2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

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

Mikhail Moiseev, Intel Corporation, Hillsboro, USA (mikhail.moiseev@intel.com)

Roman Popov, Intel Corporation, Hillsboro, USA (roman.i.popov@intel.com)

Ilya Klotchkov, Intel Corporation, Hillsboro, USA (ilya.v.klotchkov@intel.com)

***Abstract*—We present the SystemC-to-Verilog Compiler (SVC) – a tool that translates cycle accurate SystemC to synthesizable Verilog. SVC supports SystemC synthesizable subset in method and thread processes and arbitrary C++ at elaboration phase. SVC produces human-readable Verilog for complex multi-module designs in tens of seconds. The tool performs design checks to detect non-synthesizable code and common coding mistakes. SVC is focused on improving productivity of design and verification engineers, and leaves optimization work for an underlying logic synthesis tool.**

Keywords— SystemC; hardware design; code translation; high level design

I. INTRODUCTION

Hardware design and verification of complex IP cores and SoCs under tight schedules and with a lack of engineering resources require new methodologies, languages and tools to increase designer productivity. In comparison with hardware design flow, algorithm and software development is typically done at higher level of abstraction and utilizes programming languages with large ecosystems of tools, libraries and frameworks. A possible way to improve hardware designer productivity is adapting abstractions and languages used for software development. There are several attempts to adopt a general-purpose programming language for hardware design: Chisel based on Scala, MyHDL based on Python, Clash based on Haskell and SystemC based on C++ which is used in variety of high-level synthesis (HLS) tools.

In this paper we present SystemC-to-Verilog Compiler (SVC) tool that does translation of cycle-accurate SystemC into synthesizable Verilog. SVC is focused on productivity of design and verification engineers that provided with the following advantages distinguish it from other SystemC tools:

- Arbitrary C++ at elaboration phase (in module constructors);
- C++11/14/17 support;
- Human-readable generated Verilog code;
- Fast and simple code translation procedure.

One of the SystemC advantages is supporting generation and parameterization of hardware modules at elaboration phase. That enables development of highly reusable IPs and assembling a new design from pre-verified modules. SVC tool has dynamic elaboration that allows executing arbitrary C++ code at elaboration phase and gives even more flexibility for reusable IP development. As SVC is based on Clang [1], it supports C++11/14/17 standards in the same way as Clang does. For process functions SVC tool supports SystemC synthesizable subset [2].

SVC generates human-readable Verilog that looks similar to the SystemC source. Verilog variables have the same names as SystemC source variables everywhere it is possible. Verilog always blocks have labels from SystemC process names, function calls are inlined with function name printed in the comments. General structure of process/always block control flow is also preserved. Verilog code, similar to the SystemC source, allows quick

fixes for bugs detected in Verilog simulation, simplifies timing report analysis and ECO fixes. Existing SystemC/C++ HLS tools generate Verilog code which typically looks very different from SystemC source.

SVC tool does not perform optimizations, leaving them to a logic synthesis tool, which is the next in design flow. That distinguishes SVC from existing SystemC/C++ HLS tools, which typically perform area and performance optimizations. Since SVC excludes optimizations, it works very fast, usually design elaboration and checking takes several seconds, and code generation takes a few tens of seconds for design of average complexity. SVC uses CMake build system and does not requires any special build script or configuration files.

The presented SVC tool is targeted to cycle-accurate SystemC designs. Cycle-accurate design means timing behavior is fully specified by designer with *wait()* statements. SVC translates cycle-accurate SystemC to equivalent Verilog, preserving timing behavior. Cycle-accurate SystemC is used in implementation of any protocol with timing requirements like memory and bus. Evaluation shows that for such SystemC designs quality of results (area and performance) of a FPGA logic synthesis tool for Verilog codes generated by SVC and by one of existing commercial HLS tools are comparable.

Rest of the paper consists of five sections. Section II briefly describes other programming languages used for hardware design and compares our approach with conventional SystemC HLS flow. Section III presents SystemC dynamic elaboration. Section IV considers the SVC tool code analysis, translation rules and Verilog generation. In Section V evaluation results for a number of industrial designs are given. The last section concludes the paper.

