Intel Compiler for SystemC User Guide

version 1.6

MIKHAIL MOISEEV, Intel Corporation

Author's address: Mikhail Moiseev, Intel Corporation ,

Contents

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, SingleSource library of communication channels 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 2023,
- 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` and SingleSource `sct_signal`;
- Port - SystemC `sc_in` and `sc_out` and SingleSource `sct_in` and `sct_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 9. 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.

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 9.

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 9.1.

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 22.04 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++20 (for gcc it is version 9.0.0 or later)
- CMake version 3.12 or later
- git to clone ICSC repository

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

```
$ export ICSC_HOME=/home/user/my_iscs_folder
$ export CMAKE_CXX_IMPLICIT_INCLUDE_DIRECTORIES=/usr/include/c++/11
$ git clone https://github.com/intel/systemc-compiler $ICSC_HOME/icsc
```

`CMAKE_CXX_IMPLICIT_INCLUDE_DIRECTORIES` is required if there are multiple C++ compilers or ICSC cannot find C++ standard headers.

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       -- ISCS sources
    * systemc       -- patched SystemC 3.0.0 RC 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_iscs_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 18.1.8 and Clang 18.1.8 from https://releases.llvm.org/download.html#12.0.1 into `$ICSC_HOME“‘` folder.

```
$ cd $ICSC_HOME/build_deps
$ mv clang-18.1.8.src clang
$ mv cmake-18.1.8.src cmake
$ cd llvm-18.1.8.src
$ mkdir build -p && cd build
$ cmake ../ -DLLVM_ENABLE_ASSERTIONS=ON -DLLVM_TARGETS_TO_BUILD=X86
          -DLLVM_ENABLE_PROJECTS=clang -DCMAKE_BUILD_TYPE=Release -G "Unix Makefiles"
          -DCMAKE_INSTALL_PREFIX=$ICSC_HOME -DGCC_INSTALL_PREFIX=$GCC_INSTALL_PREFIX
          -DCMAKE_CXX_STANDARD=17 -DLLVM_INCLUDE_BENCHMARKS=OFF -DLLVM_INCLUDE_TESTS=OFF
$ make -j8
$ make install
```

LLVM_ENABLE_ASSERTIONS – LLVM assertions, can be used in release as well CMAKE_BUILD_TYPE - release or debug, use debug to step into Clang sources (much slower) GCC_INSTALL_PREFIX - use the correspondent STD library from used gcc (13.2.0), the same for clang compiler!!!

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
  * designs           -- folder for examples, tests and user designs
  * icsc              -- ICSC sources
    * build           -- build folder, can be removed
    * cmake           -- CMake files
    * components      -- assertions, fifo and other library components
    * doc             -- user guide latex and pdf files
    * examples        -- a few illustrative examples
    * sc_elab         -- elaborator sources
    * sc_tool         -- ISCS sources
    * systemc         -- patched SystemC 3.0.0 RC 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
  * gdbinit-example.txt -- GDB configuration to be copied into ~/.gdbinit
  * 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 DesignTargetName      # run SV generation for DesignTargetName
```

where `DesignTargetName` is a target name in `CMakeLists.txt`.

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 icsc/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 -p build && cd build
$ cmake ../                       # prepare Makefiles
$ make counter                    # compile SystemC simulation for counter example
$ cd icsc/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. SystemVeriog 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)
```

Tool option	Description
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 SV intrinsic, see 9.1
UNSIGNED	Unsigned mode for designs with unsigned arithmetic only, see 4.7.1
INIT_LOCAL_VARS	Initialize non-initialized process local variables with 0 to avoid latches, that related to CPP data types only, SC data types always initialized with 0
INIT_RESET_LOCAL_VARS	Initialize non-initialized clocked thread local variables declared in reset section with zero, that related to CPP data types only, SC data types always initialized with 0
INIT_LOCAL_REGS	Initialize local variables declared in CTHREAD main loop body in reset section with zero, that related to the variables which become registers
PORT_MAP_GENERATE	Generate port map file and top module wrapper with flatten port arrays, port map file used for SC/SV mixed language simulation, top module wrapper used for logic synthesis tools which do not support unpacked port array in top module interface
NO_SVA_GENERATE	Do not generate SVA from immediate and temporal SystemC assertions, SVA are generated by default
NO_REMOVE_EXTRA_CODE	Do not remove unused variable and unused code, normally such code is removed to improve readability

Table 1. svc_target parameters

4.7 Tool options and defines

ICSC has several options, which can be specified as svc_target parameters. These options given in Table 1.

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.

4.7.1 Unsigned mode. Unsigned mode is intended for designs with unsigned arithmetic only. That means all variables and constants types are unsigned, all expressions are evaluated as non-negative.

In this mode variables and constants types as well as expressions types are checked to be unsigned. C99 types `uint8_t` and `uint16_t` (declared in `<cstdint>`) are not recommended to use in this mode as they leads to false warnings.

Literals could be signed and unsigned (with suffix `U`) in all operations except shifts. In shift expressions both arguments, including literals, should be unsigned.

```
int i;                  // Warning, signed type variable
unsigned u = 1;
unsigned long ul = 2
sc_uint<12> x = 3;
sc_int<12> y;           // Warning, signed type variable
sc_biguint<32> bx = 4;
uint8_t z;
const unsigned N = 42;
ul = u + 1;
ul = 1 << x;            // Warning, signed literal in shift
ul = 1U << x;
ul = x + z;             // False warning for uint8_t
```

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 flatten 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 flatten 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);
    }
};
```

Port can be connected to a signal through module hierarchy. That means the port and signal can belong to different modules, not necessary to child and parent or two children of the same parent.

```
SC_MODULE(A) {
    sc_in<T>    SC_NAMED(i);
};
SC_MODULE(B) {
    A           SC_NAMED(mod_a);
};
SC_MODULE(C) {
    sc_signal<T> SC_NAMED(s);
    B           SC_NAMED(mod_b);
    SC_CTOR(C) {
       mod_b.mod_a.i(s); // Cross-binding of port "i" to signal "s"
    }
};
```

5.3 Modular interface access

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.

5.4 Module constructor and before_end_of_elaboration() callback

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 inside.

Module/modular interface constructor body can contain:

- processes creation with SC_METHOD, SC_THREAD and SC_CTHREAD macros,
- non-SystemC objects allocation in dynamic memory with sc_new operator,
- child modules and other SystemC objects allocation with new operator,
- child module port bindings to port/signals,
- function calls of this module or its child modules.

Another place where module elaboration can be done is before_end_of_elaboration callback. In this call back the same things as in module constructor can be done. Using before_end_of_elaboration callback allows to adjust the module behavior after parameters set up by its function calls from another modules.

Module member constants can be initialized in place, in module constructor initialization list and in before_end_of_elaboration callback. Global and member constants are translated into localparam in SystemVerilog. Member constants of any integral type should be 64 bit or less (stored in uint64/int64 field). Such constants with more than 64 bit has unknown value after elaboration, therefore error is reported. Local constants in process function are translated to SystemVerilog local variables. Such constants can be more than 64bit.

Non-constant module fields also can be initialized in constructor or in place. If a module field has integral type and not modified in any process, it considered as simulation time constant and translated into localparam in SystemVerilog.

```
// Constants initialized in and after constructor
class MyModule : public sc_module {
    const sc_uint<8> A = 0;       // In-place initialization
    const int B[4] = {0,1,2,3};   // In-place initialization
    const bool C;
    const unsigned* D = nullptr;
    int E;

    MyModule(const sc_module_name& name, int par) :
        sc_module(name),
        C(par == 42)              // In initialization list
    {
        setParam(par);
    }

    unsigned d;
```

```
    void setParam(unsigned par) {
       d = par;
    }

    // After constructor initialization
    void before_end_of_elaboration() override {
       D = sc_new<unsigned>(d);
       E = 42;
    }
}
MyModule mod("mod", 12, true);
```

5.5 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 6.8.

```
// 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 6.6.

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 6.7.

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;
}
```

5.6 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 if 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 though 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 6.9.

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

CTOR(MyModule) {
   SC_CTHREAD(test_cthread, clk.pos());
```

```
   async_reset_signal_is(rst, false);
}

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

5.7 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.

Method process can have arbitrary number of resets in its sensitivity list.

```
SC_CTOR(test_reset) {
   SC_METHOD(proc);
   sensitive << sreset << areset << ...;
}
void proc() {
   if (!sreset || !areset) {
       // Reset behavior
   } else {
       // Normal behavior
   }
}
```

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
```

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;
      wait();
   }
}
```

For clocked thread process ICSC generates pair of `always_comb` and `always_ff` blocks as described in 6.9.

There are some limitation to variable use in reset section of clocked thread described in 6.10.

5.8 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 2.

SC/C++ type	SV type	SC synthesizable subset
sct_uint<N>	logic [N]	N is 0...+inf
sct_int<N>	logic signed [N]	N is 0...+inf
sc_uint<N>	logic [N]	N is 1...64
sc_biguint<N>	logic [N]	N is 1...+inf
sc_int<N>	logic signed [N]	N is 1...64
sc_bigint<N>	logic signed [N]	N is 1...+inf
sc_bv<N>	logic [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 2. Data type conversion

It is recommended to use SystemC data types instead of C++ types where it is possible. Using C++ data types make sense for 1bit value – `bool` type and as `for` - loop iterators. Using `sc_bigint<N>`/`sc_biguint<N>` results in more accurate arithmetic than `sct_int<N>`/`sct_uint<N>` and is free from implicit promotion to `int64_t`/`uint64_t`. Drawback of `sc_bigint<N>`/`sc_biguint<N>` is some simulation speed slow down.

There are two special types `sct_uint<N>` and `sct_int<N>` provided. They are bit-accurate integer types which support any number of bits:

- `sct_int<N>` is signed integer with N bits, where N is zero or positive,
- `sct_uint<N>` is unsigned integer with N bits, where N is zero or positive.

`sct_int<N>` and `sct_uint<N>` types automatically substitute `sc_int<N>` or `sct_bigint<N>` and `sct_uint<N>` or `sct_biguint<N>` depends on bit width. These types are implemented in `sct_sel_types.h`.

Zero width integer types `sct_int<0>` and `sct_uint<0>` intended to represent optional variables, signals, ports and record fields. In the generated SV such variables are not declared, assignment to such variable is not generated, using such variable as RValue replaced with `0`. The same works for zero width signals, ports and record fields.

```
template <unsigned N>
struct MyModule : public sc_module {
```

```
  sct_uint<N> optVar;
  sc_in<sct_uint<N>> optInPort{"optInPort"};
  sc_signal<sct_uint<N>> optSig{"optSig"};
  struct Rec {
     sct_uint<N> optField;
  };
}
```

Uninitialized local variable of SystemC types (sc_uint, sc_biguint, sc_int, sc_biguint) 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_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 6.16.

5.9 Literals

SystemC literals are used to initialize local and member variables as well as constant and static constants. ICSC supports numerical literals represented as integer value as well as C string. Numerical literals in C string form support more than 64bit values. Numerical literals can have binary, octal, decimal or hexadecimal radix. Negative literal more than 64bit can have only decimal radix.

```
int a = 42;              // Decimal literal
sct_uint<16> b = 0xF1;   // Hexadecimal literal
sct_int<16> c = -0xF1;   // Hexadecimal literal negative value
sct_uint<65> d = "0x1FFFFFFFFFFFFFFFF"; // Literal >64bit in C string
sct_int<70> e = "-36893488147419103231"; // Negative literal >64bit in C string
```

To initialize a variable with all bits 0 or 1 there are special templates:

- `sct_zeros<N>` is N-bit zeros literal of `sct_uint<N>` type,
- `sct_ones<N>` is N-bit ones literal of `sct_uint<N>` type.

