async_reset_signal_is(rst, false);

        SC_THREAD(threadProc);
        sensitive << clk.pos();
        async_reset_signal_is(rst, false);
    }
}

```

Considering SystemC synthesizable subset restrictions on SC_THREAD process, there is no difference between SC_THREAD and SC_CTHREAD processes.

Clocked thread normally has reset section and infinite loop called as *main loop*. Reset section contains local variable declaration, local and module variables initialization, signal and port initialization. In reset section read of any signal/port is prohibited.

The clocked thread main loop can use any loop statements while, for, do...while with *true* loop condition. Main loop can contain multiple wait() calls directly or in called functions. The main loop must contain at least one wait() call at each path through the loop body.

```

// Clock thread with wait() in reset section and main loop
void threadProc() {
    // Reset behavior
    // ...
    wait();
    while (true) {      // Main loop
        // Operational behavior
        ...
        wait();
    }
}

```

```

// Clock thread with one common wait() for reset and main loop
void threadProc() {
    // Reset behavior
    // ...
    while (true) {      // Main loop
        // Reset and operational behavior
        wait();
    }
}

```

```

        // Operational behavior
        ...
    }
}

```

Any other loops in clocked thread may contain or not contain `wait()` calls. If a loop contain a `wait()` call, it must contain at least one `wait()` call at each path through the loop body – the same requirements as for main loop. The same is true for `wait(N)` calls.

There is a simple example of clocked thread. For thread process ICSC generates pair of `always_comb` and `always_ff` blocks as described in 6.7.

```

// SystemC simple thread example
sc_in<unsigned>  a{"a"};
sc_out<unsigned> b{"b"};

CTOR(MyModule) {
    SC_CTHREAD(test_ctypead, clk.pos());
    async_reset_signal_is(rst, false);
}

void test_ctypead () {
    unsigned i = 0;
    b = 0;
    while (true) {
        wait();
        b = i;
        i = i + a;
    }
}

```

5.6 Clock and reset

Clocked thread process (`SC_CTHREAD` and `SC_THREAD`) is activated by one clock edge. Method process (`SC_METHOD`) can be sensitive to any change in the signals including clock positive or/and negative edge.

Clocked thread process can have one, several or no reset. Clocked thread cannot have more than one asynchronous reset, but can have multiple synchronous resets. SystemC synthesizable subset does not allow clocked thread without resets and multiple synchronous resets. As soon as no reset clocked threads and clocked thread with multiple synchronous resets are required for some applications that is supported by ICSC tool.

Method process can be sensitive to all resets required.

```
// Clocked thread with multiple resets
SC_CTOR(test_reset) {
    SC_CTHREAD(proc, clk.pos());
    reset_signal_is(sreset, true);
    async_reset_signal_is(areset, true);
    async_reset_signal_is(areset2, false);
}

void proc() {
    enable = 0;
    wait();
    while (true) {
        wait();
    }
}
}
```

```
// SystemVerilog generated for clocked thread with multiple resets
always_ff @(posedge clk or posedge areset or negedge areset2 /*sync sreset*/)
begin : proc
    if (areset || ~areset2 || sreset) begin
        enable <= 0;
    end
    else begin
        ...
    end
end
end
```

Clocked process without reset supported with limitations: such process can have only one wait() and cannot have any code in reset section.

```
// Clocked thread without reset
SC_CTOR(test_reset) {
    SC_CTHREAD(proc, clk.pos());
}

void proc() {
    while (true) {
        int i = 0;
    }
}
```

SC/C++ type	SV type	SC synthesizable subset
sc_uint<N>	logic [N]	
sc_biguint<N>	logic [N]	
sc_int<N>	logic signed [N]	
sc_bigint<N>	logic signed [N]	
bool	logic	
char, signed char	logic signed [8]	
unsigned char	logic [8]	
short	logic signed [16]	
unsigned short	logic [16]	
int	integer	32bit
unsigned int	integer unsigned	32bit
long	logic signed [64]	32/64bit depends on platform
long long	logic signed [64]	64bit
unsigned long	logic [64]	32/64bit depends on platform
unsigned long long	logic [64]	64bit
__uint128_t	logic [128]	
__int128_t	logic signed [128]	

Table 1. Data type conversion

```

        wait();
    }
}

```

For thread process ICSC generates pair of `always_comb` and `always_ff` blocks as described in 6.8.

5.7 Data types

This section describes types can be used in process functions. SystemC integer types `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint` are supported. C++ `bool`, `char`, `short`, `integer`, `long integer` and `long long integer` and their unsigned versions are supported. Generated SystemVerilog data types shown in Table 1.

Uninitialized local variable of SystemC types (`sc_uint`, `sc_biguint`, `sc_int`, `sc_bigint`) has got zero value in generated code as it got this value in the default constructor. That means declaration of such variable leads to its initialization by 0.

Not supported SystemC data types: `sc_bv`, `sc_lv`, `sc_logic`, `sc_signed`, `sc_unsigned`, `sc_fix`, `sc_ufix`, `sc_fixed`, `sc_ufixed`. Not supported floating point C++ data types: `float`, `double`.

C++ types can be used together with SC types. Signed and unsigned types should never be mixed, as that can lead to unexpected result for operations with negative values. For operations with mix of signed and unsigned arguments of `sc_int/sc_uint` types SystemC simulation can differ from generated SystemVerilog simulation. See more details in Section 6.13.

5.8 Pointers and references

This section describes operations with pointers and references in process functions. Pointers can be declared as module members and as local variables in functions. Member pointers are normally initialized at elaboration phase and used (de-referenced) in process functions. Local pointers are initialized and used in functions. Local pointers are limited to non-port/non-channel object types.

```
// Pointer initialization example
sc_signal<int>    s{"s"};
sc_signal<int>*  sp;
sc_signal<int>*  sd;
int    i;
int*   q;
int*   d;

SC_CTOR(MyModule){
    sp = &s;
    sd = new sc_signal<int>("sp");
    q  = &i;
    d  = sc_new<int>();
}
void someProc() {
    int* p = d; // Local pointer assignment in initialization
    *p = 1;
    p = q;      // ERROR, no local pointer assignment
}
```

In module constructors there is no limitations on pointer/references and new/delete usages. In process functions pointer de-reference (*) operation is supported. Pointer reference (&) is not supported there. Operator new/new[] and operator delete/delete[] is not supported. Pointer assignment supported at declaration only, general pointer assignment not supported. Pointer comparison supported, other pointer arithmetic not supported. Pointer can be assigned to boolean variable, as well as, used in comparison and condition. Pointer null value (nullptr) considered as false, object values as true. Pointer function parameter supported except pointer to record which is not supported. Return value from function by pointer not supported.

```
// Pointer arithmetic example
T a;
T* p = nullptr;
T* q = &a;

void someProc() {
    bool b = !p;          // Result is true
    if (p || p == q) {    // Result is false
        ...
    }
    b = p && (*p == 1); // De-reference pointer if its not null
}
```

Reference/constant reference type members and local variables supported. Constant reference can be initialized with variable, literal or constant expression. Reference function parameter supported. Return value from function by reference not supported.

```
// Reference example
template <class T>
T const_ref(const T& val) {
    T j = val+1;
    return j;
}

void refProc() {
    int a;
    int &b = a;          // Local reference
    b = 1;
    int i = const_ref(a); // Parameter passed as reference
    i = const_ref(1);
}
```

5.9 Array and SC vector types

Arrays are supported as module members and function local variables. One-dimensional and multidimensional arrays are supported. Array of modules/modular interfaces and modules/modular interfaces pointers are supported. In array of module/modular interface base class pointers all elements must be the same class. Array of signals/ports and signal/port pointers are supported. Array of port interfaces (sc_port<IF>) not supported. Array of records and

record pointers are supported. In array of base class pointers all elements must be the same class.