II. BACKGROUND

Using a general-purpose programming language as a hardware description language is a popular idea and there are multiple implementations based on different programming languages. Commonly named benefits of this approach are:

- Raising the level of abstraction by utilizing programming paradigms like object orientation, functional programming, parameterized types, dynamic typing and type inference;
- Native support for writing hardware generators in underlying programming language;
- Reuse of large software ecosystems of libraries and frameworks.

Besides SystemC, among actively maintained projects are MyHDL, Chisel and Clash. MyHDL [12] is a Python-based DSL for hardware modeling and discrete-event simulation. Hardware processes in MyHDL are modeled with Python generators, which are Python implementation of coroutines. Similarly to VHDL, MyHDL has signals for inter-process communication and variables for process-local data.

MyHDL Verilog/VHDL converter uses Python introspection mechanisms to extract generated design structure and Python *ast* module to extract Abstract Syntax Tree (AST) of process bodies. It generates human-readable Verilog/VHDL with behavioral processes that closely match to input Python code. Main difference between SystemC and MyHDL are underlying programming languages. C++ is compiled, statically typed language with zero-overhead abstractions. Python is interpreted language with dynamic type system.

Chisel [3] is a Scala-based DSL that is focused on writing hardware generators. Unlike SystemC and MyHDL, Chisel has no behavioral processes, it is a purely structural language. Instead of converting pure Scala code to HDL, Chisel relies on custom types and operator overloading to implement a meta-language for describing circuit structure. Similarly to SystemC, Chisel uses classes and objects composition to model hardware hierarchy. It utilizes class constructors for value-based parameterization, and generics for type-based parameterization.

Clash [4] is a Haskell-based purely functional language for writing hardware generators. Similarly to Chisel, Clash is structural design language, but it follows a different implementation way. Instead of creating a meta-language using custom datatypes and overloads, Clash is implemented as a Haskell GHC compiler extension, and converts source-language AST into HDL. Comparing to Chisel and Clash, that are purely structural hardware generation languages, SystemC supports both behavioral and structural modeling. In our experience, behavioral modeling with clocked threads is essential for describing complex state machines.

C++ language and SystemC library are widely utilized in EDA and semiconductor industries. Common SystemC applications are virtual platform modeling, high-level synthesis, functional verification and mixed-signal modeling. Pure C/C++ HLS tools automatically transform untimed algorithmic code into optimized hardware microarchitecture. Input for C/C++ HLS tool is usually a function. Some HLS tools also support SystemC input. SystemC extends C++ with facilities to model concurrency, design hierarchy and cycle-accurate behavior.

In comparison with SVC, which does source-to-source translation, HLS tools perform design transformation for area and performance optimizations. Typical design transformations are: operation and register sharing, loop transformation, clocked process state and logic transformation. For cycle-accurate SystemC designs those optimizations usually duplicate work performed by logic synthesis tool. Our experiments demonstrate for cycle-accurate designs QoR of a FPGA logic synthesis tool are comparable for Verilog generated by SVC tool and an HLS tool.

SVC tool does not change timing behavior of SystemC design. SVC generates Verilog equals to the input design, therefore SystemC design should be cycle-accurate. That means SystemC processes should have all the *wait()* statements placed by designer. In contradiction to cycle-accurate designs, there are fully/partially untimed designs which require scheduling procedure, usually provided by SystemC HLS tools. During scheduling the HLS tool examines performance of the untimed code and automatically insert required number of states (registers) to meet timing requirements. So, current SVC tool cannot be applied to designs where automatic state insertion (scheduling) is required.

Existing HLS tools are commonly focused on behavioral modeling. In SystemC HLS tools elaboration-time programming is typically limited with SystemC synthesizable subset. SVC tool fully supports SystemC synthesizable subset for behavioral modeling and extends it with arbitrary C++ code at elaboration phase for writing hardware generators. SVC tool also supports *sc_vector* and some STL containers including *std::vector*, *std::array*, *std::pair*.

III. Design Elaboration

In SystemC, design hierarchy is created during elaboration phase by executing module constructors and phase callbacks. A tool like SVC does not have to convert elaboration code into equivalent Verilog code, instead it should reproduce SystemC design hierarchy in equivalent hierarchy of Verilog modules and primitives. In this section we present reflection based dynamic elaboration which extracts SystemC design structure directly from process memory. The extracted information includes module hierarchy, links from pointers to pointee objects, values of scalar types, sensitivity and reset lists for processes.