```
// Variable declaration with 0/1 initialization
sct_uint<12> a = sct_zeros<12>; // All zeros
sct_uint<66> b = sct_ones<66>; // All ones
sct_int<67> c = -sct_zeros<65>; // Negative value
auto d       = sct_zeros<90>; // Using auto type deduction
```

Literals are limited with 16384bit width.

5.10 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);
}
```

5.11 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);
  ...
}
```

5.12 Record type

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 members. Record member functions can contain `wait()` calls. Record can have members of C++/SC data types. Record cannot have another record members. Record cannot have signal/port or module/modular interface members. Record cannot have any pointer members. Record can have array members of C++/SC data types. Record cannot have array members of record. Record cannot have non-default copy/move constructors and operators.

Record can have constructor, field in-place initialization and initializer list. Record constructor can contain function calls. Record field in-place initialization and constructor initializer list cannot contain function calls.

Record can have one or multiple base classes. Record base class cannot have constructor body (constructor body should be empty), but can have initialization list and in-place initialization. Virtual functions in records are not supported.

Array of records supported. Array of pointers to record is not supported.

Record reference supported. Pointer to record is not 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;
    }
};
```

Record declaration with and without parameters supported in the following forms:

```
Rec1 r1;              // OK
Rec1 r1();            // OK
Rec1 r1{};            // OK
Rec1 r1 = Rec1{};     // Error
Rec1 r1 = Rec1();     // Error
Rec2 r2(42);          // OK
Rec2 r2{42};          // OK
```

5.12.1 Record type in signals and ports. Signal and port of record type are supported. Record used as signal and ports type should have a constructor without parameters. This constructor is used for the signal/ports initialization. Such a record should have `operator==()` and `operator«(std::ostream)` and `sc_trace()` defined. Signal and ports of record can be read/written and assigned to a record variable.

```
struct SRec {
    int x;
    sc_int<2> y;
    bool operator == (const SRec& other) {
        return (x == other.x && y == other.y);
    }
```

```
};
::std::ostream& operator << (::std::ostream& os, const SRec& s) {
    os << s.x << s.y;
    return os;
}
...
sc_in<SRec>   in{"in"};
sc_out<SRec>  out{"out"};
sc_signal<SRec> s{"s"};
...
SRec r1;        // OK
r1 = in.read(); // OK
r1 = s;         // OK
s = Rec1();     // OK
s.write(r1);    // OK
out = r1;       // OK
out = Rec1{};   // OK
int x = in.read().x; // OK
```

Code generation rules for record described in Section 6.15.

5.13 Union type

Union type not supported.

5.14 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 cast `const_cast` is prohibited in left part of assignment, and ignored elsewhere. Reinterpret and dynamic type casts are not supported.

Type cast can be used to change width or/and signness of the variable, literal or expression. Type cast to change unsigned object to signed is supported in binary, unary and compound operations. In other operations type cast to signed as well as all type casts to unsigned are ignored.

Multiple casts for one object are supported. SystemC type conversion to C++ integer methods `to_int()`, `to_uint()`, `to_long()`, `to_ulong()`, `to_int64()`, `to_uint64()` 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 cast negative value to unsigned is prohibited. Type cast unsigned with set high bit to signed negative is prohibited.

```
unsigned u = 0x1FFFFFFFFUL;
int i = -1;
long l = (int)u;                // Prohibited as result is negative value
unsigned long ul = (unsigned)i; // Prohibited as operand has negative value
```

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

5.15 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
};
```

5.16 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.

5.16.1 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
```

5.16.2 switch. `switch` statement is translated into SystemVerilog `case`, see Section 6.14. `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;
}
```

5.16.3 for. `for` statement is translated into SystemVerilog `for` or into `if` see Section 6.13.

`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; ) {...}
```

5.16.4 while. `while` statement is translated into SystemVerilog `while` or into `if` see Section 6.13. `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()) {...}
```

5.16.5 do...while. `do..while` statement is translated into SystemVerilog `do..while` or into `if` see Section 6.13. `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);
```

5.16.6 break. `break` statement is translated into SystemVerilog `break` or substituted with code after the loop body, see Section 6.13.

5.16.7 continue. `continue` statement is translated into SystemVerilog `continue` or substituted with code in the loop body, see Section 6.13.

5.16.8 goto. Not supported.

5.17 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.

5.18 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_MODULE_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 channel arrays
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
// Syncrhonous 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

// Syncrhonous 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 Non-modified member variables generation

Module or modular interface member variables which are non-modified in any process, considered as initialized at elaboration phase. Because of dynamic elaborator used in the tool, it is not possible to detect if a variable is not initialized at elaboration phase.

For such member variables which are scalar (non-array) and non-record type, SV local parameters are generated. No local parameters generated for array and record members, so these variables remain non-initialized. Such member variable should be initialized in reset section of a process where it is used.

```
class MyModule : public sc_module {
        bool C = true;
        int D;
        unsigned E;

        SC_CTOR (MyModule) {
                D = 42;
        }
        void setE(unsigned par) { E = par; }
}
...
MyModule m{"m"};
...
m.setE(43);  // In parent module constructor
```

```
// Generated SystemVerilog
localparam logic C = 1;
localparam logic signed [31:0] D = 42;
localparam logic [31:0] E = 43;
```

Nothing is generated if the variable is not used in any process. If such variable is member of array of modular interfaces, it should be used in a process of each modular interface.

6.4 Constants generation

For constants and static constants SV local parameters are 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;
```

There is `REPLACE_CONST_VARIABLES` option to replace the constants in the code with their values, see 4.7. A 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.

6.5 Integer variable initialization from string

Integer variable of SC type (`sc_uint`, `sc_int`, `sc_biguint`, `sc_bigint`) can be initialized from string literal or string variable. The string should contain correct integer value in any radix supported. For `sc_biguint` and `sc_bigint` initialization value could be more than 64 bits. String variable should be initialized at elaboration phase (in a module constructor).

```
const char* cstr = "43";
std::string str;                // Initialized in constructor or a method call

void someProc() {
    sc_uint<16> ux = "0b110011"; // Initialization with string literal
    ux = "42";                  // String literal assignment
    ux = sc_uint<5>("0x42");    // Cast at initialization supported
    ux = cstr;                  // C string assignment
    ux = str.c_str();           // std::string assignment
    sc_biguint<65> bu;
    bu = "0x1FFFF1111FFFF1111"; // More than 64bit literal
}
```

6.6 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
```

6.7 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;
```

6.8 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 **??**. 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
```

6.9 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 6.12.

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

// Syncrhonous 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
```

6.10 Register variables in thread reset section

There are some limitations to use register variables in reset section of clocked thread. Register variable is a variable which keeps its values between thread states. Register variable `i` is represented with a pair of variables `i` and `i_next` in SV code. Register next to current value has non-blocking assignment in `always_ff` block, therefore such variable cannot have blocking assignment in reset section. For operations which modify register variable in reset section error is reported:

```
void proc() {
    int i = 1;     // Register variable
    i++;           // Error reported
    i += 1;        // Error reported
    wait();
    while (true) {
       out = i;    // Value from reset used here
       wait();
    }
```

```
logic signed [31:0] i;
logic signed [31:0] i_next;
always_ff @(posedge clk or negedge arstn)
begin : read_modify_reg_in_reset_ff
    if ( ~arstn ) begin
        i <= 1;
```

```
        i++;        // Blocking assignment
        i = i + 2;
    end
    else begin
        i <= i_next;
    end
```

Another problem with register variable is reading it in reset section. As soon as the variables has non-blocking assignment it values could be incorrect in RHS of the following statements. For operations which read register variable in reset section warning is reported:

```
void proc() {
    int i = 1;     // Register variable
    int j = i;     // Warning reported
    i = j;         // X results in SV simulation
    wait();
    while (true) {
       out = i;    // Value from reset used here
       wait();
    }
```

6.11 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
endfunction

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

6.12 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;
                waitn_PROC_STATE_next = 1; return;
            end;
```

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

6.13 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
        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
```

6.14 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
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 `swtich` 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
```

6.15 Records generation

Records can be used as member and local variables. Record can be 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
```

Record variable can be assigned to another variable, references to records are supported.

```
// Local variables record type
struct Rec1 {
    int x;
    sc_int<2> y;
};
void record_assignment() {
    Rec1 r; Rec1 p;
    r = p;
    Rec1& ref = r;
    r = p;
}
```

```
always_comb
begin : record_assignment  // test_chan_type_meth.cpp:89:5
    integer r_x;
    logic signed [1:0] r_y;
    integer p_x;
    logic signed [1:0] p_y;
    r_y = 0;
    p_y = 0;
    r_x = p_x; r_y = p_y;
    r_x = p_x; r_y = p_y;
end
```

Records can be used as signal/port type. Such record should have `operator==()`, `operator«(std::ostream)` and `sc_trace()` implemented.

```
// Local variables record type
struct Rec1 {
    int x;
    sc_int<2> y;
    bool operator == (const Rec1& other) {
```

```
        return (x == other.x && y == other.y);
    }
};
inline ::std::ostream& operator << (::std::ostream& os, const Rec1& s) {
    os << s.x << s.y;
    return os;
}
namespace sc_core {
void sc_trace(sc_trace_file* , const Rec1&, const std::string&) {}
}

sc_signal<Rec1> sig{"sig"};
void record_channels() {
    Rec1 r;
    r = sig.read();
    r = sig;
    sig.write(r);
    sig = r;
    int x = sig.read().x;
}
```

```
always_comb
begin : record_channels  // test_chan_type_meth.cpp:97:5
    integer r_x;
    logic signed [1:0] r_y;
    integer x;
    r_y = 0;
    r_x = sig_x; r_y = sig_y;
    r_x = sig_x; r_y = sig_y;
    sig_x = r_x; sig_y = r_y;
    sig_x = r_x; sig_y = r_y;
    x = sig_x;
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
```

6.16 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.

6.16.1 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.

6.16.2 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 unsinged `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.

6.16.3 Unsigned operation overflow. Operations with SC unsigned type arguments (`sc_uint` and `sc_biguint`) can lead to overflow. Using such expressions as part of another expression can lead to non-equivalent simulation results. It is recommended to explicitly cast such expressions to desired width. Left shift for C++ data types is cyclic which is not supported in the generated SV code. Therefore left shift overflow for C++ data type is prohibited.