```
static const unsigned N = 5;
static const unsigned M = 10;
sc_uint<16>          a[N];
sc_in<bool>          in[N];
sc_out<sc_uint<4>>   out[N][M];
sc_signal<int>*      sig[N];

SC_CTOR(myModule) {
    for (int i = 0; i < N; i++) {
        char sname[32];
        sprintf(sname, "sig_%d", i);
        sig[i] = new sc_signal<int>(sname);
    }
}
```

Array of any pointers must be homogeneous, all elements created with operator new, but not pointers to existing objects. Array and array of pointers, including array of signals/ports, can be a function parameter.

Signal/port array index cannot be the same signal/port array. Array of record cannot be accessed at index which is the same array element.

```
sc_in<T> a[N], b[M];
...
a[a[i]] = 0;    // Not supported
a[b[i]] = 0;    // Supported
a[a[i].m].m = 0; // Not supported
a[b[i].n].m = 0; // Supported
```

SC vector (sc_vector) supported for signals/ports and modules. SC vector is not supported for modular interface yet. Two-dimensional vector (vector of vectors) also supported. SC vector instances can be instantiated in modules and modular interfaces, including array of modular interfaces. SC vector cannot be passed to or returned from function.

```
sc_vector<sc_in<bool>> req{"req"};
sc_vector<sc_out<bool>> res{"resp", 3};
sc_vector<sc_vector<sc_signal<int>>> sig2d{"sig2d", 2};
SC_CTOR(MyModule) {
    req.init(3);
    sig2d[0].init(3);
    sig2d[1].init(3);
}
```

```
...
}
```

5.10 Record and union types

This section describes usage of record which are C++ structures or classes which are not modules and modular interfaces. Records intended to represent set of plain data fields. Record can be module member as well as local variables in process functions.

Records are supported with limitations. Record can have member functions and data type members. Record cannot have another record members, signal/port or module/modular interface members. Record cannot have any pointer or array members. Record cannot have any base classes. Record can have constructor, field in-place initialization and initializer list. Record constructor cannot have function calls.

Array of records supported. Array of pointers to record supported.

Record can be passed to function by value as well as by reference/constant reference, such record must have trivial copy constructor. Record can be returned from function by value, such record must have trivial copy constructor.

```
// Simple record example
struct Rec1 {
    int x;
    sc_int<2> y;
};
// Record with constructor
struct Rec2 {
    sc_uint<16> a;
    bool b;
    Rec2(int i) : b(i == 42) {
        a = i + 1;
    }
};
```

Code generation rules for record described in Section 6.12.

Union type not supported.

5.11 Type cast

This section describes type cast operations can be used in process functions.

Type cast in C style ((T)x), functional style (T(x)), and static cast supported for right side of assignment statement and function arguments. Type cast for left side of assignment is ignored. Constant type cast ignored. Reinterpret and dynamic type cast not supported. SystemC type conversion to C++ integer methods to_int(), to_uint(), to_long(), to_ulong(),

to_int64(), to_uint64() supported. Multiple type narrowing and widening casts supported.

```
int i;
bool b;
sc_uint<4> x;
sc_uint<8> y;
b = (bool)i;
i = x.to_int();
y = (sc_uint<3>)x;
y = (sc_uint<6>)((sc_uint<2>)x);
```

```
b = |i;
i = 32'(x);
y = 3'(x);
y = 6'(2'(x));
```

Type cast to base class supported for function call (T::f()) and member access (T::m).

5.12 Dynamic memory allocation

Dynamic memory allocation is supported at elaboration phase only, i.e. in module constructors and functions called from there. Dynamic memory allocation not supported in process functions.

Dynamic allocation supported for all types including modules, interfaces, signals and ports. That is also supported for array of pointers to modules, interfaces, signals and ports.

ICSC uses dynamic elaboration that provides arbitrary C++ code support at elaboration phase, but not able to distinguish between pointer to dynamically allocated object and dangling pointer. To solve this problem ICSC uses overriding operators new and new[]. For modules, interfaces, signals, ports and other inheritors of sc_object operators new and new[] overridden in the patched SystemC library used by ICSC.

For dynamic memory allocation for non-sc_object types, like C++ types, there are special functions sc_new and sc_new_array. sc_new is used for scalar types instead of new, sc_new_array used for arrays instead of new[]. sc_new and sc_new_array declarations:

```
template<class T, class... Args>
T* sc_new(Args&&... args);

template<class T>
T* sc_new_array(size_t array_size);
```

Using sc_new and sc_new_array examples:

```

struct MyRec {
    int i;
    MyRec(int i_) : i(i_) {};
};
sc_signal<bool>* ap;
bool* bp;
sc_uint<8>* vp;
MyRec* mp;
sc_in<int>** ports;
sc_signal<int>* signals;

SC_MODULE(MyModule) {
    ap = new sc_signal<bool>("a"); // OK, signal is sc_object
    bp = new bool;                // ERROR, non-sc_object
    bp = sc_new<bool>();            // OK
    vp = new sc_uint<8>();          // ERROR, non-sc_object
    vp = sc_new<sc_uint<8>>();      // OK
    mp = new MyRec(42);             // ERROR, new for non-sc_object
    mp = sc_new<MyRec>(42);         // OK, using sc_new
    ports = new sc_in<int>* [10];   // ERROR, new for pointer, non-sc_object
    ports = sc_new_array<sc_in<int>*>(10); // OK, using sc_new_array
    signals = new sc_signal <int>[10]; // OK, array of sc_objects
};

```

5.13 Control flow operators

All control flow operators are supported. Conditions of `if`, `?`, `for`, `while`, `do..while` should be expression without side effects. Complex conditions with `||`, `&&`, `!` and brackets supported. If left part of logical expressions with `||` and `&&` evaluated as constant, right part code is not generated. That allows to check pointer is not null and do the pointer de-reference it in the condition expression. `if` and `?` conditions, including complex conditions, can contain function call without side effects and without `wait()`. `for`, `while`, `do..while` conditions cannot have any function call.

There are two kind of synthesizable loops:

- (1) loop without `wait()`/`wait(N)`, for these loops iteration number must be statically determinable,
- (2) loop with `wait()`/`wait(N)`, for these loops iteration number may be unknown.

5.13.1 *if*. `if` statement is translated into SystemVerilog `if`.

```
// Operator if examples
if (a || b) {...}
if (true || a) {...}
if (false && b) {...}
```

```
// SystemVerilog generated for if example
if (a || b) begin
    ...
end
if (1) begin
    ...
end
if (0) begin    // Empty if generated
end
```

5.13.2 switch. switch statement is translated into SystemVerilog case, see Section 6.11. switch statement can have one or more cases including optional default case. Each case must have one and only one final break as the last statement of the case. default case also must contain final break. Another option is empty case or empty default case. For empty case the next non-empty case (possibly default case) code is copied in the generated SV.

switch case code can contain if/loop statements as well as inner switch statements. switch case code can contain function calls. switch statement in called function can contains return statements in the end of all cases. For such switch cases final break statement not allowed, no mix of return and break supported.

```
// Operator switch example
switch (i) {
case 0: i++; break;
case 1: i--; break;
default: i = 0; break;
}
```

switch statement in called function can contain return statements in the end of all cases. For such switch no break statements required.