A. Overview of existing solutions

There are two well studied approaches how SystemC analysis tool can extract generated design hierarchy. Good review of existing solutions is given in [8] and [10]. First approach is to interpret C++ code of module constructors and build design hierarchy on fly, by analyzing calls of SystemC library functions. This idea is usually utilized by SystemC HLS tools. Benefit of this approach is that during interpretation SystemC analysis tool can detect some bugs, like out-of-bounds array access or reads of uninitialized memory. The drawback is that supported language subset is usually limited and use of libraries without source code is not allowed.

Second approach is extracting design hierarchy directly from process memory, after executing elaboration phase in SystemC application. For SystemC this approach was pioneered by Pinapa [8], and later used in other SystemC analysis tools [5], [6], [11]. Since C++ does not provide native API for runtime reflection, all these tools utilize various compiler intermediate representations as a source for reflection metadata.

In SVC we combine both solutions. We use reflection-based approach for design hierarchy extraction, as presented in next subsection and interpretation approach for analyses of process bodies, as presented in Section IV. Novelty of our elaboration solution, comparing to previous work on reflection-based design elaboration, is in systematic handling of pointers and pointer-like objects created and assigned during elaboration phase. We support all kind of pointers, including pointers to array elements and pointers to dynamically allocated objects.

B. *Reflection-based elaboration*

SVC is implemented as a shared library, built on top of SystemC and Clang libraries. It substitutes SystemC library for linking and, instead of running simulation, it runs Verilog code generation after executing SystemC elaboration phase. SVC extracts generated design structure directly from process memory. Extracted information includes module hierarchy, links from pointers to pointee objects (including ports and channels), values of scalar types, sensitivity and reset lists for processes. Extracted information is sufficient to generate equivalent Verilog design hierarchy without process function code.

SVC tool relies on Clang AST to get information about types used in design. Clang provides information about type layout, but it has no convenient API to utilize this information to traverse objects in memory. To fill this gap, we have implemented a tiny reflection library on top of Clang AST. The elaboration algorithm works in two passes. On the first pass it recursively traverses design hierarchy in memory, starting from top-level module and builds a design tree. Second pass of algorithm iterates over all pointers and pointer-like objects discovered on first pass. For each pointer object it finds a pointee object inside the design tree.

The only missing piece is support for dynamic, heap-allocated objects. C++ leaves no trace of dynamic memory allocations, so out-of-the-box SVC is not able to distinguish between pointer to dynamically allocated object and dangling pointer. We have solved this problem by overriding operators *new* and *new[]*. This allows to trace all memory allocations that take place during elaboration.

IV. PROCESS CODE TRANSLATION

Process code translation consists of constant propagation analysis, used/defined variable analysis, design correctness checking, clocked thread code generation, and control flow analysis with code generation. Constant propagation results helps to determine number of loop iterations, eliminate dead code and others. Used/defined information for SystemC variables allows to split them to local variables and registers in generated code.

A. *Static code analysis*

Static code analysis includes constant propagation analysis and used/defined variable analysis. Constant propagation analysis extracts values of variables and propagates them through the process control flow graph (CFG). Used/defined variable analysis retrieves information about written and read variables.

The constant propagation analysis operates with simple memory representation where each object can have exact value or unknown value. It considers the following object types: variable, dynamically allocated object, array element. Value types include all object types and additionally: numeric literal, record object, SystemC channel, and unknown. Such object and value types allow to represent numerical variables, pointers, references, arrays, different kinds of records, and SystemC communication primitives. The analysis provides sound, but not complete results. If an object has unknown value that means it can have any value of the object type. The analysis evaluates condition expressions of *if*, *switch*, loops and ternary operators which can be true, false or unknown. An evaluated condition has exact value, if the value is the same for all loop iterations, or unknown value otherwise. Function body is analyzed separately for each function call context, so the analysis is context sensitive.

The analysis algorithm works with memory state that represents values of SystemC objects in the particular statement and function call context. The algorithm starts at process function with memory state that contains values for global and module member objects, and no values for local variables. It updates memory state at each statement, adding and removing values. On control statement with unknown condition state is copied to the branches and in node with several predecessors input states are joined with set union. Having array element type which contains array object and offset, it is possible to support multi-dimensional arrays, pointers to array elements and pointer arithmetic. More details about static analysis algorithms can be found in [7], [9].