```
sc_uint<7> k, m;
k = 41; m = 42;
sc_uint<8> res = (k - m) % 11; // Non-equivalent
res = sc_uint<8>(k - m) % 11; // OK
unsigned u = 1;
res = u << 32;                // Non-equivalent: 1 in SC, 0 in SV
```

6.16.4 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));
```

6.16.5 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
```

6.17 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 SINGLESOURCE LIBRARY

7.1 Introduction

SingleSource library consists of communication channels which implements SC interfaces. There are nine main channels in the library (Fig. 1).

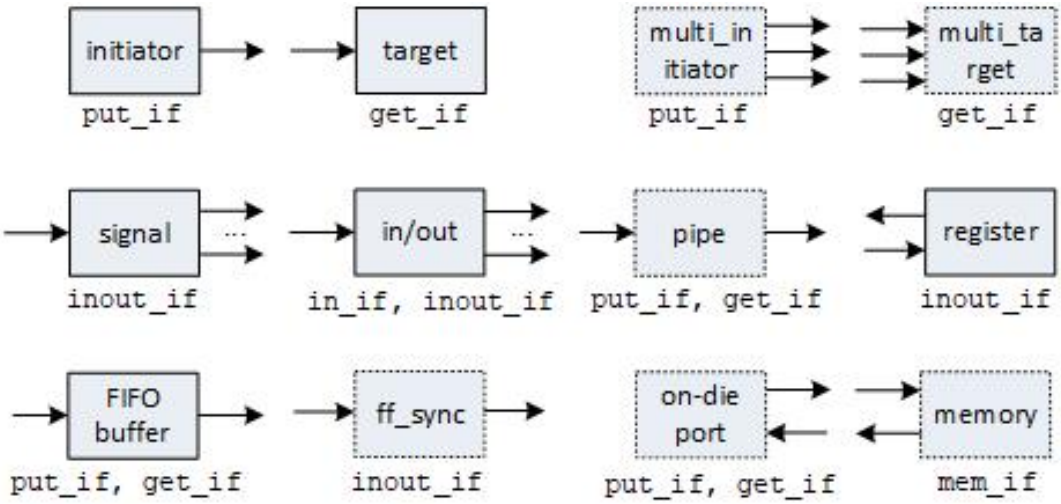

Fig. 1. SingleSource channels

Target and Initiator are intended to connect two SC modules with 1:1 connection. Multi-target and Multi-initiator modules provides 1:N connection to connect multiple SC modules. FIFO is intended to connect two processes in the same module or to serve as a buffer for one process. On-die port represents any external port to connect SC design to other IPs or fabric. Memory represents any kinds of on-chip SRAM, RF or ROM memory. Register is used to add state for METHOD process. The common use cases of the modules are given in the picture below.

Multi-initiator, multi-target, FF synchronizer, on-die port, and memory are not open-sourced yet.

The Single Source modules work in two modes: cycle accurate (RTL) and approximate time (TLM). Cycle accurate (RTL) mode intended for hardware synthesis. In RTL mode the modules provide cycle accurate simulation. Approximate time (TLM, Transaction Level Modeling) mode provide fast simulation, intended for virtual prototyping. In TLM mode the modules provide approximate time simulation, there is no clock. Simulation is request-driven, executed in ordered delta-cycles (DC).

7.1.1 Library files.

- sct_common.h – includes of all library headers and adds namespace **sct**
- sct_ipc_if.h – interfaces, general template types and defines
- sct_initiator.h – initiator module
- sct_target.h – target and combinational target modules
- sct_multi_initiator.h – initiator to connect multiple targets
- sct_multi_target.h – target to connect multiple initiators

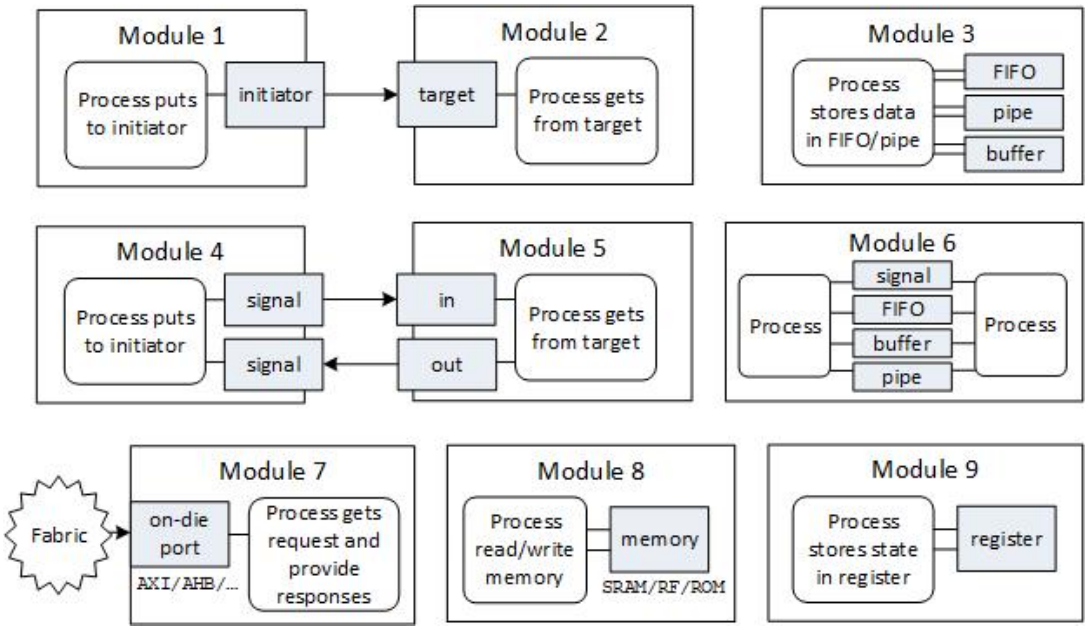

Fig. 2. SingleSource channels usage

- sct_prim_signal.h – primitive channel signal implementation with multiple drivers support
- sct_signal.h – signal implementation
- sct_ports.h – input and output ports, sc_port for target and initiator
- sct_fifo.h – FIFO module
- sct_prim_fifo.h – primitive channel FIFO implementation, used as base channel in TLM mode
- sct_buffer.h – buffer channel, which is a fast FIFO to use in clocked process(es) only
- sct_pipe.h – pipeline register
- sct_register.h – register to store METHOD state
- sct_clock.h – clock with enable/disable
- sct_clk_gate_cell.h – clock gate and clock gate signal
- sct_ff_sync_cell.h – Flip-Flop synchronizer
- sct_ff_sync_cell.sv – Flip-Flop synchronizer RTL implementation

7.1.2 Library defines. `SCT_TLM_MODE` could be provided as compile definition: if `SCT_TLM_MODE` defined TLM mode is used, RTL mode is used otherwise.

There are multiple options for clock/reset levels:

- `SCT_CMN_TRAITS` – clock edge and reset level, one of six following options:
- `SCT_POSEDGE_NEGRESET` – positive clock edge, negative reset level
- `SCT_POSEDGE_POSRESET` – positive clock edge, positive reset level
- `SCT_NEGEDGE_NEGRESET` – negative clock edge, negative reset level
- `SCT_NEGEDGE_POSRESET` – negative clock edge, positive reset level
- `SCT_BOTHEDGE_NEGRESET` – both clock edges, negative reset level

- SCT_BOTHEDGE_POSRESET – both clock edges, positive reset level

Usually, positive clock edge and negative reset level are used. That is provided by define SCT_CMN_TRAITS:

```
#ifndef SCT_CMN_TRAITS
  #define SCT_CMN_TRAITS SCT_POSEDGE_NEGRESET
#endif
```

If other clock edge/reset levels required, SCT_CMN_TRAITS value should be provided as compile definition.

There is an CMakeLists.txt example where sct_def_traits target has definitions for TLM mode, negative clock edge and positive reset level:

```
add_executable(sct_def_traits sc_main.cpp)
target_compile_definitions(sct_def_traits PUBLIC -DSCT_TLM_MODE)
target_compile_definitions(sct_def_traits PUBLIC -DSCT_CMN_TRAITS=SCT_NEGEDGE_POSRESET)
```

7.2 Library interfaces

The interfaces contain non-blocking functions except b_put and b_get which are may-blocking.

Interface sct_put_if:

- bool ready() – Return true if it is ready to put request,
- void reset_put() – Reset this initiator/FIFO,
- void clear_put() – Clear (remove) request put in this cycle,
- bool put(const T& data) – Non-blocking put request into initiator/FIFO if it is ready, return ready to request,
- bool put(const T& data, sc_uint<N> mask) – Non-blocking put request into initiator/FIFO if it is ready, mask used to enable/disable put or choose targets in multi-cast put, return ready to request,
- void b_put(const T& data) – May-blocking put request, could be used in THREAD process only,
- void addTo(sc_sensitive& s) – Add put related signals to process sensitivity,
- void addTo(sc_sensitive* s, sc_process_handle* p) – Add put related signals to process sensitivity.

Interface sct_get_if:

- bool request() – Return true if it has request to get,
- void reset_get() – Reset this target/FIFO,
- void clear_get() – Clear (return back) request got in this cycle,
- T peek() – Peek request, return current request data, if no request last data returned,
- T get() – Non-blocking get request and remove it from FIFO/target, return current request data, if no request last data returned,
- bool get(T& data, bool enable) – Non-blocking get request and remove it from FIFO/target if enable is true, return true if there is a request and enable is true,
- T b_get() – May-blocking get request, could be used in THREAD process only,

- `void addTo(sc_sensitive& s)` – Add get related signals to process sensitivity,
- `void addTo(sc_sensitive* s, sc_process_handle* p)` – Add get related signals to process sensitivity,
- `void addPeekTo(sc_sensitive& s)` – Add peek related signal to process sensitivity.

Interface `sct_fifo_if` inherits sct_put_if<T> and sct_get_if<T> and has the following additional functions:

- `unsigned size()` – FIFO size,
- `unsigned elem_num()` – Number of elements in FIFO, value updated last clock edge for METHOD, last DC for THREAD,
- `bool almost_full(const unsigned& N)` – Return true if FIFO has (LENGTH-N) elements or more, value updated last clock edge for METHOD, last DC for THREAD,
- `void clk_nrst(sc_in<bool>& clk_in, sc_in<bool>& nrst_in)` – Bind clock and reset to FIFO,
- `void addTo(sc_sensitive& s)` – Add put and get related signal to process sensitivity,
- `void addToPut(sc_sensitive& s)` – Add put related signals to process sensitivity,
- `void addToGet(sc_sensitive& s)` – Add get related signals to process sensitivity.

Interface `sct_in_if`:

- `const T& read()` – Read from signal/register,
- `void addTo(sc_sensitive* s, sc_process_handle* p)` – Add signals to process sensitivity.

Interface `sct_inout_if`:

- `const T& read()` – Read from signal/register,
- `void write(const T& val)` – Write to signal/register,
- `void addTo(sc_sensitive* s, sc_process_handle* p)` – Add signals to process sensitivity.

Functions addTo, addToPut, addToGet and addPeekTo are used to add the channel to process sensitivity list. For target and initiator instead addTo `operator «` can be used. For FIFO instead addToPut and addToGet operator `« fifo.PUT`, `« fifo.GET`, and `« fifo.PEEK` can be used.

7.3 Processes

SystemC design with single source library can use method and thread processes created with `SC_METHOD` and `SC_THREAD` correspondently. It is recommended to use `SCT_METHOD` and `SCT_THREAD` macros instead of them:

- `SCT_METHOD(proc)` – creates combinational method process,
- `SCT_METHOD(proc, clk)` and `SCT_METHOD(proc, clk, rst)` – create sequential method process,
- `SCT_THREAD(proc, clk)` and `SCT_THREAD(proc, clk, rst)` – create sequential thread process.

Clocked thread process created with `SC_CTHREAD` is normally not used.

Difference between sequential method and thread processes is in simulation speed, method process is faster. Thread process allows to use `wait()` to introduce multiple states, which can simplify the process function code. Sequential method process function should have reset and body sections, and cannot have `wait()` calls.

```
void combMethod() {
   // combinational logic ...
}
void seqMethod() {
   if (rst) {
      // reset logic ...
   } else {
      // sequential logic ...
   }
}
void seqThread() {
   // reset logic
   wait();
   while (true)
      // sequential logic ...
      wait();
      // multiple wait() calls allowed
   }
}
```

All the channels used in a thread process should use the same clock and edge as the process sensitive. A channel could have different reset or reset level than thread process. If all the channels have different reset or reset level, process reset is specified with third parameter of SCT_THREAD macro.

All the channels used in a thread process should use the same clock and edge as the process sensitive. A channel could have different reset or reset level than thread process. If all the channels have different reset or reset level, process reset is specified with third parameter of SCT_THREAD macro.

If a thread process has reset signal, it should have the reset specification with async_reset_signal_is or/and sync_reset_signal_is. A thread process should be sensitive to all single source channels accessed in its function code. If process is sensitive only to signal and input/output ports, the clock is provided with second parameter of SCT_THREAD macro.

Method process should be sensitive to all single source channels accessed in its function code. Method should not be sensitive to reset, as its provided implicitly through the channels used.