```
// Operator switch in function example
void f() {
    ...
    switch (i) {
```

```

    case 0: i++; return;
    case 1: i--; return;
  }
  return;
}

```

```

// Operator switch with empty case
switch (i) {
case 0:
case 1: k = 1; break;
default: k = 2; break;
}

```

5.13.3 for. *for* statement is translated into SystemVerilog *for* or into *if* see Section 6.10.

for loop can have only one counter variable with optional initialization, condition and increment sections. The variable can be declared in the loop initialization. Initialization section can have simple variable initialization or assignment only, cannot have function call. Condition section can have one comparison operator for the loop variable, cannot have function call. Increment section can have increment or decrement of the loop variable, cannot have function call.

Several examples of correct *for* loop is given in the following listing:

```

// Operator for examples
const unsigned N = 10;
for (int i = 0; i < N; i++) {...}

int i = N;
for (; i != 0; --i) {...}

int j = 0;
for (; j < N; ) {...}

```

5.13.4 while. *while* statement is translated into SystemVerilog *while* or into *if* see Section 6.10. *while* condition is an arbitrary expression without function call. Several examples of correct *while* loop is given in the following listing:

```

// Operator while examples
const unsigned N = 10;
int i = 0;
while (i < N) {...}

```

```

int j = N;
int k = 0;
while (j != 0 && j != k) {...}

// Waiting for enable, this while loop should contain wait() at each path
while (!enable.read()) {...}

```

5.13.5 do...while. do..while statement is translated into SystemVerilog do..while or into if see Section 6.10. do..while condition is an arbitrary expression without function call. Several examples of correct do..while loop is given in the following listing:

```

// Operator do..while examples
const unsigned N = 10;
int i = 0;
do {
    ...
} while (i < N);

int j = N;
int k = 0;
do {
    ...
} while (j != 0 && j != k);

```

5.13.6 break. break statement is translated into SystemVerilog break or substituted with code after the loop body, see Section 6.10.

5.13.7 continue. continue statement is translated into SystemVerilog continue or substituted with code in the loop body, see Section 6.10.

5.13.8 goto. Not supported.

5.14 Function calls

This section describes functions and function calls rules. Module/modular interface/record static and non-static functions supported. Global/namespace functions supported. Recursive functions not supported.

Function can have parameters and returned value. Function can have local variable of non-channel type. Local variables can be non-static or constant static. No static non-constant local variables allowed.

Function parameters can be passed by value, by reference reference, and by pointer, including pointer to channel. Constant reference parameter argument can be literal of the corresponding type. Function can return result by value only. Return by reference or by pointer not supported. Function with return type void can use return statement without argument.

```
// No parameters function
void f1() {
    m = m + 1;
}
// Parameters passed by value
int f2(int i, bool b = false) {
    return (b) ? i : i+1;
}
// Parameters passed by reference
void f3(int& i) {
    i++;
}
// Parameters passed by pointer
unsigned f4(sc_uint<16>* i) {
    return (*i+1);
}
// Array passed
int f5(int arr_par[3]) {
    int res = 0;
    for (int i = 0; i < 3; i++) {
        res += arr_par[i];
    }
    return res;
}
}
```

Function with multiple return supported. Function return statement(s) is replaced with function result to variable assignment in SV code. That leads to a function must have no code after return. In particular, return statement in loop is not supported. Function with return type void can have return statement(s) without argument. For return statement without argument no code is generated.

```
// Multiple returns
int f6(int& val) {
    if (val) {
        return 1;
    } else {
        return 2;
    }
}
```

```

}
// Multiple returns in switch
unsigned f7(unsigned val) {
    switch (val) {
        case 1: return 1;
        case 2: return 2;
        default: return 3;
    }
}

```

Module member function can access this module fields/functions and child modular interface instance(s) fields/functions. Access to child modular interface instance members allowed through a port interface (`sc_port<IF>`) or a pointer to the modular interface. The accessed modular interface is flatten in module/modular interface with does access to it, see [5.1](#).

Virtual functions supported. Function overload and hide function in child class supported.

5.15 Naming restrictions

Prefixes `sct_` and `SCT_` are used for special function and cannot be used in user SystemC code. Suffix `_next` is used for register variables, so it is not recommended to use such suffix for SystemC variables.

ICSC tool provides `__SC_TOOL__` define for input SystemC project translation. Module/interface field `__SC_TOOL_MEMORY_NAME__` is reserved for vendor memory name. Module/interface field `__SC_TOOL_VERILOG_MOD__` is reserved for disable module generation in SystemVerilog code.

6 SYSTEMVERILOG GENERATION RULES

This section describes how C++/SystemC modules, processes, functions, declarations, statements and expressions are translated into SystemVerilog code. For an input SystemC design ICSC generates one output SystemVerilog file with all the modules inside.

6.1 Module generation

Module hierarchy generation started with top module and passed through all the child modules. For any module there are multiple SV code sections generated:

- Input and output ports;
- Variables generated for SystemC signals;
- Local parameters generated for C++ constants;
- Assignments generated for SystemC array of channels;
- Process variables and always blocks;
- Child module instances;
- SVA generated for SystemC temporal assertions.

Assignments for SystemC array of channels provides conversion of array elements to individual channels in module interface. That is required if individual array elements bound to different channels. In case array of channels is bound to other same size array of channels, module interface contains array of channels and no assignments required.

There is an illustrative example for module with all the sections generated:

```
// Module structure example
SC_MODULE(MyChild) {...};

SC_MODULE(MyModule) {
    sc_in_clk          clk{"clk"};
    sc_in<sc_uint<4>>  a{"a"};
    sc_out<sc_int<5>>  b{"b"};
    sc_in<bool>        f[2];

    sc_signal<bool>    s{"s"};
    sc_signal<sc_uint<32>> t{"t"};

    MyChild            m{"m"};

    static const bool C = true;
    static const int  D = 42;

    SC_CTOR(MyModule) {
        SC_CTHREAD(threadProc, clk.pos());
    }
}
```

```

    async_reset_signal_is(rst, 0);
}

void proc() {
    ...
}
};

```

```

// Generated SystemVerilog
// Input and output ports
module MyModule // "tb_inst.top_mod"
(
    input logic clk,
    input logic [3:0] a,
    output logic signed [4:0] b,
    input logic f0,
    input logic f1
);

//-----
// Variables generated for SystemC signals
logic s;
logic [31:0] t;
logic f[2];

//-----
// Local parameters generated for C++ constants
localparam logic signed C = 'd1;
localparam logic signed [31:0] D = 'd42;

//-----
// Assignments generated for SystemC array of channels
assign f[0] = f0;
assign f[1] = f1;

//-----
// Process variables and always blocks
// Thread-local variables
logic [31:0] t_next;
// Next-state combinational logic
always_comb begin : threadProc_comb // test_module_sections.cpp:89:5
...

```

```

end
// Synchrononous register update
always_ff @(posedge clk or negedge rst)
begin : threadProc_ff
...
end

//-----
// Child module instances
MyChild m
(
...
);

//-----
// SVA generated for SystemC temporal assertions
`ifndef INTEL_SVA_OFF
sctAssertLine55 : assert property (...)
`endif // INTEL_SVA_OFF
endmodule

```

6.2 Variables generation

There are several types of variables used in SC process functions: local variables (non-channel only), member non-channel variables, member channel variables. For these types of SC variables different types of SV variables are generated depends on variable usages.

Non-channel variable can be generated as a register or as a combinational variable. In clocked thread process:

- If non-channel variable defined before read after last wait()/wait(N) at every path, it is combinational variable;
- If non-channel variable can be read before defined after last wait()/wait(N), it is register variable.

In method process only combinational variables are generated. That is because a non-channel variable must be defined before used.

Channel variable can be generated as an input or a register:

- If channel variable is never assigned, it is an input (no register);
- If channel variable is assigned at least once, it is a register.

If a signal/port register is not initialized in reset section, warning is reported.

Register in generated code consists of two variables: *current value* and *next value*. The current value variable is updated in `always_ff` block with next value variable. The next value variable is updated in `always_comb` block with current value variable.

In the following example, local variable `j` there will be `j` and `j_next` SV variables generated. SV variable `j` stores old value of the variable, assigned at last process activated cycle. `j_next` stores a new value assigned in the current cycle.