B. *Design correctness checking*

SystemC kernel does some design correctness checks during elaboration and simulation. Additional checks required for process function code to comply it with SystemC synthesizable standard. SVC tool reports of incorrect

or non-synthesizable statements with clear and expressive error messages that reduces designer efforts. SVC tool is checking for the following violations:

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

* Normally, latch is not desired in generated Verilog, but can be required in some cases. To support latches there is special function *sct_assert_latch()* which suppress error reporting for given variable.

C. Clocked thread code generation

Combinational method process is directly translated into *always_comb* in Verilog. All the variables declared in a combinational method are translated into local variables in *always_comb* block. As clocked thread process in SystemC can have multiple states implicitly specified with *wait()* calls, it cannot be directly translated to *always_ff*. To translate semantics of multi-state thread into a form which is accepted by most Verilog tools, SVC converts the thread into pair of *always_comb* and *always_ff* blocks. *always_ff* block implements reset and update logic for state registers. *always_comb* block contains combinational logic that computes the next state. Listings 1 and 2 shows translation of SystemC thread into pair of Verilog always blocks.

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. A main case branch corresponds 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.

always_comb contains all the logic of the SystemC thread and provides combinational outputs which are stored into registers in the *always_ff* block. All the variables used in the thread can be divided into two groups: local variables that are always assigned before use, and register variables that can retain their value from previous clock cycle. For a register variable two Verilog variables are generated, first represents the register, and second is used inside *always_comb* to compute the next register value. Listing 2 contains register variable *x* which is translated to pair of *x* and *x_next* in Verilog code, and *y*, which becomes local variable in *always_comb*.

```
void simple_thread() {
   sc_uint<3> x = 0;
   data_out = 1;              // write output signal
   wait();                    // STATE 0
   while (true) {
      sc_uint<2> y = data_in; // read input signal
      x = y + 1;
      wait();                 // STATE 1
      data_out = x;           // write output signal
}}
```

Listing 1. Simple clocked thread process

```
logic[2:0] x, x_next;
logic PROC_STATE, PROC_STATE_next;
always_comb simple_thread;
function void simple_thread;
   logic[1:0] y;
   data_out_next = data_out;
   x_next = x; PROC_STATE_next = PROC_STATE;
   case (PROC_STATE)
   0: begin
      y = data_in; x_next = y + 1;
```

```
      PROC_STATE_next = 1; return;
   end
   1: begin
      data_out = x_next;
      y = data_in; x_next = y + 1;
      PROC_STATE_next = 1; return;
   end
   endcase
endfunction

always_ff @(posedge clk or negedge rstn)
begin
   if ( ~rstn ) begin
      x <= 0; data_out <= 1; PROC_STATE <= 0;
   end else begin
      x <= x_next; data_out <= data_out_next; PROC_STATE <= PROC_STATE_next;
   end
end
```

Listing 2. Simple clocked thread generated Verilog

D. *Control flow analysis and code generation*

SVC runs control flow analysis and code generation for each method process, starting from function entry, and for each clocked thread state, starting at *wait()* call. The analysis proceeds from one CFG node to its successor nodes considering branching statement condition computed during in the static analysis pass to skip dead branches. For each loop only one iteration is traversed to generate statements in the loop body.

The flow control statements like *if*, *switch* and loops without *wait()* calls are converted into equivalent Verilog statements with minimal changes. Loops with *wait()* calls are divided into several states. Loop statement in that case is replaced by *if* statement in the generated Verilog. If such a loop contains *break* and *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. Listing 3 and 4 contains *while* loop with *break*, which is replaced with code marked with *break begin* and *break end* comments.

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

Listing 3. While loop with break process

```
function void thread_break;
   case (PROC_STATE)
   0: begin
      PROC_STATE = 1; return;
   end
   1: begin
      if (!enabled) begin
         if (stop) begin
            // break begin
            ready = 0;
            PROC_STATE = 1; return;
            // break end
         end
         PROC_STATE = 2; return;
      end
      ready = 0;
```

```
        PROC_STATE = 1; return;
    end
    2: ...
    endcase
endfunction
```

Listing 4. While loop with break generated Verilog

E. Current limitations

Synthesizable subsets of Verilog and SystemC have much in common. Both of them have similar data types, control flow statements, arithmetical and logical operators. There are a few difference in SystemC and Verilog semantics which require some efforts to implement. First of all that is multiple *wait()* in SystemC clocked thread process. Clocked thread with multiple *wait()* generated as case with number of states that makes Verilog code less readable. Another one is loop with *wait()* statements, especially if it contains *break* and *continue* statements.