```
template <class T>
class MyModule : public sc_module {
   sc_in<bool>    clk{"clk"};
   sct_target<T>  targ{"targ"};
   sct_initiator<T> init{"init"};
   sct_signal<T>  s{"s"};

   explicit MyModule(const sc_module_name& name) : sc_module(name) {
       SC_THREAD(seqThread);
       sensitive << targ;                      // Clock and reset taken from @targ
       async_reset_signal_is(nrst, 0);         // Reset specification required
```

```
        SCT_THREAD(seqThread, clk, nrst);       // Sequential thread, clock and reset
            explicitly provided
        sensitive << targ;
        async_reset_signal_is(nrst, 0);         // Reset specification required

        SCT_METHOD(combMethod);                 // Combinational method
        sensitive << init;                      // No reset in sensitivity

        SCT_METHOD(seqMethod, clk, nrst);       // Sequential method, clock and reset
            explicitly provided
        sensitive << init;
    }
};
```

If any process sensitive to a channel which is not read inside or not sensitive to a channel which is read inside, error reported by ICSC. The error is reported for single channels and for vector/array of channels, no individual channels in vector/array are considered here.

7.4 Target and Initiator

Target and Initiator are channels intended to connect two user defined modules. Initiator implements `sct_put_if` interface and could be used in one METHOD or THREAD process to put requests. Target implements `sct_get_if` interface and could be used in one METHOD or THREAD process to get requests which put by the connected Initiator.

To connect two modules, Target placed in one modules, Initiator in another one. Target and Initiator should be connected to clock and reset with `clk_nrst()` function. Target and Initiator are connected to each other with method `bind()`, called in their common parent module constructor. Both Target and Initiator have method `bind()`, any of them can be called.

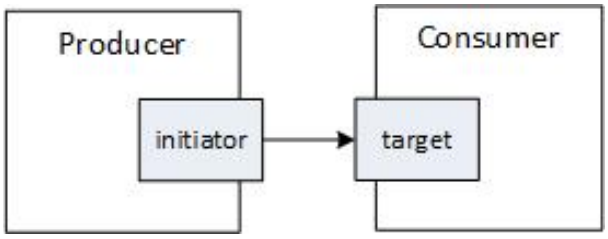

Fig. 3. Target and Initiator channels

```
struct Producer : public sc_module {
    sc_in<bool>       clk{"clk"};
    sc_in<bool>       nrst{"nrst"};
    sct_initiator<T>  init{"init"};
    explicit Producer (const sc_module_name& name) : sc_module(name) {
        init.clk_nrst(clk, nrst);
```

```
    }
}
struct Consumer : public sc_module {
    sc_in<bool>       clk{"clk"};
    sc_in<bool>       nrst{"nrst"};
    sct_target<T>     targ{"targ"};
    explicit Consumer (const sc_module_name& name) : sc_module(name) {
        targ.clk_nrst(clk, nrst);
    }
}
struct Top: public sc_module {
    Producer prod{"prod"};
    Consumer cons{"cons"};
    explicit Top(const sc_module_name& name) : sc_module(name) {
        prod.clk(clk); prod.nrst(nrst);
        cons.clk(clk); cons.nrst(nrst);
        // Call bind() method of initiator or bind() method of target
        prod.init.bind(cons.targ);
    }
}
```

Target and Initiator have the same template parameters:

```
template<
    class T,                          // Payload data type
    class TRAITS = SCT_CMN_TRAITS,    // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE> // RTL (0) or TLM (1) mode
class sct_initiator {};

template<
    class T,                          // Payload data type
    class TRAITS = SCT_CMN_TRAITS,    // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE> // RTL (0) or TLM (1) mode
class sct_target {};
```

Target and Initiator constructor parameters:

```
sct_target(const sc_module_name& name,
          bool sync_ = 0,             // Is register required to pipeline request
          bool always_ready_ = 0);    // Is always ready to get request

sct_initiator(const sc_module_name& name,
          bool sync_ = 0);            // Is register required to pipeline request
```

7.5 Target and initiator usage

Target and initiator can be used in SystemC method process. The method process should be created with SC_METHOD or SCT_METHOD macro in the module constructor. The method process should have sensitivity list with all the targets/initiators accessed in the process function.

```
// Initiator and target in method process example
struct Producer : public sc_module {
    sct_initiator<T>      init{"init"};
    explicit Producer (const sc_module_name& name) : sc_module(name) {
       SC_METHOD(initProc);
       sensitive << init;
    }
    void initProc {
       // Put data into init
    }
}

struct Consumer : public sc_module {
    sct_target<T>     targ{"targ"};
    explicit Consumer (const sc_module_name& name) : sc_module(name) {
       SC_METHOD(targProc);
       sensitive << targ;
    }
    void targProc{
       // Get data from targ
    }
}
```

Target and initiator can be used in clocked thread process. Clocked thread process should be created with SC_THREAD or SCT_THREAD macro, but not with SC_CTHREAD. The thread process should have sensitivity list with all the targets/initiators accessed in the process function as for method process. If the thread process has reset signal, it should have the reset specification with async_reset_signal_is or/and sync_reset_signal_is.

```
// Initiator and target in thread process example
struct Producer : public sc_module {
    sct_initiator<T>      init{"init"};
    explicit Producer (const sc_module_name& name) : sc_module(name) {
       SC_THREAD(initProc);
       sensitive << init;
       async_reset_signal_is(nrst, 0);
    }
    void initProc {
       // Reset init to set default values
       wait();
```

```
      while(true) {
         // Put data into init
         wait();
      }
   }
}

struct Consumer : public sc_module {
   sct_target<T>     targ{"targ"};
   explicit Consumer (const sc_module_name& name) : sc_module(name) {
      SC_THREAD(targProc);
      sensitive << targ;
      async_reset_signal_is(nrst, 0);
   }
   void targProc{
      // Reset init to set default values
      wait();
      while(true) {
         // Get data from targ
         wait();
      }
   }
}
```

There are three kinds of connections which could be organized:

- Combinational,
- Buffered,
- Buffered with FIFO.

7.5.1 Combinational connection. In combinational connection request part of connection contains `core_req` and `core_data` signals, which could be used directly or through the pipelining register (specified with second parameter of Target/Initiator constructor). There is no back-pressure signal, so Target process should be always ready to get request. Initiator process does not need to check ready to put request (method `ready()` always returns true).

Combinational connection is provided with last parameter of `sct_target<>` constructor or with using special target class `sct_comb_target<>`.

In combinational connection put data into Initiator can be done without checking if the Initiator is ready.

```
// Initiator and always ready target in method process example
struct Producer : public sc_module {
   sct_initiator<T>      init{"init"};
   explicit Producer (const sc_module_name& name) : sc_module(name) {
      SC_METHOD(initProc); sensitive << init;
   }
```

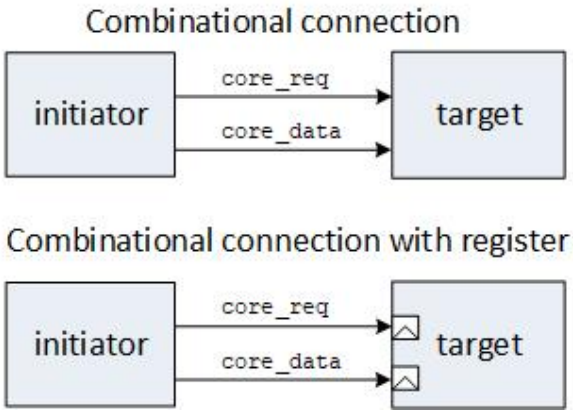

Fig. 4. Combinational connection

```
    void initProc {
       T val = getSomeValue();       // Put at every path, reset is not required
       init.put(val);                // Do not check ready() as Target is always ready
    }
}

struct Consumer : public sc_module {
    // Combinational target
    sct_comb_target<T>     targ{"targ"};
    explicit Consumer (const sc_module_name& name) : sc_module(name) {
       SC_METHOD(targProc); sensitive << targ;
    }
    void targProc{
       T val;
       if (targ.get(val)) {          // Get at every path, reset is not required
           doSomething(val);
       }
    }
}
```

In thread process it needs to reset Initiator and Target in the reset section.

```
// Initiator and always ready target in thread process example
struct Producer : public sc_module {
    sct_initiator<T>       init{"init"};
    explicit Producer (const sc_module_name& name) : sc_module(name) {
       SC_THREAD(initProc); sensitive << init;
       async_reset_signal_is(nrst, 0);
    }
    void initProc {
```

```
      init.reset_put();             // Reset is required in thread process
      wait();
      while(true) {
         T val = getSomeValue();    // Put every cycle
         init.put(val);             // Do not check ready() as Target is always ready
         wait();
      }
   }
}

struct Consumer : public sc_module {
   sct_comb_target<T>    targ{"targ"};
   explicit Consumer (const sc_module_name& name) : sc_module(name) {
      SC_THREAD(targProc); sensitive << targ;
      async_reset_signal_is(nrst, 0);
   }
   void targProc{
      targ.reset_get();             // Reset is required in thread process
      wait();
      while(true) {
         if (targ.request()) {
             doSomething(targ.get());
         }
         wait();
      }
   }
}
```

Using Target and Initiator in method and thread process looks very similar. In the next sections examples using method and thread process will be mixed.

7.5.2 Buffered connection. In buffered connection `core_ready` signal is used as backpressure when Target is not ready to get request. This connection called buffered as it has the buffer register inside Target or Initiator to store one request if Target is not ready. This kind of connection is the most common and used as default one.

Request part of the connection contains `core_req` and `core_data` signals, which could be used directly or through the pipelining register (specified with second parameter of Target/Initiator constructor). The pipelining register is additional to the buffer register. Response part contains `core_ready` signal which is passed through register to avoid combinational loop. If target process is method this register explicitly added, if it is thread this register is implicitly provided by the process.

```
struct Producer : public sc_module {
   sct_initiator<T>      init{"init"};
   explicit Producer (const sc_module_name& name) : sc_module(name) {
      SC_METHOD(initProc); sensitive << init;
   }
```

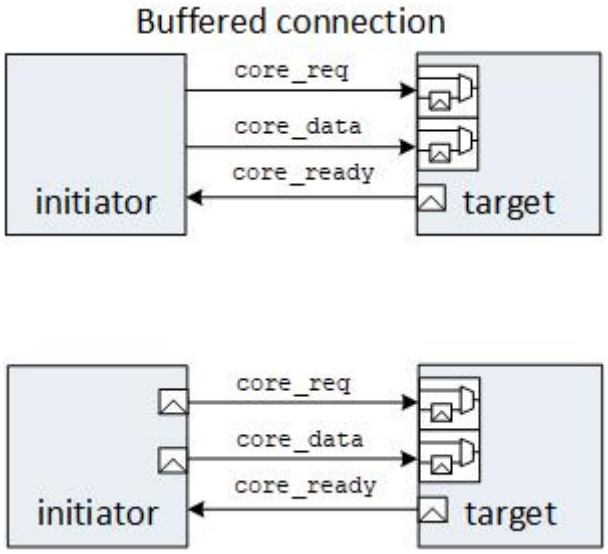

Fig. 5. Buffered connection

```
    void initProc {
       init.reset_put();              // Reset required as put done at some paths only
       if (init.ready()) {            // Check ready required, target can be not ready
          init.put(getSomeValue());
       }
    }
}

struct Consumer : public sc_module {
    sct_target<T>     targ{"targ"};
    explicit Consumer (const sc_module_name& name) : sc_module(name) {
       SC_THREAD(targProc); sensitive << targ;
       async_reset_signal_is(nrst, 0);
    }
    void targProc {
       targ.reset_get();
       wait();
       while(true) {
          if (targ.request()) {
              doSomething(targ.get());
          }
          wait();
       }
    }
}
```

7.5.3 Buffered connection with FIFO. The buffered connection with FIFO provides additional buffer to store requests until their processed by the target process. FIFO can be added to Target with `add_fifo()` method:

