SystemC-to-Verilog Compiler: a productivity-focused tool for hardware design in cycle-accurate SystemC

Mikhail Moiseev, Intel Corporation
Roman Popov, Intel Corporation
Ilya Klotchkov, Intel Corporation

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Introduction

- SystemC-to-Verilog Compiler (SVC) translates cycle-accurate SystemC to synthesizable Verilog code
- SVC is focused on improving productivity of design and verification engineers
- SVC is not a HLS tool
- SVC has multiple advantages which distinguish it from other tools
  - C++11/14/17 support
  - Arbitrary C++ at elaboration phase (in module constructors)
  - Fast and simple code translation procedure
  - Human-readable generated Verilog code

C++ and SystemC support

- SVC uses SystemC 2.3.3
  - SystemC Synthesizable Standard fully supported
- SVC supports modern C++ standards
  - C++11, C++14, C++17
  - Partial support of STL containers
- No limitations on elaboration stage programming, arbitrary C++ supported
  - Load input data from file/database
  - Enables to design highly reusable IPs

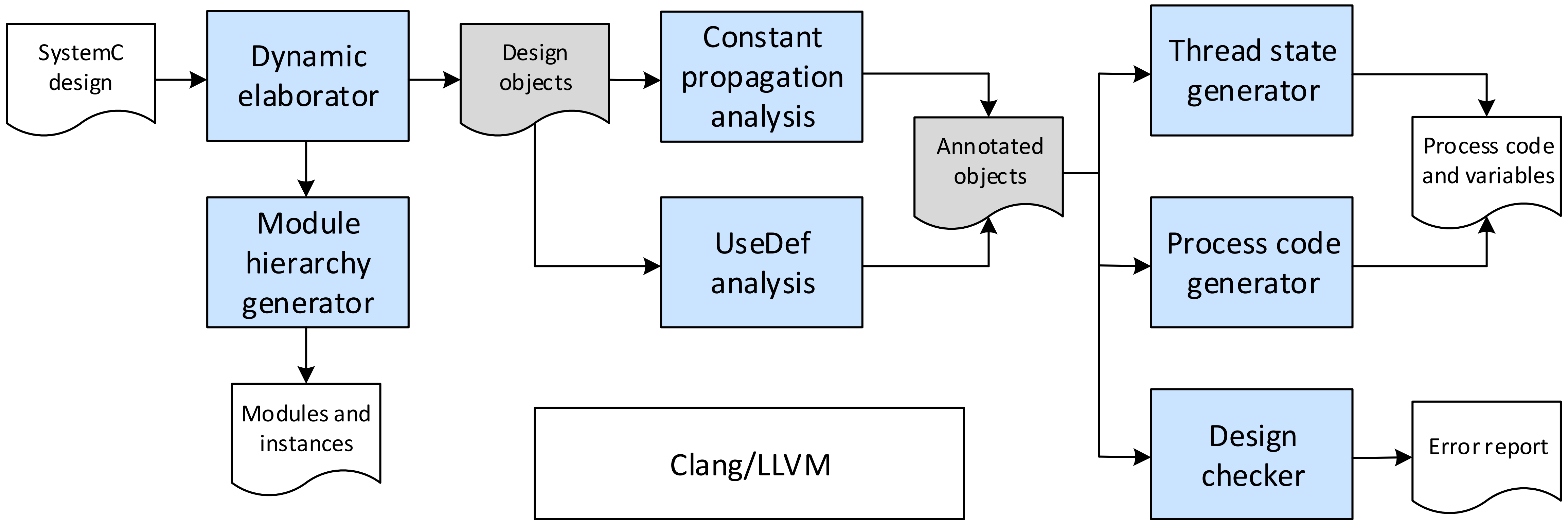
Fast and simple code translation

- SVC does minimal optimizations, leaving others to logic synthesis tool
  - Constant propagation and dead code elimination
  - Used optimizations intended to generate better looking code
- SVC works very fast
  - Elaboration takes several seconds
  - Code translation a few tens of seconds
- SVC uses conventional build system (CMake)
  - *svc_target()* runs tool for specified target and top module
  - No build script or configuration files required
  - No code polluting with pragmas

Human-readable generated Verilog

- SVC generates Verilog RTL which looks like SystemC source
  - Verilog variables have the same names everywhere it is possible
  - General structure of process/always block control flow is preserved
- Productivity advantages of human readable code
  - DRC and CDC bugs in generated Verilog can be quickly identified in input SystemC
  - Violated timing paths from ASIC logic synthesis tool can be easily mapped to input SystemC
- ECO fixes have little impact on generated Verilog