```

sc_out<int> a; // Member channel
sc_signal<int> b; // Member channel
int c; // Member non-channel
int d; // Member non-channel

void threadProc() {
    int j = 0;
    a = 0;
    c = 0;
    d = 1;
    wait();

    while (true) {
        a = j; // Channel a is defined
        int i = 0; // Local j is defined before use
        j = b + i; // Local j is used before define
        c = b; // Non-channel c is defined before use
        i = a.read() + c + d; // Local i is not used
        d = a.read(); // Non-channel d is used before define
        wait();
    }
}

```

```

// SystemVerilog generated
module MyModule // "tb_inst.top_mod"
(
    ...
    output logic signed [31:0] a
);

// Thread-local variables
logic signed [31:0] a_next;
logic signed [31:0] j;
logic signed [31:0] j_next;

```

```

logic signed [31:0] d;
logic signed [31:0] d_next;
logic signed [31:0] c;

// Next-state combinational logic
always_comb begin : threadProc_comb // test_module_sections.cpp:89:5
    threadProc_func;
end
function void threadProc_func;
    integer i;
    a_next = a;
    d_next = d;
    j_next = j;
    a_next = j_next;
    i = 0;
    j_next = b + i;
    c = b;
    i = a + c + d_next;
    d_next = a;
endfunction

// Synchronous register update
always_ff @(posedge clk or negedge rst)
begin : threadProc_ff
    if ( ~rst ) begin
        integer c;
        j <= 0;
        a <= 0;
        c = 0;
        d <= 1;
    end
    else begin
        a <= a_next;
        j <= j_next;
        d <= d_next;
    end
end

```

6.3 Constants generation

For constants and static constants SV local parameters are generated. There is `KEEP_CONST_VARIABLES` option to replace the constants in the code with their values, see 4.7. Constant is replaced with

its value if the value if there is no reference to the constant. If constant is replaced with its value or not used in the code, no SV local parameter is generated.

```
static const bool C = true;
static const int D = 42;
```

```
// Generated SystemVerilog
// C++ constants
localparam logic signed C = 'd1;
localparam logic signed [31:0] D = 'd42;
```

6.4 Method process generation

Method process is directly translated into `always_comb` block. All the local variables of the method are translated into local variables in the `always_comb` block.

```
// Method process example
void methodProc() {
    bool x;
    int i;
    i = a.read();
    x = i == b.read();
    sig = (x) ? i : 0;
}
```

```
// SystemVerilog generated for method process example
always_comb
begin : methodProc // test_module_sections.cpp:110:5
    logic x;
    integer i;
    i = a;
    x = i == b;
    sig = x ? i : 0;
end
```

6.5 Method process with empty sensitivity

Method process with empty sensitivity are typically used to assign constant value signal/port initialization. In the SystemVerilog code one or more assign statements are generated for such process.

```
// SystemC method process with empty sensitivity
static const bool COND = true;
void emptySens()
{
    a = 0;
    if (COND) {
        b = 1;
    } else {
        c = 2;
    }
    int i = 1;
    d = (!COND) ? i : i + 1;
}
```

```
// SystemVerilog generated
assign a = 0;
assign b = 1;
assign i = 1;
assign d = i+1;
```

6.6 Method process with latch(es)

Normally ICSC does not allow to have latch in SystemC source, but there are some cases where latch is required. There is ICSC assertion `sct_assert_latch` which intended to specify latch in method process, see 8.3. It suppresses ICSC error message for latch variable, signal or port.

Normal method process translated into `always_comb` block in SystemVerilog. For method with latch `always_latch` block is generated.

```
// SystemC source for Clock Gate cell
#include "sct_assert.h"
void cgProc() {
    if (!clk_in) {
        enable = enable_in;
    }
}
```

```

    // To prevent error reporting for latch
    sct_assert_latch(enable);
}
void cgOutProc() {
    clk_out = enable && clk_in;
}

```

```

// SystemVerilog generated
always_latch
begin : cgProc
    if (!clk_in) begin
        enable <= enable_in;
    end
end
end

```

6.7 Thread process generation

Clocked thread process in SystemC has one or multiple states specified with `wait()`/`wait(N)` calls, therefore it cannot be directly translated into `always_ff` block. To translate semantics of multi-state thread into a form which is accepted by most SystemVerilog tools, ICSC converts the thread into pair of `always_comb` and `always_ff` blocks. `always_ff` block implements reset and update logic for state register and other registers. `always_comb` block contains combinational logic that computes the next state. `always_comb` contains all the logic of the SystemC thread and provides combinational outputs which are stored into registers in the `always_ff` block.

Thread states are identified by `wait()` call statements in the SystemC code, so number of states is the number of `wait()` calls, including such in called functions. `wait(N)` calls are discussed in 6.9.

For thread process with one state, generated `always_comb` block has all the after-reset logic.

For thread with multiple states, generated `always_comb` block has main case for thread process states. The main case branches correspond to SystemC code that starts from specific `wait()` and finishes in the next `wait()`. Implicit thread states are represented by automatically generated `PROC_STATE` variable with process name prefix.

```

// Thread process example
void multiStateProc() {
    sc_uint<16> x = 0;
    sig = 1;
    wait();                // STATE 0
}

```

```

    while (true) {
        sc_uint<8> y = a.read();
        x = y + 1;
        wait();           // STATE 1
        sig = x;
    }
}

```

```

// SystemVerilog generated for thread process example
// Thread-local variables
...
logic multiStateProc_PROC_STATE;
logic multiStateProc_PROC_STATE_next;

// Next-state combinational logic
always_comb begin : multiStateProc_comb // test_module_sections.cpp:122:4
    multiStateProc_func;
end
function void multiStateProc_func;
    logic [7:0] y;
    sig_next0 = sig;
    x_next = x;
    multiStateProc_PROC_STATE_next = multiStateProc_PROC_STATE;

    case (multiStateProc_PROC_STATE)
        0: begin
            y = a;
            x_next = y + 1;
            multiStateProc_PROC_STATE_next = 1; return;
        end
        1: begin
            sig_next0 = x_next;
            y = a;
            x_next = y + 1;
            multiStateProc_PROC_STATE_next = 1; return;
        end
    endcase
endfunction

// Synchronous register update
always_ff @(posedge clk or negedge rst)

```

```

begin : multiStateProc_ff
    if ( ~rst ) begin
        x <= 0;
        sig <= 1;
        multiStateProc_PROC_STATE <= 0; // test_module_sections.cpp:125:8;
    end
    else begin
        sig <= sig_next0;
        x <= x_next;
        multiStateProc_PROC_STATE <= multiStateProc_PROC_STATE_next;
    end
end
end

```

6.8 Thread process without reset

Thread process without reset supported with limitations: such process can have only one wait() and cannot have any code in reset section.

```

// Clocked thread without reset
SC_CTOR(test_reset) {
    SC_CTHREAD(proc, clk.pos());
}

```

```

void proc() {
    while (true) {
        int i = 0;
        wait();
    }
}

```

```

// SystemVerilog generated for clocked thread without reset
always_comb begin
    ...
    case (PROC_STATE)
        default : begin
            i = 0;
        end
        0: begin
            i = 0;
        end
    endcase
end

```

endfunction

```
always_ff @(posedge clk)
begin
    begin
        PROC_STATE <= PROC_STATE_next;
    end
end
```

6.9 wait(N) conversion

For wait(N) call auxiliary variable counter is generated. This counter is set at entry to wait(N) state and decremented at wait(N) state until becomes one. There is one counter per thread process, which used for all wait(N) calls in the thread. The counter variable has required width.