```
template<unsigned LENGTH>              // FIFO size (maximal number of elements)
void add_fifo(bool sync_valid = 0,     // Is register required to pipeline request
         bool sync_ready = 0,          // Is register required to pipeline ready
         bool init_buffer = 0);        // Initialize all the elements with zeros
                                       // First element to get is always initialized
```

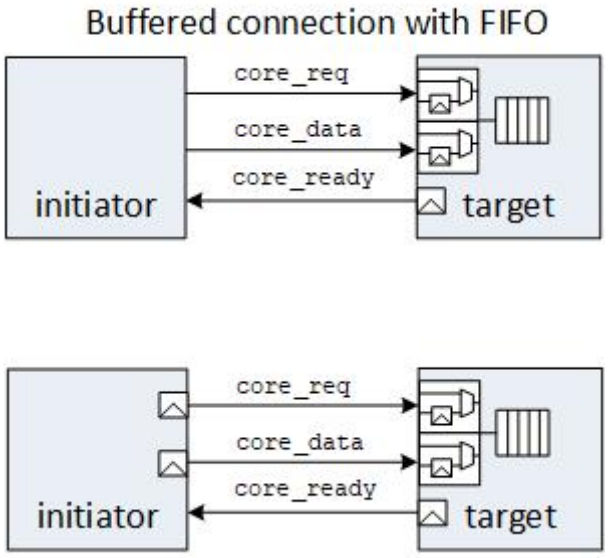

Fig. 6. Buffered connection with FIFO

```
template<class T>
struct A : public sc_module {
    sct_target<T>     run{"run"};
    explicit A(const sc_module_name& name) : sc_module(name) {
        run.clk_nrst(clk, nrst);
        run.template add_fifo<2>(1, 1); // Add FIFO with 2 element and registers
    }
}
```

7.5.4 Initiator-to-Target Protocol. The discussed protocol considers buffered connection w/o FIFO. Request is taken by Target when `core_req` and `core_ready` both are high. Target can return it to the target process immediately or store the request in the buffer. Initiator sets new request when the previous one has been taken.

The first diagram below represents Target and Initiators accessed in thread processes. The second diagram represents Target and Initiators accessed in method processes.

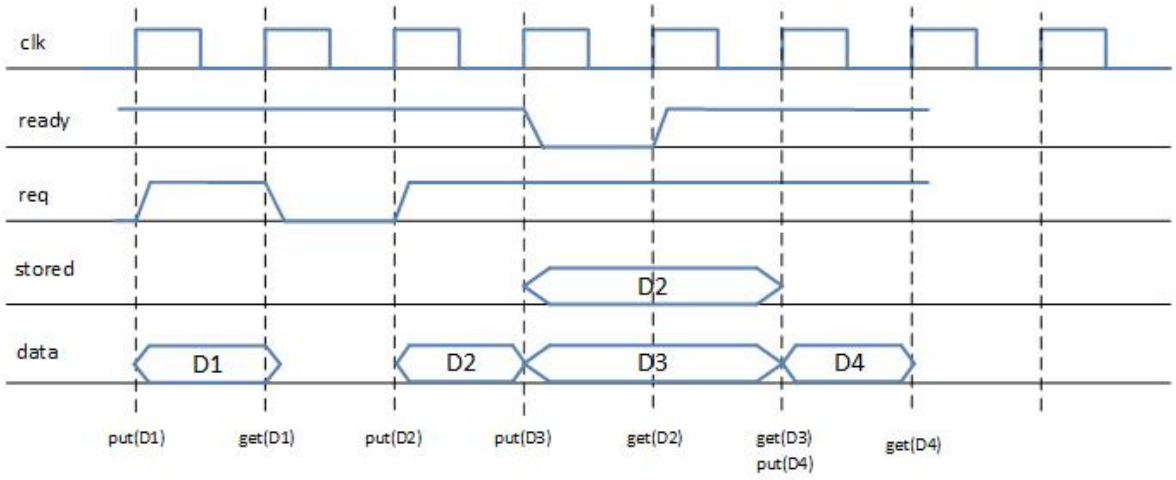

Fig. 7. Initiator-to-Target in THREAD processes

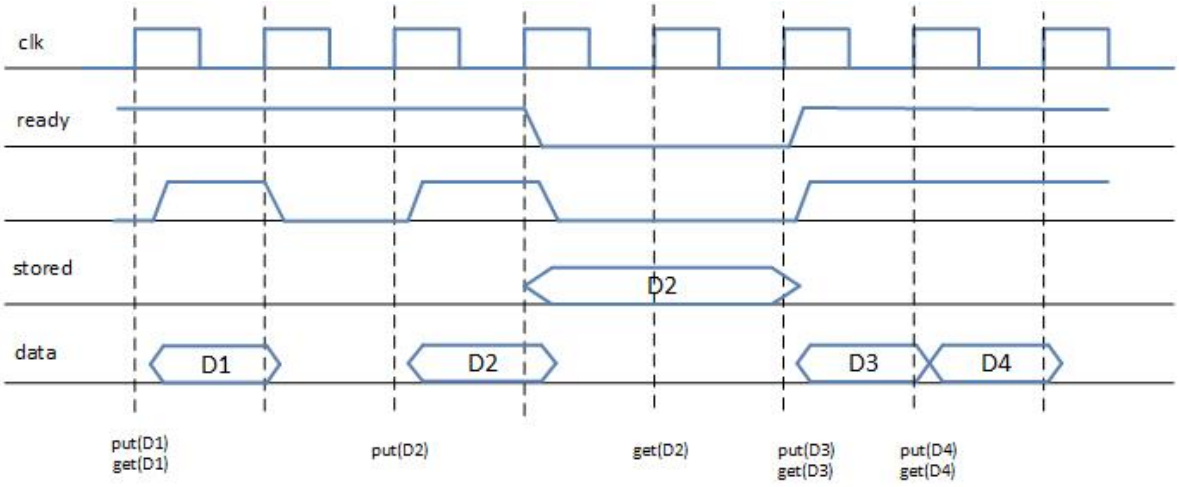

Fig. 8. Initiator-to-Target in METHOD processes

7.6 Signal and ports

Signal can be used for inter-process communication between processes in the same module. For communication between processes in different modules input/output ports are used together with signal.

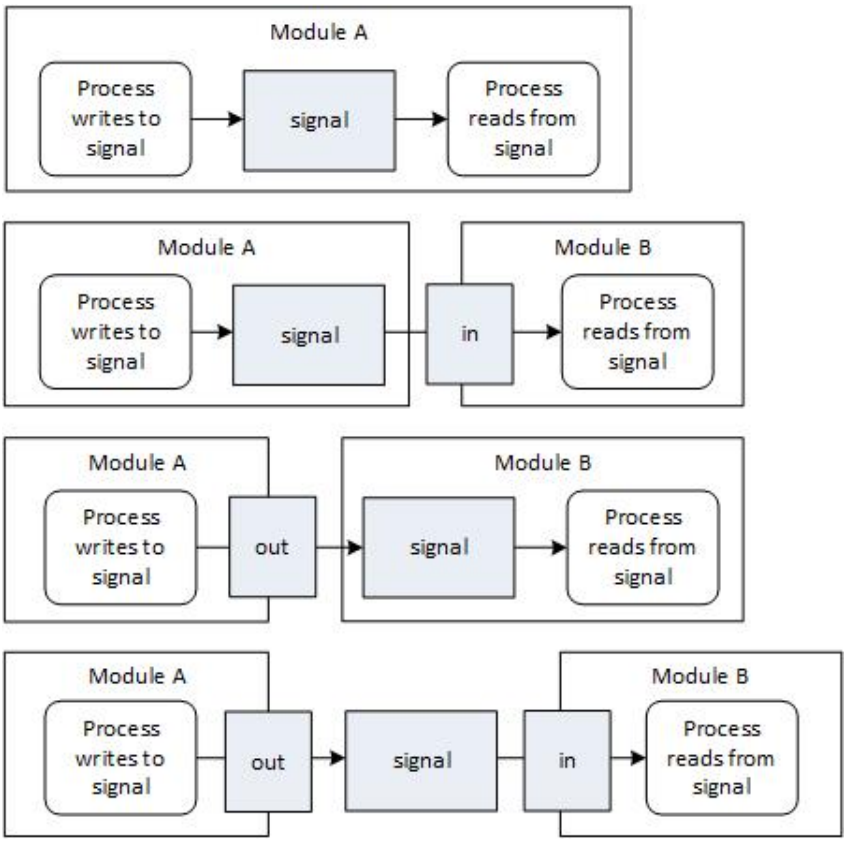

Fig. 9. Signal and ports

Signal and output port implement `sct_inout_if`, and can be written by one process. Signal, input and output ports implement `sct_in_if`, and can be read by one or more processes.

```
template<
    class T, bool TLM_MODE = SCT_CMN_TLM_MODE>
class sct_signal {};
template<
    class T, bool TLM_MODE = SCT_CMN_TLM_MODE>
class sct_in {};
template<
    class T, bool TLM_MODE = SCT_CMN_TLM_MODE>
class sct_out {};
```

Using signal and input/output ports in thread process requires to have clock/reset for these channels which provided with `SCT_THREAD` macro:

```
// Used if the process sensitive to signals/ports only
```

```
SCT_THREAD(proc, clk, rst);
// Used if the process sensitive to signals/ports and other channels
SCT_THREAD(proc, clk);
```

In this example sigThread sensitive to signals only:

```
sct_signal<T> s{"s"};
MyModule(const sc_module_name& name) : sc_module(name) {
   // Clock edge/reset level taken from SCT_CMN_TRAITS
   SCT_THREAD(sigThread, clk, nrst);
   // Only signal `s` is read inside the process
   sensitive << s;
   async_reset_signal_is(nrst, 0);
}
```

sc_vector of sct_signal, sct_in and sct_out supported. Binding of while vector to another vector is supported.

```
class A : public sc_module {
    sc_vector<sct_out<T>>   resp{"resp", 3};
};
class Top {
    A  a{"a"};
    sc_vector<sct_signal<T>> resp{"resp", 3};

    Top (const sc_module_name& name) : sc_module(name) {
        a.resp(resp);               // All vector elements bound
    }
}
```

In RTL mode sct_signal is based on sc_signal, sct_in/sct_out are based on sc_in/sc_out.

7.7 FIFO

The FIFO can be used for inter-process communication between processes in the same module and for storing requests inside one process. Also the FIFO could be used inside of Target as an extended buffer.

FIFO implements sct_fifo_if interface. FIFO has size template parameter which is a positive number.

```
template<
    class T,
    unsigned LENGTH,                    // Size (maximal number of elements)
    class TRAITS = SCT_CMN_TRAITS,      // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE> // RTL (0) or TLM (1) mode
>
class sct_fifo {};
```

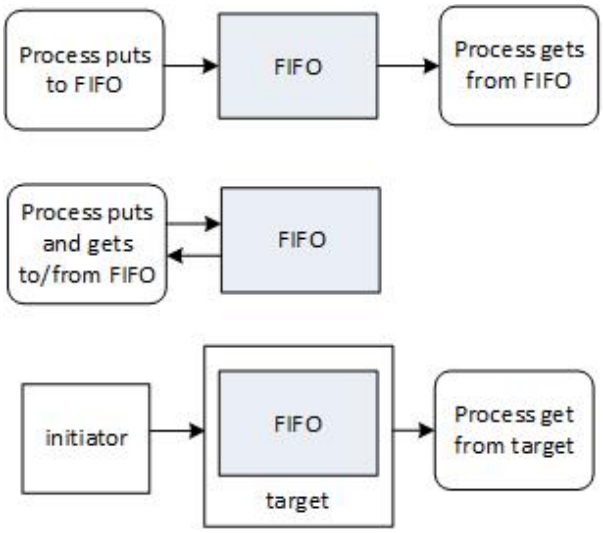

Fig. 10. FIFO

FIFO can have combinational or registered request (`core_req` and `core_data`) and response (`core_ready`) kind which specified in constructor parameters.