Tool architecture



Design Elaboration

- SVC implemented as library on top of SystemC and Clang libraries
  - Substitutes SystemC library for linking
  - Runs Verilog code generation after SystemC elaboration phase
- SVC extracts design structure directly from process memory
  - Module hierarchy, links from pointers to pointee objects, values of scalar types, sensitivity and reset lists for processes
- SVC gets information about types form Clang AST (Abstract Syntax Tree)
- Distinguish between pointer to dynamically allocated object and dangling pointer problem
  - Overriding *new* and *new[]* for sc_object inheritors, and *sc_new* for other types

Static Analysis of Process Functions

- Static analysis runs on CFG (Control Flow Graph) provided by Clang/LLVM
- Constant propagation analysis
  - Helps to determine number of loop iterations, eliminate dead code, substitute constant into generated code
- Used/defined variable analysis
  - Allows to split SystemC variables into local variables and registers
- Design correctness checking
  - Array out-of-bound access and dangling/null pointer dereference, Incomplete sensitivity lists for combinational methods, Inter-process communication through non-channel objects, Non-channel object read before initialization …

SC_METHOD example

```
sc_in<bool>        in;
sc_out<bool>       out;
sc_signal<bool>    s;

SC_CTOR(MyModule) {
   SC_METHOD(method_proc);
   sensitive << in << s;
}

void method_proc () {
   bool a = in;
   if (s != 0) {
      out = a;
   } else {
      out = 0;
   }
}
```

```
logic in, out;
logic s;

always_comb
begin                  // method_simple.cpp:112:5
    logic a;
    a = in;
    if (s != 0)
    begin
        out <= a;
    end else begin
        out <= 0;
    end
end
```

Clocked thread state generation

- SVC converts a thread into pair of *always_comb* and *always_ff* blocks
  - *always_ff* block implements reset and update logic for registers
  - *always_comb* block contains combinational logic that computes the next state
- Clocked thread can have multiple states specified with *wait()*
  - Number of states depends on number of *wait()* calls
  - Thread states are represented by automatically generated *PROC_STATE* variable
  - Main *case* of *PROC_STATE* in represents the SystemC thread FSM
- Thread variables divided into two groups
  - Local variables that are always assigned before use
  - Register variables that can retain their value from previous clock cycle

SC_CTHREAD example #1

```
sc_in<unsigned>    a;
sc_out<unsigned>   b;

SC_CTOR(MyModule) {
    SC_CTHREAD(thread1, clk.pos());
    async_reset_signal_is(rst, false);
}

void thread1 () {
   unsigned i = 0;
   b = 0;
   while (true) {
      wait();
      b = i;
      i = i + a;
   }
}
```

```
logic [31:0] a;
logic [31:0] b, b_next;
logic [31:0] i, i_next;
always_comb begin    // cthread_simple.cpp:101:5
    thread1_func;
end
function void thread1_func;
    b_next = b; i_next = i;
    b_next = i_next;
    i_next = i_next + a;
endfunction
always_ff @(posedge clk or negedge rst)
begin : thread1_ff
    if ( ~rst ) begin
        i <= 0; b <= 0;
    end else begin
        i <= i_next; b <= b_next;
    end
end
```

SC_CTHREAD example #2

```
sc_in<sc_uint<8>>    a;
sc_out<sc_uint<8>>   b;

SC_CTOR(MyModule) {
    SC_CTHREAD(thread2, clk.pos());
    async_reset_signal_is(rst, false);
}

void thread2() {
   sc_uint<8> i = 0;
   b = 0;
   wait();                       // STATE 0
   while (true) {
      auto j = a.read();
      i = j + 1;
      wait();                    // STATE 1
      b = i;
   }
}
```

```
logic PROC_STATE, PROC_STATE_next;
always_comb begin // cthread_simple.cpp:114:5
    thread2_func;
end
function void thread2_func;
    integer unsigned j;
    b_next = b; i_next = i;
    case (PROC_STATE)
        0: begin
            j = a; i_next = j + 1;
            PROC_STATE_next = 1; return;
        end
        1: begin
            b_next = i_next;
            j = a; i_next = j + 1;
            PROC_STATE_next = 1; return;
        end
    endcase
endfunction
```

Clocked thread code generation

- The flow control statements *if*, *switch* and loops without *wait()* calls are converted into equivalent Verilog statements
- Loops with *wait()* calls are divided into several states
  - Loop statement in that case is replaced by *if* statement
- If loop with *wait()* contains *break* or *continue* statements they are replaced with the code up to the next *wait()* call