```
// Clocked thread with wait(N)
void threadProc () {
    wait();
    while (true) {
        wait(3);
    }
}
```

```
// Generated Verilog
logic [1:0] waitn_WAIT_N_COUNTER;
logic [1:0] waitn_WAIT_N_COUNTER_next;
...
function void waitn_func;
    waitn_WAIT_N_COUNTER_next = waitn_WAIT_N_COUNTER;
    waitn_PROC_STATE_next = waitn_PROC_STATE;

    case (waitn_PROC_STATE)
        0: begin
            waitn_WAIT_N_COUNTER_next = 3;
            waitn_PROC_STATE_next = 1; return;
        end
        1: begin
            if (waitn_WAIT_N_COUNTER != 1) begin
                waitn_WAIT_N_COUNTER_next = waitn_WAIT_N_COUNTER - 1;
            end
        end
    endcase
endfunction
```

```

        waitn_PROC_STATE_next = 1; return;
    end;
    waitn_WAIT_N_COUNTER_next = 3;
    waitn_PROC_STATE_next = 1; return;
end
endcase
endfunction

```

6.10 Loops with wait generation

Loop with wait()/wait(N) call cannot be directly translated into SystemVerilog loop. Loop with wait()/wait(N) is divided into several states. For such loop statement if with loop condition is generated in SV code. If the loop has at least one iteration, the loop first iteration has no check of the loop condition.

```

void loopProc() {
    enable = 0;
    wait();                // STATE 0
    while (true) {
        for (int i = 0; i < 3; ++i) {
            enable = 0;
            wait();        // STATE 1
        }
        enable = 1;
    }
}

```

```

function void loopProc_func;
    enable_next = enable; i_next = i;
    loopProc_PROC_STATE_next = loopProc_PROC_STATE;

    case (loopProc_PROC_STATE)
        0: begin
            i_next = 0;                // No loop condition check
            enable_next = 0;
            loopProc_PROC_STATE_next = 1; return;
        end
        1: begin
            ++i_next;
            if (i_next < 3)

```

```

        begin
            enable_next = 0;
            loopProc_PROC_STATE_next = 1; return;
        end
        enable_next = 1;
        i_next = 0;
        enable_next = 0;
        loopProc_PROC_STATE_next = 1; return;
    end
endcase
endfunction

```

If a loop with `wait()`/`wait(N)` contains `break` or `continue` statements, they are removed and control flow analysis traverses further to the loop exit or the loop entry correspondingly. That means `break` and `continue` statements are replaced with some code up to the next `wait()` call. Example below contains while loop with `break`, which is replaced with code marked with `break begin` and `break end` comments.

```

void breakProc() {
    ready = 0;
    wait();                // STATE 0
    while (true) {
        wait();            // STATE 1
        while (!enable) {
            if (stop) break;
            ready = 1;
            wait();        // STATE 2
        }
        ready = 0;
    }
}

```

```

function void breakProc_func;
    ready_next = ready;
    breakProc_PROC_STATE_next = breakProc_PROC_STATE;

    case (breakProc_PROC_STATE)
        0: begin
            breakProc_PROC_STATE_next = 1; return;
        end
    endcase
endfunction

```

```

1: begin
  if (!enable)
    begin
      if (stop)
        begin
          // break begin
          ready_next = 0;
          breakProc_PROC_STATE_next = 1;
          // break end
        end
      ready_next = 1;
      breakProc_PROC_STATE_next = 2;
    end
  ready_next = 0;
  breakProc_PROC_STATE_next = 1; return;
end
2: begin
  ...
end
endcase
endfunction

```

6.11 Switch generation

switch statement is translated into SystemVerilog case.

```

// Operator switch example
switch (i) {
case 0: i++; break;
case 1: i--; break;
default: i = 0; break;
}

```

```

// SystemVerilog generated for switch example
case (i)
0 : begin
  i++;
end
1 : begin
  i--;
end
end

```

```

default : begin
    i = 0;
end
endcase

```

switch with default case only is removed, the default case statement translated into SV code.

```

// Operator switch with default case only
switch (a) {
    default: i = 1; break;
}
j = 2;

```

```

// SystemVerilog generated for switch with default case only
i = 1;
j = 2;

```

Another special case is switch with empty case(s). Empty case does not have break or return at the end is translated into SV code of the first following non-empty case.

```

// Operator switch with empty case
switch (i) {
case 0:
case 1: k = 1; break;
default: k = 2; break;
}

```

```

// SystemVerilog generated for switch with empty case
case (i)
0 : begin // Empty case without break
    k = 1;
end
1 : begin
    k = 1;
end
default : begin
    k = 2;
end
endcase

```

6.12 Records generation

Records can be used as member and local non-channel variables. Record cannot be used as type parameter of a channel. Record fields are generated in SV code as separate variables. Field variable name in SV code is field name with record variable name prefix.

In the following example there is local record variable with non-empty constructor. Constructor code inserted after record field variables declaration.

```
// Local variables record type
struct Rec1 {
    int x;
    sc_int<2> y;
    Rec1() : y(1) {
        x = 2;
    }
};
void record_local_var1() {
    Rec1 r;
    r.x = r.y + 2;
}

always_comb
begin : record_local_var1 // test_simple_method.cpp:110:5
    integer r_x;
    logic signed [1:0] r_y;
    r_y = 1;
    // Call Rec1() begin
    r_x = 2;
    // Call Rec1() end
    r_x = r_y + 2;
end
```

Records can be used in thread processes. In thread process record variable can be register as well as combinational variable.

```
// Local/global records assign in initialization
SinCosTuple grr;
void record_glob_assign1() {
    wait();
    while (true) {
        grr.cos = 1;
        SinCosTuple r = grr;
    }
}
```

```

        wait();
    }
}

// Thread-local variables
logic signed [31:0] grr_sin;
logic signed [31:0] grr_sin_next;
logic signed [31:0] grr_cos;
logic signed [31:0] grr_cos_next;
// Next-state combinational logic
always_comb begin : record_glob_assign1_comb // test_reg_cthread1.cpp:91:5
    record_glob_assign1_func;
end
function void record_glob_assign1_func;
    integer r_sin;
    integer r_cos;
    grr_cos_next = grr_cos;
    grr_sin_next = grr_sin;
    grr_cos_next = 1;
    r_sin = grr_sin_next; r_cos = grr_cos_next;
endfunction

```

6.13 Arithmetical operations

SystemC/C++ type promotion rules differ from each other and from SV rules. For literal and non-literal terms signed-to-unsigned and unsigned-to-signed implicit cast detected and used as base for sign in SV.

6.13.1 Signed and unsigned literals. SystemC/C++ literals translated into SV literals with the following rules:

- Zero not casted.
- C++ literal has integer type and represented in simple form is signed.
- C++ literal with suffix U represented in based form is unsigned.
- SC literal represented in based form and is signed or unsigned depends on its type.
- All negative literals are signed.

In binary operators if first argument is signed type non-literal and second is no cast literal, second argument is casted to signed literal.

6.13.2 Signed and unsigned types. The following rules based on C++ type promotion rules and SC types operators implementation. General idea is non-literal mix of signed and unsigned considered as signed and unsigned operand converted to signed in SV with signed'. If signed

literal mixed with unsigned non-literal, that considered as unsigned arithmetic, no signed cast in SV.

enum types can be signed and unsigned as well. There is implicit cast to `int`, so unsigned enum casted to signed, no special rule required. There are several rules for non-literals in binary operators (+, -, *, /, &, ^, |, %):

- If first argument has signed-to-unsigned cast or signed expression and second has no cast and is not signed type or signed expression, second argument is casted to signed.
- If first argument of `sc_bigint` type and second is not signed type or signed expression, second argument is casted to signed.
- If first argument of `sc_biguint` type and second is signed type, first argument is casted to signed.

Mixing negative signed operand and unsigned/`sc_uint` operand can provide incorrect result, so warning is reported. This operations work well with `sc_biguint`.

There is rule for non-literals in unary operators, if argument of `sc_biguint` type in unary minus, the argument is casted to signed.

6.13.3 Type cast in assignment. Explicit type cast can be used to narrow/widen the argument. It translated into SV type cast. Multiple type casts are also possible: internal one narrows value, external one extend type width, which may be required for concatenation.

Implicit type cast, including SC data type constructor, not translated to SV. SV has implicit narrowing of argument, for example:

```
sc_uint<3> i;
sc_uint<2> k = i;
bool b = k;
```

```
// Generated SystemVerilog
logic [2:0] i;
logic [1:0] k;
logic b;
k = i;    // All that work fine in ICSC, no warnings
b = k;
```

Explicit cast can be combined with signed cast. In this case signed cast extend data width by 1 only if required (signed changed after explicit cast):

```
sc_int<6> x;
sc_uint<4> ux;
z = x + ux;
z = x + sc_uint<6>(ux);
z = x + sc_int<7>(ux);
```

```
// Generated SystemVerilog
z = x + signed' ({1'b0, ux});
z = x + 6'(ux);
z = x + signed' (7'(ux));
```

6.13.4 Operator comma. Operator `,` is applied to concatenate two SC integers with specified length. For C++ integers and results of operation explicit type conversion is required.