```
sct_fifo(const sc_module_name& name,
        bool sync_valid = 0,         // Request path has synchronous register
        bool sync_ready = 0,         // Response path has synchronous register
        bool use_elem_num = 0,       // Element number/Almost full or empty used
        bool init_buffer = 0)        // Initialize all buffer elements with zeros in reset
                                     // First element to get is always initialized to zero
```

7.7.1 Minimal FIFO size required. Minimal FIFO size required given in Table. 3.

Using FIFO in method process(es) with both `sync_valid` and `sync_ready` set to 0 is prohibited as that results in combinational loop. Using FIFO in one method process is allowed with `sync_valid` and `sync_ready` both set to 1 only. If `sync_valid` or `sync_ready` set to 0, such FIFO can be used in two different method processes.

7.7.2 Using FIFO for inter-process communication. FIFO could be used for processes communication instead of set of signals. FIFO has only one writer and one reader process, in comparison with `sct_signal` which could be read in multiple processes. For 1:N communication array or `sc_vector` of FIFOs could be used.

```
struct Top : public sc_module {
    sct_fifo<T, 2>    fifo{"fifo", 1};   // Pipelining register for request
    explicit Top(const sc_module_name& name) : sc_module(name) {
        fifo.clk_nrst(clk, nrst);
        SC_THREAD(producerProc);
```

Initiator process	Target process	sync_valid	sync_ready	Minimal FIFO size
method	method	0	0	prohibited
method	method	0	1	1, two processes
method	method	1	0	1, two processes
method	method	1	1	2
method	thread	0	0	1
method	thread	0	1	2
method	thread	1	0	2
method	thread	1	1	3
thread	method	0	0	1
thread	method	0	1	2
thread	method	1	0	2
thread	method	1	1	3
thread	thread	0	0	2
thread	thread	0	1	3
thread	thread	1	0	3
thread	thread	1	1	4

Table 3. Minimal FIFO size

```
        sensitive << fifo.PUT;           // Process puts to FIFO
        async_reset_signal_is(nrst, 0);
        SC_METHOD(consumerProc);
        sensitive << fifo.GET;           // Process gets from FIFO
    }
}

void producerProc() {
    fifo.reset_put();
    wait();
    while (true) {
       if (fifo.ready()) {              // If FIFO is ready put next value
          fifo.put(getSomeVal());
       }
       wait();
    }
}
void consumerProc() {
    fifo.reset_get();
    T val;
    if (fifo.get(val)) {
       doSomething(val);
    }
}
```

7.7.3 One process stores requests in FIFO. One process stores requests in FIFO example.

```
struct Top : public sc_module {
    sc_in<bool>       clk{"clk"};
    sc_in<bool>       nrst{"nrst"};
    sct_fifo<T, 5>    fifo{"fifo"};
    explicit Top(const sc_module_name& name) : sc_module(name) {
        fifo.clk_nrst(clk, nrst);
        SC_THREAD(storeProc);
        sensitive << fifo;              // Process puts and gets to FIFO
        async_reset_signal_is(nrst, 0);
    }
}

void storeProc() {
    fifo.reset();
    wait();
    while (true) {
       if (fifo.ready()) {
          fifo.put(getSomeValue());
       }
       wait();
       if (fifo.request()) {
          doSomething(fifo.get());
       }
    }
}
```

7.8 Buffer

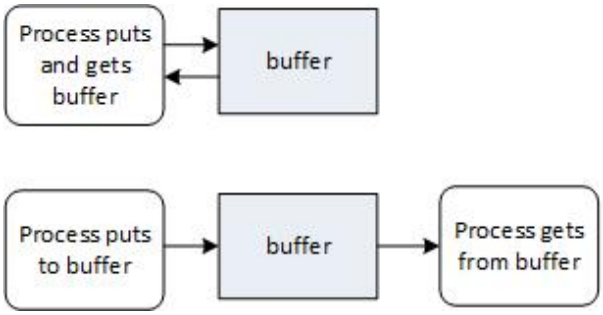

Fig. 11. Buffer

Buffer is a channel kind of FIFO to be used in single or two sequential processes. Buffer implements `sct_fifo_if` interface, the same as FIFO. Buffer differs from FIFO in higher simulation speed which is

achieved by implementation it as primitive channel (inheritor of sc_prim_channel). Buffer has one common implementation for cycle accurate and approximate time modes.

Buffer size can be 2 elements or more.

```
template<
    class T,
    unsigned LENGTH,                // Size (maximal number of elements)
    class TRAITS = SCT_CMN_TRAITS   // Clock edge and reset level traits
>
class sct_buffer {};
```

Buffer constructor has the same parameters as FIFO has. Parameters sync_valid and sync_ready should be 0 (false).

```
sct_buffer(const sc_module_name& name,
          bool sync_valid = 0,     // Request path has synchronous register
          bool sync_ready = 0,     // Response path has synchronous register
          bool use_elem_num = 0,   // Element number/Almost full or empty used
          bool init_buffer = 0)    // Initialize all buffer elements with zeros in reset
                                   // First element to get is always initialized to zero
```

Buffer can be used by one process to store data and for inter-process communication between two processes. peek() function of Buffer can be called from any process including combinational method process.

7.9 Pipeline register

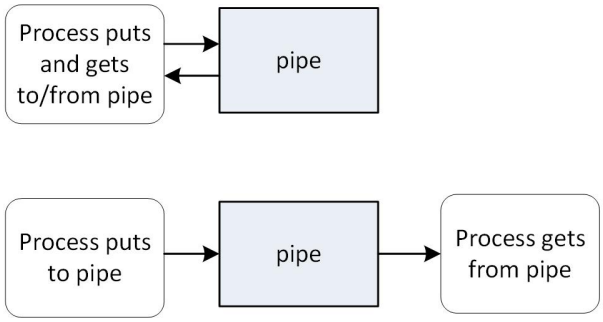

Fig. 12. Pipeline register

Pipeline register (sct_pipe) is intended to pipeline combinational logic and enable re-timing feature of a logic synthesis tool. In generated SystemVerilog code it can be replaced with DW_pl_reg or other implementation. Pipeline register can be used in one method or thread process as well as put in one process and get in other process. Pipeline register is normally added to sensitivity list of process where put or get done.

Pipeline register supports put bubbles and get backpressure. If there is no get, but some empty registers, they are shifted to provide next request put.

The pipeline register implements `sct_fifo_if` interface. It has size template parameter which is a positive number.

```
template<
    class T,
    unsigned N,                      // Number of pipeline registers, one or more
    class TRAITS = SCT_CMN_TRAITS,   // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE> // RTL (0) or TLM (1) mode
>
class sct_pipe {};
```

The pipeline register can have input or output registers, which are not used for re-timing.

```
sct_pipe(const sc_module_name& name,
        bool addInReg = 0,          // Add input register not moved by re-timing
        bool addOutReg = 0,         // Add output register not moved by re-timing
        const std::string& rtlName = "DW_pl_reg") // Pipeline register instantiated
            component name
```

Typical use case for pipeline register is combinational logic re-timing in method process.

```
void methProc() {
    run.reset_get();
    resp.reset_put();
    pipe.reset();

    if (pipe.ready() && run.request()) {
        T data = compute(run.get());  // Heavy computation to be pipelined
        pipe.put(data);
    }

    if (pipe.request() && resp.ready()) {
        resp.put(pipe.get());
    }
}
```

7.10 Register

Register is used to add state for METHOD process. Register is written in one method process and could be read in the same or other method process(es). Register is normally added to sensitivity list of process where it is read. Register can be read in thread process.

Register has the same template parameters as Target/Initiator:

```
template<
```

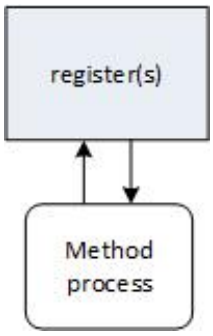

Fig. 13. Register

```
    class T,                          // Payload data type
    class TRAITS = SCT_CMN_TRAITS,   // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE> // RTL (0) or TLM (1) mode
class sct_register {};
```

Register has the following methods:

```
// Reset register, set it value to stored at last clock edge
void reset();
// Write new value to register
void write(const T& data);
// Read value stored at last clock edge
T read();
// To skip using read()
operator T ();
```

Register can initiate a new request. That means an output request can depend on register state.

```
sct_target<T>     targ{"targ"};
sct_register<T>   cntr{"cntr"};
explicit A(const sc_module_name& name) : sc_module(name) {
    targ.clk_nrst(clk, nrst);
    cntr.clk_nrst(clk, nrst);
    SC_METHOD(checkProc); sensitive << targ << cntr;
}

void checkProc() {
    cntr.reset();
    // Register accumulates received data up to N
    if (cntr.read() > N) {
        cntr.write(0);
    } else
    if (targ.get(data)) {
```

```
        cntr.write(cntr.read()+data);
    }
}
```

Read register in thread process should be done carefully. If register value is checked to generate an output or change a state, it could lead to incorrect behavior in TLM, if there is no other activation source for the process.

```
sct_register<T>   cntr{"cntr"};
sct_initiator<T>  init{"init"};
explicit A(const sc_module_name& name) : sc_module(name) {
    cntr.clk_nrst(clk, nrst);
    init.clk_nrst(clk, nrst);
    SCT_THREAD(cntrProc); sensitive << cntr << init;
    async_reset_signal_is(nrst, 0);
    // cntr is assigned in some method process
}

// Probably incorrect version
void cntrProc() {
    init.reset_put();
    wait();
    while (true) {
        if (cntr.read() > 10) {       // In TLM mode no process activation
           init.b_put(cntr.read());   // until cntr value changed
        }
        wait();
    }
}

// Correct version
void cntrProc() {
    init.reset_put();
    T lastCntr = 0;
    wait();
    while (true) {
        // Request sent only when cntr value changed
        if (cntr.read() > 10 && cntr.read() != lastCntr) {
           lastCntr = cntr.read();
           init.b_put(cntr.read());
        }
        wait();
    }
}
```

The same problem is actual for `sct_signal`.

7.11 Clock, clock gate and clock gate signal

`sct_clock<>` is implementation of clock source (generator) like `sc_clock` with enable/disable control.

```
    /// Enable clock activity, clock is enabled after construction
    void enable();
    /// Disable clock activity, can be called at elaboration phase to disable
    /// clock at simulation phase start
    void disable();
    /// Register clock gate signals/ports to control clock activity.
    /// If any of the signals/ports is high, then clock is enabled
    void register_cg_enable(sc_signal_inout_if<bool>& enable);
    /// Get clock period
    const sc_time& period() const;
```

Clock gate cell `sct_clock_gate_cell` and clock signal `sct_clk_signal` should be used together to connect clock input to gated clock source. `sct_clk_signal` is special signal without DC delay in written value becomes readable.

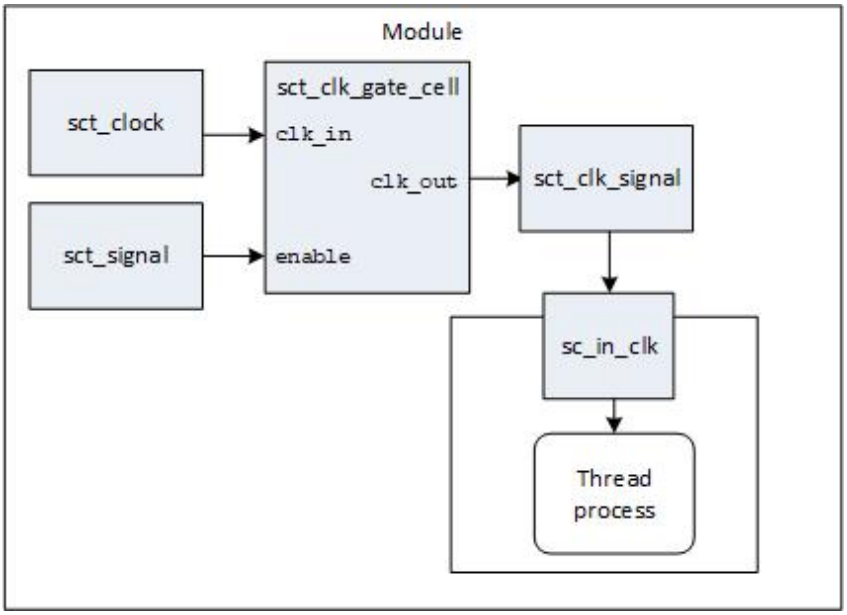

Fig. 14. Clock and clock gate

The code example illustrates using `sct_clock_gate_cell` and `sct_clk_signal`.