SC_CTHREAD with loop example

```
void thread_loop() {
    wait();                     // STATE 0
    while (true) {
        for (int i = 0; i < 10; i++) {
            k[i] = n[i] / m[i];
            wait();             // STATE 1
        }
        wait();                 // STATE 2
    }
}
```

```
function void thread_loop_func;
    case (PROC_STATE)
        0: begin
            i_next = 0;
            k[i_next] = n[i_next] / m[i_next];
            PROC_STATE_next = 1; return;
        end
        1: begin
            i_next++;
            if (i_next < 10)
            begin
                k[i_next] = n[i_next] / m[i_next];
                PROC_STATE_next = 1; return;
            end
            PROC_STATE_next = 2; return;
         end
         2: ...
    endcase
endfunction
```

SC_CTHREAD with break example

```
void thread_break() {
   wait();                 // STATE 0
   while (true) {
      wait();              // STATE 1
      while (!enabled) {
         if (stop) break;
         wait();           // STATE 2
      }
      ready = false;
   }
}
```

```
function void thread_break_func;
    case (PROC_STATE)
        0: ...
        1: begin
             if (!enabled) begin
                 if (stop) begin
                     // break begin
                     ready_next = 0;
                     PROC_STATE_next = 1; return;
                     // break end
                 end
                 PROC_STATE_next = 2; return;
             end
             ready_next = 0;
             PROC_STATE_next = 1; return;
        end
        2: ...
   endcase
endfunction
```

Synthesis time for memory designs

Design name	Number of modules (instances)	Number of processes (instances)	Generated code, LoC	Translation time
A	58 (308)	161 (711)	29181	6 sec
B	19 (252)	65 (811)	20724	81 sec
C	78 (581)	291 (1470)	53404	18 sec
D	15 (57)	41 (146)	4662	2 sec
E	167 (880)	765 (2713)	87622	21 sec
F	53 (161)	173 (523)	25715	7 sec
G	57 (157)	170 (400)	21061	5 sec

Area and performance results

Design name	SVC		Alternative implementation*	
	Area Reg / LUT	Freq MHz	Area Reg / LUT	Freq MHz
A	2.4K / 10.2K	63	2.5K / 10.3K	62
B	54K / 145K	52	59K / 151K	53
C	15.7K / 46K	35	14.3K / 48K	25
D	547 / 1812	174	484 / 1823	172

*Alternative implementation – Verilog code for these designs created in another way

Reusable module library

- SVC library contains modules with functional interfaces
  - Module/process interconnect: FIFO with blocking/non-blocking IF
  - AMBA master and slave ports
  - Memory wrapper for vendor SRAM, RF, ROM with power control
  - Memory configurator to assemble required data width/word number with multiple memories
  - Clock gate, clock synchronizer and other standard modules
- Verilog code intrinsic

Future plans

- Cope problem with pointers (to avoid of *new/new[]* patch and *sc_new*)
  - Preprocess files to replace *new* with *sc_new*
  - Extend dynamic elaboration with static one
- Temporal assertions in SystemC with automatic translation into SVA

Questions

Vendor memory

- Common interface

```
class mem_wrapper_if : public virtual sc_interface {
    virtual void writeRequest(Addr addr, Data data, Data bitEnable);
    virtual void readRequest(Addr addr);
    virtual Data getData(bool checkValid = true);
    virtual void addRDataSensitive(sc_sensitive &s);
    virtual void sleepMemory(unsigned mode = 0);
    virtual void wakeupMemory();
    ...}
```

- Classes for RF, SRAM and ROM for different technology processes

```
template <unsigned WORD_COUNT, unsigned DATA_WIDTH, bool USE_BENABLE>
class sram_tech_wrapper : public mem_wrapper_if {...}
```

Advanced verification features (WIP)

- SystemC assertion currently used at IP level verification only
  - Assertion activated by untimed condition
  - No assertion propagation to synthesized RTL
- Export IP level assertion to be used at SoC verification
  - SystemC assertions automatically converted into SVA during RTL generation

```
sc_assert(a == 1);  // SystemC assertion
assert(a == 1) else $error("Assertion at main.cpp:32 failed"); // SVA
```

- Advance SystemC assertions with temporal conditions like in SVA
  - SystemC language extension
  - Supported in SystemC simulation as well

```
sc_assert(req,  1, !empty);  // SystemC assertion
assert(req ##1 !empty) else $error("…");  // SVA
```

Area and performance results ASIC

Design name	SVC		Alternative implementation*	
	Area	Freq MHz	Area	Freq MHz
A	4.2	761	3.9	752
B	26.7	694	23.4	686
C	91.5	400	81.2	377
D	169	763	158	769
E	20.9	769	19.6	771
F	70	1470	75	1420