```
sc_uint<N> i;
sc_uint<M> j;
sc_uint<N+M> k;
k = (i, j); // OK
k = (i, sc_uint<M>(j)); // OK, extra type conversion
k = (i, i*j); // Error reported, type conversion required
k = (i, sc_uint<M>(i*j)); // OK
```

6.14 Name uniqueness

Name uniqueness in SystemC design must be provided in terms of C++ rules. There is no other rules for names of types, variables or functions. If there is name collision caused by code transformation, it resolved with adding numeric suffix.

```
// Name uniqueness example
sc_signal<bool> b; // b
void uniqProc() {
    out = b;
    bool b; // b_1
    {
        bool b; // b_2
        out = b;
    }
    out = b;
}
```

```
// SystemVerilog generated
logic b;
always_comb
begin : uniqProc // test_module_sections.cpp:186:1
    logic b_1;
```

```
    logic b_2;  
    out = b;  
    out = b_2;  
    out = b_1;  
end
```

Function name for modular interface process has prefix to distinguish it from the parent module processes. If process function name conflict with another name it gets numeric suffix like variables.

7 EXTENSIONS

7.1 Advanced FIFO

Advanced FIFO is a collection of FIFO modules with push and pop interfaces. Advanced FIFO is intended for interaction between two processes which can be called producer and consumer. The FIFO is used to store pushed request from producer when consumer is not ready to pop. Each of the processes can be method or thread, they may be located in the same or in different modules.

Advanced FIFO is implemented as a SystemC modules with signal and function interfaces. There are the following modules in `sct_fifo.h`:

- `adv_fifo_base` – base class for all FIFO modules, not intended to be used;
- `adv_fifo` – normal FIFO with signal interface;
- `adv_fifo_mif` – normal FIFO with function interface, uses `adv_fifo` inside;
- `mcp_request_fifo` – MCP (multi-clock path) FIFO for producer operating at lower frequency;
- `mcp_response_fifo` – MCP FIFO for consumer operating at lower frequency.

FIFO signal interface has the following inputs/outputs:

<code>sc_in_clk</code>	<code>clk;</code>	<code>// Common clock, positive edge is used</code>
<code>sc_in<bool></code>	<code>nrst;</code>	<code>// Asynchronous reset, low active</code>
<code>sc_in<bool></code>	<code>push;</code>	<code>// Push data into FIFO</code>
<code>sc_in<T></code>	<code>data_in;</code>	<code>// Input data</code>
<code>sc_out<bool></code>	<code>ready_to_push;</code>	<code>// Ready to push</code>
<code>sc_in<bool></code>	<code>pop;</code>	<code>// Pop data from FIFO</code>
<code>sc_out<T></code>	<code>data_out;</code>	<code>// Output data</code>
<code>sc_out<bool></code>	<code>out_valid;</code>	<code>// Output data is valid</code>
<code>sc_out<bool></code>	<code>almost_full;</code>	<code>// FIFO is almost full, number of free slots</code>
		<code>// equal or less than AFULL_ELEMENT_NUM</code>

FIFO function interface has the following methods:

```

// FIFO is ready to push
bool ready();
// Push or clear push, push is ignored if FIFO is not ready to push
// \return ready to push flag
bool push(const T& data, bool push = true);
// FIFO output data is valid, pop return can be used only if data is valid
bool valid();
// Pop or get data from FIFO, do not remove data from FIFO if pop = false
T pop(bool pop = true);
// FIFO is almost full, there is AFULL_ELEMENT_NUM elements or more used
bool full();

```

```
// Add FIFO signals to sensitivity list
void addTo(sc_sensitive& s);
// Bind FIFO clock and reset
template <typename CLK_t, typename RSTN_t>
void clk_nrst(CLK_t& clk_in, RSTN_t& nrst_in);
```

Advanced FIFO module has template parameters which specify FIFO size and other parameters:

```
template<
    typename T,                // FIFO slot data type
    bool ASYNC_VALID,          // Assert out_valid combinationally
    bool ASYNC_READY,          // Assert ready_to_push combinationally
    bool ASYNC_AFULL,          // Assert almost_full combinationally
    unsigned FIFO_LENGTH,      // Number of FIFO slots
    unsigned AFULL_ELEMENT_NUM, // Number of free element slots in FIFO
                                // when almost_full is asserted
    bool INIT_BUFFER           // Initialize FIFO slots in reset with zeros
>
class adv_fifo : public sc_module {...}
```

The FIFO push and pop may be asserted whenever. FIFO does push operation when both push/ready_to_push asserted. FIFO does pop operation when both pop/out_valid asserted.

Outputs out_valid, ready_to_push and almost_full may be asserted combinationally, de-assertion of them is done synchronously only.

The FIFO allows to pop an element in the same clock it is pushed into empty FIFO, if ASYNC_VALID is true (out_valid is asserted combinationally).

The FIFO allows to push an element into full FIFO if there is pop operation in the same clock, if ASYNC_READY is true (ready_to_push is asserted combinationally).

The FIFO provides combinational assertion of almost_full when FIFO is one element lower than AFULL_ELEMENT_NUM and there is push and no pop. This feature is not MCP ready.

7.2 SystemVerilog intrinsic insertion

This section describe how to insert SystemVerilog intrinsic ("black box") module.

ICSC supports replacement a SystemC module with given SystemVerilog intrinsic module. In this case no parsing of the SystemC module is performed, so this module can contain non-synthesizable code. To replace SystemC module it needs to define __SC_TOOL_VERILOG_MOD__ variable of std::string type in the module body. __SC_TOOL_VERILOG_MOD__ value can be specified in place or in the module constructor.

There are two common usages:

- (1) Replace with given SystemVerilog module: `__SC_TOOL_VERILOG_MOD__` contains SV module code or `#include` directive;
- (2) Do not generate module at all: `__SC_TOOL_VERILOG_MOD__` is empty string.

In second case SystemVerilog module implementation needs to be provided in an external file.