```
SC_MODULE(A) {
    sc_in_clk           SC_NAMED(clk);
    sc_in<bool>         SC_NAMED(nrst);
    sc_in<bool>         SC_NAMED(clk_enbl);
```

```
    sct_clk_signal        SC_NAMED(clk_out);
    sct::sct_clk_gate_cell SC_NAMED(clk_gate);
    sc_in<bool>           SC_NAMED(clk_in);

    explicit A(const sc_module_name& name) : sc_module(name) {
        clk_gate.clk_in(clk);     // Clock input
        clk_gate.enable(clk_enbl); // Gate clock input
        clk_gate.clk_out(clk_out); // Gated clock output
        clk_in(clk_out);

        SCT_THREAD(thrdProc, clk_in, nrst); // Use clock input bound to gated clock
        async_reset_signal_is(nrst1, 0);
}};
```

Clock gate cells can be sequentially connected to each other, gated clock output of one cell bound to clock input of anther cell.

In TLM mode is all thread processes are created with SC_THREAD/SCT_THREAD macros, clock source(s) can be disabled. Disabling sct_clock allows to speed simulation:

```
sct_clock<>   clk{"clk", 1, SC_NS};
explicit A(const sc_module_name& name) : sc_module(name) {
    if (SCT_CMN_TLM_MODE) {
         clk.disable();
    }
}
```

7.12 Reset

7.12.1 Reset section. In thread process reset logic initializes registers, local variables and output signals. This logic should be placed in reset section (code scope before first wait()).

```
sct_out<T> o{"o"};
sct_signal<T> s{"s"};
void thrdProc() {
    // Reset section
    int a = 0;               // Local variable
    s = 0;                   // Register
    o = 0;                   // Output
    wait();
    while (true) {
        ...
        wait();
    }
}
```

In method process initialization logic initializes local variables and output signals. This logic is normally be placed in the beginning of the process.

```
sct_out<T> o{"o"};
void methdProc() {
    // Initialization section
    int a = 0;               // Local variable
    o = 0;                   // Output
    ...
    a = i + 1;
    if (s) o = a;
}
```

Initialization logic in method process could be merged with its behavior logic based on inputs and registers. Such code style can have better simulation performance.

```
sct_in<T> i{"i"};
sct_out<T> o{"o"};
void methdProc() {
    int a = i+1;             // Local variable
    o = a ? s : 0;           // Output
    ...
}
```

The communication channels also need to be reset with specified `reset()`, `reset_get()` and `reset_put()` methods. In thread process every channel used in this process should be initialized in the reset section.

```
sct_initiator<T> init{"init"};
sct_target<T>   targ{"targ"};
sct_fifo<T, 2>  fifo{"fifo"};
void thrdProc() {
    init.reset();
    targ.reset();
    fifo.reset_put();        // If FIFO used for put
    fifo.reset_get();        // If FIFO used for get
    fifo.reset();            // If FIFO used for get and put both
    wait();
    while (true) {
        ...
        wait();
    }
}
```

In method process every channel used in this process is initialized in the beginning of the process or assigned at all execution path in the process code. Having no explicit reset for registers, signals, output ports and synchronizers can improve simulation performance.

```
sct_initiator<T> init{"init"};
sct_target<T>   targ{"targ"};
sct_register<T> reg1{"reg1"};
sct_register<T> reg2{"reg2"};
void methProc() {
    init.reset();
    reg1.reset();
    T val = targ.get(); // targ is accessed at all path, no reset required
    if (val > 0) {
        reg1 = val;    // reg1 accessed at some paths only, reset required
        init.put(val); // init accessed at some paths only, reset required
    }
    reg2 = val + 1;    // reg2 is accessed at all path, no reset required
}
```

7.12.2 Reset control. Reset signal can be asserted/de-asserted in TB and DUT processes as well. To have the same simulation time in RTL and TLM modes it needs to follow the rules given in this section.

If reset control thread is in TB, it could control reset based on time period and be non-sensitive to any channels. In this case such a thread should be SC_CTHREAD in RTL mode and SC_THREAD in TLM mode. To avoid extra activation in TLM mode, this thread should wait for a specified time instead of clock events.

```
SC_MODULE(A) {
   SC_CTOR(A) {
       // Thread not sensitive to anything
       #ifdef SCT_TLM_MODE
          SC_THREAD(resetProc);
       #else
          SC_CTHREAD(resetProc, clk_in.pos());
       #endif
   }
   #define rstWait(N) if (SCT_CMN_TLM_MODE) wait(N, SC_NS); else wait(N);
   void resetProc() {
        nrst = 0;
        rstWait(3);
        cout << sc_time_stamp() << " " << sc_delta_count() << " de-assert reset\n";
        nrst = 1;
        rstWait(5);
        ...
   }
};
```

If reset control thread is sensitive to any channels, it should be SCT_THREAD and have dont_initialize() in RTL mode. Such a thread can also be a normal test thread which provides stimulus and checks results:

```
SC_MODULE(A) {
  SC_CTOR(A) {
      // Thread sensitive to SS channels
       SCT_THREAD(resetProc, clk);
       #ifndef SCT_TLM_MODE
          dont_initialize();
       #endif
       sensitive << s;
  }
  sct_signal<unsigned> s{"s"};
  void resetProc() {
       nrst = 0;
       while (s.read() < 3) {s = s.read()+1; wait();}
       cout << sc_time_stamp() << " " << sc_delta_count() << " de-assert reset\n";
       nrst = 1;
  }
```

7.12.3 Specify clock edge and reset level. Clock edge and reset level normally are the same for the design. To update them for whole design SCT_CMN_TRAITS should be defined:

```
#define SCT_CMN_TRAITS SCT_NEGEDGE_POSRESET // Set negative edge and positive reset level
```

To specify clock edge and reset level for individual library modules, template parameters should be used, for example:

```
sct_target<T, SCT_NEGEDGE_POSRESET>   run{"run"};
sct_initiator<T, SCT_POSEDGE_NEGRESET> resp{"resp"};
```

7.13 Array of SingleSource channels

Array of SingleSource channels can be implemented with sc_vector. First parameter of sc_vector is name, second parameter is number of elements (should be a compile time constant). To provide additional parameters to single source channels, it needs to use lambda function as third parameter of sc_vector.

```
static const unsigned N = 16;
using T = sc_uint<16>;
sc_vector<sct_target<T>>    targ{"targ", N};  // Two parameters
sc_vector<sct_initiator<T>> init{"init", N,   // Three parameters
  [](const char* name, size_t i) {            // Lambda function
       return sc_new<sct_initiator<T>>(name, 1); // Initiator with sync register
  }};
```

7.14 Target and initiator in top module

Target and Initiator can be instantiated in top module to be connected to the correspondent modules in testbench. Such top module is synthesizable with input/output ports for the Target/Initiator instances.

Top module can contain Target which is not always ready and has no synchronous register. Top module can contain initiator which has no synchronous register. Top module cannot contain MultiTarget or MultiInitiator. Vector (`sc_vector`) of Target/Initiator in top module is supported.

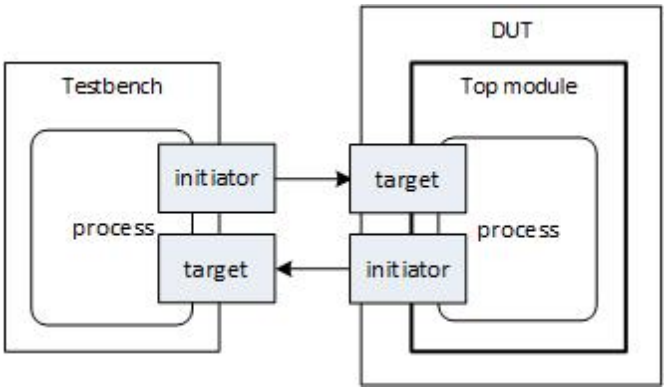

Fig. 15. Target and initiator in top module

To connect testbench Target/Initiator to the correspondent top module Initiator/Target normal `bind()` function is used always except multi-language simulation. For multi-language simulation if DUT is in SystemVerilog and testbench is in SystemC language, the simulation tool generates a special SystemC wrapper for DUT top module. To connect this wrapper to SystemC testbench `SCT_BIND_CHANNEL` macro should be used. `SCT_BIND_CHANNEL` macro cannot be applied to Target/Initiator with record type.

```
// Include DUT module generated wrapper or SystemC header
#ifdef RTL_SIM
    #include "DUT.h"        // Multi-language simulation, include generated wrapper
#else
    #include "MyDut.h"      // SystemC simulation and synthesis, include designed header
#endif

template<class T>
class MyModule : public sc_module {
   DUT                     dut{"dut"};
   sct_target<T>           targ{"targ"};
   SC_CTOR(MyModule) {
       // Bind targ to init in dut module
       #ifdef RTL_SIM
           SCT_BIND_CHANNEL(dut, init, targ);     // Multi-language simulation
       #else
           targ.bind(dut.init);                   // SystemC simulation and synthesis
       #endif
```

```
   }
}
```

7.15 Array of Target/Initiator in top module

Array of Targets/Initiators supported in any module including top module. Instead of C++ array `sc_vector` should be used (C++ array is not supported). To bind the Targets/Initiators `SCT_BIND_CHANNEL` macro with 4 parameters is provided.

```
// Include DUT module generated wrapper or SystemC header
#ifdef RTL_SIM
    #include "DUT.h"       // Multi-language simulation, include generated wrapper
#else
    #include "MyDut.h"     // SystemC simulation and synthesis, include designed header
#endif

template<class T, unsigned N>
class MyModule : public sc_module {
   DUT            dut{"dut"};
   sc_vector<sct_target<T>> targ{"targ", N};
   SC_CTOR(MyModule) {
       // Bind all elements of targ to elements of init in dut module
       #ifdef RTL_SIM
           SCT_BIND_CHANNEL(dut, init, targ, N); // Multi-language simulation
       #else
           for (unsigned i = 0; i != N; ++i)
               targ[i].bind(dut.init[i]);      // SystemC simulation and synthesis
       #endif
   }
}
```

7.16 Hierarchical connection of Target and Initiator

Target and Initiator can be connected through module hierarchy from child module up to parent module. That is possible explicitly or with `sc_port` of Initiator/Target.

There is an example of explicit binding Target to Initiator through module hierarchy:

```
template<class T>
struct Child : public sc_module {
    sct_target<T>     run{"run"};
};

template<class T>
struct Parent: public sc_module {
    Child<T>         child{"child"};
```

```
    SC_CTOR(Parent) {}
};

SC_MODULE(Top) {
    sct_initiator<T>  resp{"resp"};
    Parent<T>         parent{"parent"};
    SC_CTOR(Top) {
        parent.child.run.bind(resp);
}};
```

Ports (`sc_port`) of Target/Initiator contain pointer to them. To bind Initiator to Target through ports it needs to use `get_instance()` method which provides Target/Initiator from its port.