```
struct my_register : sc_module {
    std::string __SC_TOOL_VERILOG_MOD__[] = R"(
        module my_register (
            input logic [31:0] din,
            output logic [31:0] dout
        );
        assign dout = din;
        endmodule)";

    SC_CTOR (my_register) {...}
    ...
}
```

```
// SystemVerilog generated
// Verilog intrinsic for module: my_register
module my_register (
    input logic [31:0] din,
    output logic [31:0] dout
);
assign dout = din;
endmodule
```

7.3 Memory module name

This section describes how to create own memory module with module name specified.

To support vendor memory it needs to specify memory module name at instantiation point and exclude the SV module code generation (memory module is external one). To exclude SV module code generation empty `__SC_TOOL_VERILOG_MOD__` should be used. To specify memory module name it needs to define `__SC_TOOL_MEMORY_NAME__` variable in the module body and initialize it with required name string.

If there are two instances of the same SystemC module, it is possible to give them different names, but `__SC_TOOL_VERILOG_MOD__` must be declared in the module. If `__SC_TOOL_VERILOG_MOD__` is not declared the SystemC module, only one SV module with first given name will be generated.

Module name could be specified for module with non-empty `__SC_TOOL_VERILOG_MOD__`, but module names in `__SC_TOOL_MEMORY_NAME__` and `__SC_TOOL_VERILOG_MOD__` should be the same.

If specified module name in module without `__SC_TOOL_VERILOG_MOD__` declaration conflicts with another module name, it updated with numeric suffix. Specified name in module with `__SC_TOOL_VERILOG_MOD__` declaration never changed, so name uniqueness should be checked by user.

```
// Memory stub example
struct memory_stub : sc_module {
    // Disable Verilog module generation
    std::string __SC_TOOL_VERILOG_MOD__[] = "";
    // Specify module name at instantiation
    std::string __SC_TOOL_MEMORY_NAME__;
    explicit memory_stub(const sc_module_name& name,
                        const char* verilogName = "") :
        __SC_TOOL_MEMORY_NAME__(verilogName)
    {}
};

// Memory instance at some module
memory_stub stubInst1{"stubInst1", "pxxxrf256x32ben"};
memory_stub stubInst2{"stubInst2", "pxxxsram1024x32ben"};
memory_stub stubInst3{"stubInst3"};
stubInst1.clk(clk);
stubInst2.clk(clk);
stubInst3.clk(clk);
...
```

```
// SystemVerilog generated
pxxxrf256x32ben  stubInst1(.clk(clk), ...);
pxxxsram1024x32ben stubInst2(.clk(clk), ...);
memory_stub      stubInst3(.clk(clk), ...);
```

8 ADVANCED VERIFICATION FEATURES

8.1 Immediate assertions

There are several types of C++, SystemC, and ICSC assertions to use in design verification:

- `assert(expr)` – general C++ assertion, in case of violation leads to abort SystemC simulation, ignored by ICSC;
- `sc_assert(expr)` -- SystemC assertion, leads to report fatal error (`SC_REPORT_FATAL`), ignored by ICSC;
- `sct_assert(expr [, msg = ""])` -- ICSC assertion, in simulation has the same behavior as `assert`, SVA generates System Verilog assertion (SVA) for it. Second parameter `const char* msg` is optional, contains message to print in simulation and used in SVA error message;

Immediate assertions are declared in `sct_assert.h` (`include/sct_common/sct_assert.h`). This assertion can be used for SystemC simulation as well as for generated Verilog simulation. In generated Verilog there is equivalent SVA `assert` with error message if specified. Error message should be string literal (`const char*`).

```
// SystemC source
#include "sct_assert.h"
void sct_assert_method() {
    sct_assert(cntr == 1);
    sct_assert(!enable, "User error message");
}
```

```
// Generated SystemVerilog
assert (cntr == 1) else $error("Assertion failed at test_sva_assert.cpp:55:9");
assert (!enable) else $error("User error message at test_sva_assert.cpp:56:9");
```

SVA for `sct_assert` generated in `always_comb` block that requires to consider exact delta cycle when used in the assertion signals/ports changed their values. That makes using `sct_assert` more complicated than temporal assertion `SCT_ASSERT` which described below. So, it is strongly recommended to use `SCT_ASSERT(expr, clk.pos())` instead of `sct_assert(expr)`.

8.2 Temporal assertions

Temporal assertions in SystemC intended to be used for advanced verification of design properties with specified delays. These assertions looks similar to System Verilog assertions (SVA). The assertions can be added in SystemC design in module scope and clocked thread process:

```
SCT_ASSERT(EXPR, EVENT);           // In module scope
SCT_ASSERT(LHS, TIME, RHS, EVENT); // In module scope
SCT_ASSERT(LHS, TIME, RHS);        // In clocked thread
SCT_ASSERT_LOOP(LHS, TIME, RHS, ITER); // In for-loop inside of clocked thread
```

These ways are complementary. Assertions in module scope avoids polluting process code. Assertions in clock thread allows to use member and local variables. Assertions in loop can access channel and port arrays.

Temporal assertions in module scope and clocked thread have the same parameters:

- **EXPR** – assertion expression, checked to be true,
- **LHS** – antecedent assertion expression which is pre-condition,
- **TIME** – temporal condition is specific number of cycles or cycle interval,
- **RHS** – consequent assertion expression, checked to be true if antecedent expression was true in past,
- **EVENT** – cycle event which is clock positive, negative or both edges.
- **ITER** – loop iteration counter variable(s) in arbitrary order.

Assertion expression can be arithmetical or logical expression, with zero, one or several operands. Assertion expression cannot contain function call and ternary operator ?.

Temporal condition specified with:

```
SCT_TIME(TIME) -- time delay, TIME is number of cycles,
SCT_TIME(LO_TIME, HI_TIME) -- time interval in number of cycles.
```

Temporal condition specifies time delay when RHS checked after LHS is true. Temporal condition is number of cycles or cycle interval, where cycle is clock period. Specific number of cycles is integer non-negative number. Cycle interval has low time and high time, each of them is integer non-negative number. Low time and high time can be the same. There is reduced form of time condition with brackets only.

Temporal assertions are declared in `sct_assert.h` (`include/sct_common/sct_assert.h`), it needs to be included.

To disable temporal assertions macro `SCT_ASSERT_OFF` should be defined. That can be required to use another HLS tools which does not support these assertions. To avoid SVA assertion generating `NO_SVA_GENERATE` option of `svc_target` should be used.

8.2.1 Temporal assertions in module scope. Temporal assertions in module scope added with

```
SCT_ASSERT(EXPR, EVENT);
SCT_ASSERT (LHS, TIME, RHS, EVENT);
```

If `clk` is clock input, then `EVENT` specified with `clk.pos()`, `clk.neg()` or `clk` correspondingly.

Assertion expression can operate with signals, ports, template parameters, constants and literals. Member data variables (not signals/ports) access in assertion leads to data race and therefore not supported.

There are several examples:

```

static const unsigned N = 3;
sc_in<bool> req;
sc_out<bool> resp;
sc_signal<sc_uint<8>> val;
sc_signal<sc_uint<8>>* pval;
int m;
sc_uint<16> arr[N];
...
SCT_ASSERT(req || val == 0, clk.pos());           // OK
SCT_ASSERT(req, SCT_TIME(1), resp, clk.pos()); // OK
SCT_ASSERT(req, SCT_TIME(N+1), resp, clk.neg()); // OK, constant in time parameter
SCT_ASSERT(req, (2), val.read(), clk);           // OK, brackets only form
SCT_ASSERT(val, SCT_TIME(2,3), *pval, clk.pos()); // OK, time interval
SCT_ASSERT(arr[0], (N,2*N), arr[N-1], clk.pos()); // OK, brackets only form
SCT_ASSERT(val == N, SCT_TIME(1), resp, clk.pos()); // OK, constant in assertion
    expression
SCT_ASSERT(m == 0, (1), resp, clk.pos());         // Error, member data variable used
SCT_ASSERT(resp, (0,2), arr[m+1], clk.pos());    // Error, access array at
    non-literal index

```

Generated SVA:

```

`ifndef INTEL_SVA_OFF
sctAssertLine80 : assert property (
    @(posedge clk) true |-> req || val == 0 );
sctAssertLine81 : assert property (
    @(posedge clk) req |=> resp );
sctAssertLine82 : assert property (
    @(negedge clk) req |-> ##4 resp );
sctAssertLine83 : assert property (
    @(negedge clk) req |-> ##2 val );
...
`endif // INTEL_SVA_OFF

```

8.2.2 Temporal assertions in clocked thread process. Temporal assertions in clocked thread added with

```
SCT_ASSERT (LHS, TIME, RHS);
```

No clock parameter required, as clock cycle determined by the current process sensitivity. These assertions can operate with local data variables and local/member constants. They also can operate with module member data variables which are modified in this process.

Assertion in thread process can be placed in reset section (before first `wait()`) or after reset section before main infinite loop. Assertions in main loop not supported. Assertions can be placed in if branch scopes, but this if must have statically evaluated condition. Variable condition of assertion should be considered in its antecedent (left) expression.

```

void thread_proc() {
    // Reset section
    ...
    SCT_ASSERT(req, SCT_TIME(1), ready); // Assertions in reset section
    wait();
    SCT_ASSERT(req, SCT_TIME(2,3), resp); // Assertions after reset section

    // Main loop
    while (true) {
        ...                                // No assertion in main loop
        wait();
    }
}

```

Assertion in reset section generated in the end of `always_ff` block, that makes it active under reset. Assertion after reset section generated in else branch of the reset if, that makes it inactive under reset.

```

// Generated Verilog code
always_ff @(posedge clk or negedge nrst) begin
    if (~nrst) begin
        ...
    end else
    begin
        ...
        assert property (req |-> ##[2:3] resp); // Assertions after reset section
    end
    assert property (req |=> ready);           // Assertions from reset section
end

```

There an example with several assertions:

```

static const unsigned N = 3;
sc_in<bool> req;
sc_out<bool> resp;
sc_signal<bool> resp;

```

```

sc_uint<8> m;
...
void thread_proc() {
    int i = 0;
    SCT_ASSERT(req, SCT_TIME(0), ready);      // OK
    SCT_ASSERT(req, SCT_TIME(N+1), ready);    // OK, constant in time parameter
    SCT_ASSERT(req, (2,3), i == 0);           // OK, local variable used
    wait();
    if (N > 1) {
        SCT_ASSERT(req, SCT_TIME(1), resp);   // OK, statically evaluated condition
    }
    SCT_ASSERT(m > 1, (2), ready);             // OK, member variable used
    while (true) {
        ...
        SCT_ASSERT(req, SCT_TIME(0), ready);  // Error, assertion in main loop
        wait();
    }
}

```

8.2.3 Temporal assertions in loop inside of clocked thread. Temporal assertions in loop inside of clocked thread added with

```
SCT_ASSERT_LOOP (LHS, TIME, RHS, ITER);
```

ITER parameter is loop variable name or multiple names separated by comma.

Loop with assertions can be in reset section or after reset section before main infinite loop. The loop should be for-loop with statically determined number of iteration and one counter variable. Such loop cannot have wait() in its body.

```

void thread_proc() {
    // Reset section
    ...
    for (int i = 0; i < N; ++i) {
        SCT_ASSERT_LOOP(req[i], SCT_TIME(1), ready[i], i);
        for (int j = 0; j < M; ++j) {
            SCT_ASSERT_LOOP(req[i][j], SCT_TIME(2), resp[i][N-j+1], i, j);
        }
    }
    wait();
    while (true) {
        ... // No assertion in main loop
        wait();
    }
}

```

```
// Generated Verilog code
always_ff @(posedge clk or negedge nrst) begin
    if (~nrst) begin
        ...
    end else
    begin
        ...
    end
    for (integer i = 0; i < N; i++) begin
        assert property ( req[i] |>= ready[i] );
    end
    for (integer i = 0; i < N; i++) begin
        for (integer j = 0; j < M; j++) begin
            assert property ( req[i][j] |>= ##2 resp[i][M-j+1] );
        end
    end
end
end
```

8.3 Special assertions

There are special assertions mostly intended for tool developers.

- `sct_assert_latch(var [, latch = true])` – assert that given variable, signal or port is latch if second parameter is true (by default), or not latch otherwise. Latch object is defined only at some paths of method process.
- `sct_assert_const(expr)` – check given expression is true in constant propagation analysis
- `sct_assert_level(level)` – check current block level with given one
- `sct_assert_unknown(value)` – check give value is unknown, i.e. not statically evaluated
- `sct_assert_defined(expr)` – check given expression is defined
- `sct_assert_read(expr)` – check given expression is read
- `sct_assert_register(expr)` – check given expression is read before defined
- `sct_assert_array_defined(expr)` – check given expression is array and some element is defined at least on some paths

9 ERROR MESSAGES

9.1 Design correctness checking

SystemC kernel does some design correctness checks during elaboration and simulation. Additional checks required for process function code to comply it with SystemC synthesizable standard. ICSC reports of incorrect or non-synthesizable statements with clear and expressive error messages that reduces designer efforts.

ICSC checks the following violations:

- Non-channel object read before initialization;
- Array out-of-bound access and dangling/null pointer dereference;
- Inter-process communication through non-channel objects;
- Objects not assigned at all control flow paths in combinational methods, that leads to latch;
- Incomplete sensitivity lists for combinational methods;
- Loop without `wait()` with infinite or non-determinable iteration number.

9.2 Error messages

Table 2 contains ICSC error messages.

ID	Error description
SC.1	Default initializer (CXXDefaultInitExpr) cannot be used in process function
SC.2	List initializer (InitListExpr) cannot be used in process function
SC.3	Class constructor (CXXConstructExpr) cannot be used in process function
SC.4	Operator new (CXXNewExpr) cannot be used in process function
SC.5	Operator delete (CXXDeleteExpr) cannot be used in process function
SC.6	Incorrect range, different variables in lo/hi
SC.7	Incorrect range, low index bigger than high or negative one
SC.8	Incorrect range, high index out of variable width
SC.9	Unsupported kind of port
SC.10	Call <code>wait()</code> in method process prohibited
SC.11	Call <code>wait(N)</code> where N is non-constant
SC.12	Call <code>wait(N)</code> where N is negative or zero
SC.13	Port not bound
SC.14	Functions with NORETURN attribute not supported
SC.15	Dangling pointer
SC.16	Multiple pointers point to object
SC.17	Unsupported pointer to element of integer array
CPP.1	Incorrect referenced object in <code>&</code> statement
CPP.2	Complex initialization in FOR loop is not supported
CPP.3	Complex increment in FOR loop is not supported

CPP.4	Array out-of-bound
CPP.5	Goto statement not supported
CPP.6	Unsupported statement
CPP.7	For loop without counter initialization
CPP.8	Pointers to different objects comparison
CPP.9	No virtual function found in the dynamic class
CPP.10	Pure virtual function call
CPP.11	Uninitialized reference
CPP.12	User assertion (sct_assert) failed
CPP.13	OpaqueValueExpr found, check for C++ compiler error message
SYNTH.1	Unsupported to synthesis
SYNTH.2	Multidimensional array initialization not supported yet
SYNTH.3	Initializer not supported to synthesis
SYNTH.4	Type is not supported to synthesis
SYNTH.5	Pointers to dynamically allocated object declared in different modules
SYNTH.6	Pointer initialization with non-zero integer not supported
SYNTH.7	Unsupported wait() kind for synthesis
SYNTH.8	Channel method is not supported
SYNTH.9	Prohibited statement in SC_METHOD with empty sensitivity
SYNTH.10	SC_METHOD is non-sensitive to variable which read inside
SYNTH.11	Variable defined at some paths of method process, latch detected
SYNTH.12	Pointer operation not supported yet
SYNTH.13	Unsupported return in current loop/switch (incorrect loop stack)
SYNTH.14	No array or global array object in [] operator
SYNTH.15	Loop with wait() has potential fallthrough path
SYNTH.16	Signed and unsigned SC type mixed, negative value casted to unsigned
SYNTH.17	Unsupported return in current loop/switch (incorrect loop stack)
SCT.1	ScTool internal error

Table 2. Error messages