```
template<class T>
struct Child : public sc_module {
    sct_target<T>     run{"run"};
};

template<class T>
struct Parent: public sc_module {
    sc_port<sct_target<T>>    run;
    Child<T>                  child{"child"};
    SC_CTOR(Parent) {
        run(child.run); // Bind port to child module initiator
}};

SC_MODULE(Top) {
    sct_initiator<T>  resp{"resp"};
    Parent<T>         parent{"parent"};
    SC_CTOR(Top) {
        // get_instance() provides Initiator from its sc_port
        resp.bind(parent.run->get_instance());
}};
```

Process which calls Target/Initiator functions should be in the module where Target/Initiator declared. If a process calls Target/Initiator through its port (`sc_port<sct_target>/sc_port<sct_initiator>`) the process module and target initiator module should be synthesized in the same parent module.

7.17 Module interconnect with FIFO

Connection between modular interfaces inside of common parent module can be done with FIFO. One modular interface should have a FIFO instance and other modular interface should have a FIFO port (`sc_port< sct_fifo<> >`). The same can be done if the FIFO is instantiated in the parent module.

```
template<class T, unsigned N>
struct A : public sc_module, sc_interface {
   sct_fifo<T, N>  SC_NAMED(fifo);
```

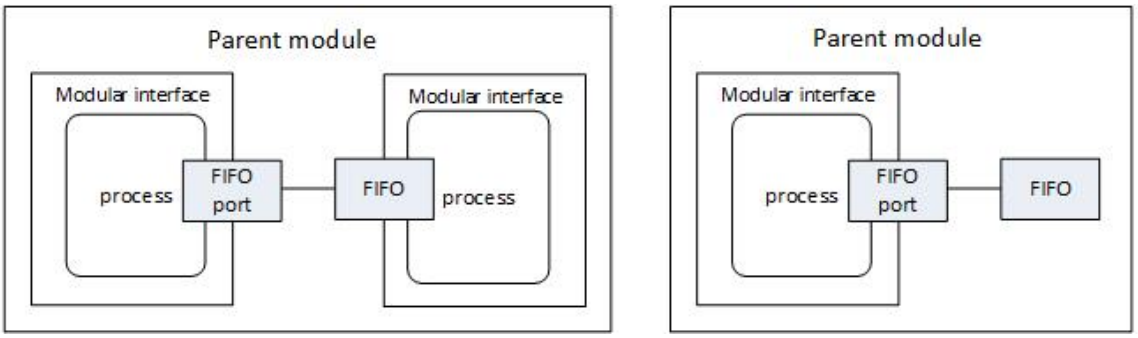

Fig. 16. FIFO port

```
   ...
};

template<class T, unsigned N>
struct B : public sc_module, sc_interface {
   sc_port<sct_fifo<T, N>> SC_NAMED(fifo_port);
   ...
};

struct Parent : public sc_module{
   A<int, 3>  SC_NAMED(a);
   B<int, 3>  SC_NAMED(b);
   SC_CTOR(Parent) {
      b.fifo_port(a.fifo);  // Bind FIFO port to FIFO instance
   }
};
```

7.18 Cycle accurate and SingleSource code mix

Conventional cycle accurate SystemC modules can be mixed with single source modules without limitations. `sct_clock` should be used instead of normal `sc_clock`.

Cycle accurate threads created with `SC_CTHREAD` macro are activated by clock event. Such processes can use SingleSource channels to communicate to each other and SingleSource threads created with `SCT_THREAD` macro. Cycle accurate processes should be sensitive to all the SingleSource channels used inside.

```
template<class T>
class MyModule : sc_module {
    sct_target<T>     in{"in"};
    sct_signal<T>     s{"s"};
    sct_fifo<T, 2>    fifo{"fifo"};
```

```
    MyModule(const sc_module_name& name) : sc_module(name) {
        SC_CTHREAD(threadProc, clk);
        sensitive << in << fifo.PUT << s; // sensitivity to all used channels
        async_reset_signal_is(nrst, 0);
    }
    void threadProc() {
        in.reset_get();
        fifo.reset_put();
        wait();
        while (true) {
            if (in.request()) {
                fifo.put(s.read());
                in.get();
            }
            wait();
        }
    }
}
```

Instead of SC_CTHREAD special SCT_CTHREAD macro could be used. SCT_CTHREAD supports clock edge with third parameter:

```
SCT_CTHREAD(proc, clk, clk_edge); // clk_edge 0 -- negedge, 1 -- posedge, 2 -- both edges
SCT_CTHREAD(proc, clk);          // clk_edge is SCT_CMN_TRAITS::CLOCK
```

7.19 Record type in SingleSource channels

Record is supported as data type in all SingleSource channels. The record should comply SystemC requirements for records used in signal/port: the record should have default constructor w/o parameters, `operator==()`, `operator«(std::ostream)` and `sc_trace()` implemented.

```
struct Rec_t {
    bool enable;
    sc_uint<16> addr;
    // Default constructor
    Rec_t() : enable(false), addr(0) {}
    // Another constructor, optional
    Rec_t(bool enable_, sc_uint<16> addr_) : enable(enable_), addr(addr_) {}
    bool operator == (const Rec_t& other) const {
        return (enable == other.enable && addr == other.addr && indx == other.indx);
    }
};
namespace std {
inline ::std::ostream& operator << (::std::ostream& os, const Rec_t& r) {
    os << r.enable << r.addr << r.indx; return os;}
}
```

```
namespace sc_core {
void sc_trace(sc_trace_file* , const Rec_t& , const std::string&) {}
}
...
sct_target<Rec_t> run{"run"};
void methProc() {
   run.reset_get();
   if (run.request()) {
      Rec_t data = run.get(); // Get record fields from target
   }
}
```

See more examples at `design/tests/records`

8 ADVANCED VERIFICATION FEATURES

The SystemC compiler supports immediate and temporal assertions described in this section.

Beside that 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_unknown(value)` – check give value is unknown, i.e. not statically evaluated
- `sct_assert_register(expr)` – check given expression is read before defined

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 ans 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_STABLE(LHS, TIME, RHS, EVENT); // In module scope
SCT_ASSERT_ROSE(LHS, TIME, RHS, EVENT);   // In module scope
SCT_ASSERT_FELL(LHS, TIME, RHS, EVENT);   // In module scope
SCT_ASSERT_THREAD(EXPR, EVENT);           // In clocked thread
SCT_ASSERT_THREAD(LHS, TIME, RHS, EVENT); // In clocked thread
SCT_ASSERT_LOOP(LHS, TIME, RHS, EVENT, ITER); // In for-loop in 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.

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

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);
SCT_ASSERT_STABLE(LHS, TIME, RHS, EVENT);
SCT_ASSERT_ROSE(LHS, TIME, RHS, EVENT);
SCT_ASSERT_FELL(LHS, TIME, RHS, EVENT);
```

Time delay for these assertion means immediate or one cycle delayed checking of stable/rose/fell of the consequent expression. Time interval for SCT_ASSERT_STABLE specify how long the consequent expression should be stable.

SCT_ASSERT_STABLE, SCT_ASSERT_ROSE and SCT_ASSERT_FELL have some limitation on time parameter:

- SCT_ASSERT_STABLE can have time delay 0 and 1 and time interval (0,1),
- SCT_ASSERT_ROSE and SCT_ASSERT_FELL can have time delay 0 and 1 only.

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];
...
77: SCT_ASSERT(req || val == 0, clk.pos());       // OK
78: SCT_ASSERT(req, SCT_TIME(1), resp, clk.pos()); // OK
79: SCT_ASSERT(req, SCT_TIME(N+1), resp, clk.neg()); // OK, constant time
80: SCT_ASSERT(req, (2), val.read(), clk);        // OK, brackets only form
81: SCT_ASSERT(val, SCT_TIME(2,3), *pval, clk.pos()); // OK, time interval
82: SCT_ASSERT(arr[0], (N,2*N), arr[N-1], clk.pos()); // OK, brackets only form
83: SCT_ASSERT(val == N, SCT_TIME(1), resp, clk.pos()); // OK, constant used
84: SCT_ASSERT(m == 0, (1), resp, clk.pos());     // Error, member variable used
85: SCT_ASSERT(resp, (0,2), arr[m+1], clk.pos()); // Error, non-constant index
86: SCT_ASSERT_STABLE(req, (0), resp, clk.pos()); // OK
87: SCT_ASSERT_STABLE(req, (2), resp, clk.pos()); // Error, delay can be 0 or 1
88: SCT_ASSERT_STABLE(req, (1,3), resp, clk.pos()); // OK
89: SCT_ASSERT_ROSE(req, (0), resp, clk.pos());   // OK
90: SCT_ASSERT_FELL(req, (1), resp, clk.pos());   // OK
91: SCT_ASSERT_ROSE(req, (0,1), resp, clk.pos()); // Error, time interval for rose
```

Generated SVA:

```
`ifndef INTEL_SVA_OFF
sctAssertLine77 : assert property (
    @(posedge clk) true |-> req || val == 0 );
sctAssertLine78 : assert property (
    @(posedge clk) req |=> resp );
sctAssertLine79 : assert property (
    @(negedge clk) req |-> ##4 resp );
sctAssertLine80 : assert property (
    @(negedge clk) req |-> ##2 val );
...
sctAssertLine86 : assert property (
    @(posedge clk) req |-> $stable(resp) );
sctAssertLine88 : assert property (
    @(posedge clk) req |=> $stable(resp)[*3] );
sctAssertLine89 : assert property (
    @(posedge clk) req |-> $rose(resp) );
sctAssertLine90 : assert property (
    @(posedge clk) req |=> $fell(resp) );
`endif // INTEL_SVA_OFF
```

Assertion expression can operate with SingleSource library channels. Target method `request()`, initiator method `ready()` and FIFO methods `request()`, `ready()`, `size()`, `elem_num()` could be used. Assertion expression can contain call of functions which have no parameters, have integral return type and consists of only return statement.

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

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

These assertions can operate with local data variables and local/member constants. Non-constant member data variables (not signals/ports) access in assertion can lead to data races. Because of that only member data which has stable value after elaboration phase could be used in assertions. Thread process assertions have no advantages over module scope assertions, so modules scope assertions are recommended to use.

These assertions can operate with SingleSource library channels and can contain calls of simple functions like assertions in module scope.

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_THREAD(req, SCT_TIME(1), ready, clk.pos()); // Assertion in reset section
   wait();
   SCT_ASSERT_THREAD(req, SCT_TIME(2,3), resp, clk.pos()); // Assertion after reset

   // 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); // Assertion after reset section
   end
   assert property (req |=> ready); // Assertion from reset
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_THREAD(req, SCT_TIME(0), ready, clk.pos()); // OK
   SCT_ASSERT_THREAD(req, SCT_TIME(N+1), ready, clk.pos()); // OK
   SCT_ASSERT_THREAD(req, (2,3), i == 0, clk.pos());    // OK, local variable used
   wait();
   if (N > 1) {
       SCT_ASSERT_THREAD(req, SCT_TIME(1), resp, clk.pos()); // OK, statically evaluated
   }
   SCT_ASSERT_THREAD(m > 1, (2), ready, clk.pos())      // OK, member variable used
   while (true) {
      ...
      SCT_ASSERT_THREAD(req, SCT_TIME(0), ready, clk.pos()); // Error
```

```
    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, EVENT, 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], clk.pos(), i);
    for (int j = 0; j < M; ++j) {
      SCT_ASSERT_LOOP(req[i][j], SCT_TIME(2), resp[i][N-j+1], clk.pos(), 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
```

9 EXTENSIONS

9.1 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
```

9.2 Memory module name

This section describes how to create a custom 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_MODULE_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_MODULE_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_MODULE_NAME__;
    explicit memory_stub(const sc_module_name& name,
                         const char* verilogName = "") :
        __SC_TOOL_MODULE_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), ...);
```