SVC substitutes function calls with function body in generated Verilog. That ensures the Verilog meets all the tool requirements, but leads to difficulties in support functions with multiple returns. If function has *return* statement followed by some code, it needs to be converted in form where code execution finishes at this *return*. Current SVC does not support function with multiple *return* statements.

Current version of SVC supports C++ fundamental types and SystemC integer types as it is specified in SystemC synthesizable standard. To extend SVC to float and fixed point types, mapping to one of existing component library can be used.

V. EVALUATION RESULTS

We have evaluated SVC tool on a bunch of industrial SystemC designs. Equivalence of generated Verilog to input SystemC design has been proven for method and thread process structures and control flow statements. That also has been verified with the SystemC testbench consists of more than 600 unit tests. Also we have checked the Verilog code for compatibility with several FPGA and ASIC logic synthesis tools.

Table 1 presents compilation times for memory subsystem designs developed by our team. Design *A* has two AMBA ports, 78 memory banks and DMA controller. Design *B* is implementation of L2 cache with 16 ports. Design *C* contains 6 different bus ports, 88 memory banks with ECC and external memory controller with cache. Design *D* is simplest one and contains one AMBA AHB port and 16 memory banks. Design *E* is very large memory subsystem with 5 AMBA ports and atomic access control. Designs *F* is memory subsystem with 32 memories with security features. Memory subsystem *G* contains 24 memories with high-performance requirements.

For these designs numbers of unique modules, number of unique method and thread processes, and generated Verilog Lines-of-Code (LoC) are presented. Numbers of module and process instances in designs are shown in parentheses. Compilation time is given for the single-thread implementation of SVC tool, running on Intel Core-i7 CPU. In general, compilation time is a function of design size, but the result for the design *B* is an exception. In this particular design, some processes have deeply nested loops that were more time-consuming for our constant propagation analysis to iterate through.

Table I. Synthesis time for memory designs

Design name	***Number of modules (instances)***	***Number of processes (instances)***	***Generated code, LoC***	***Compilation 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

Table 2 compares area (register and LUT count) and performance (frequency in MHz) of Verilog RTL generated with SVC and a 3rd party SystemC tool. These numbers have been obtained with FPGA synthesis flow. The 3rd party tool was configured to produce performance-optimal design. Difference in area is not more than 10%, for two designs SVC area is less, for another two 3rd party tool area is less. Difference in maximal frequency is pretty small for all designs except design *C*, where SVC generated Verilog is 40% faster. So, the QoR of Verilog generated by SVC tool are comparable with the reference tool results.

Table II. Area and performance comparison

Design name	**SVC tool**		**HLS tool**	
	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

VI. CONCLUSION

SystemC is a powerful alternative to established hardware description languages. With SVC tool we demonstrate how to build a fast SystemC to Verilog translator using Clang libraries. SVC has been applied to several SystemC memory designs used in silicon.

Working on SVC development we consider the following enhancements: automatic sensitivity lists generation, combinational handshake channels for clocked threads, ASIC memories support with APIs for power control and memory specific parameters, automated pipeline register. To avoid creating a SystemC dialect, we are planning to offer these enhancements for discussion inside Accellera SystemC standardization workgroups.

REFERENCES

[1] CLANG: a C language family frontend for LLVM. http://clang.llvm.org/.

[2] SystemC Synthesizable Subset Version 1.4.7. http://accellera.org/downloads/standards/systemc.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In Proceedings of the 49th Annual Design Automation Conference (DAC '12). ACM, New York, NY, USA, pp.1216–1225.

[4] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. 2010. Clash: Structural Descriptions of Synchronous Hardware Using Haskell. In 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, pp.714–721.

[5] Harry Broeders and René van Leuken. 2011. Extracting Behavior and Dynamically Generated Hierarchy from SystemC Models. In Proceedings of the 48th Design Automation Conference (DAC '11). ACM, New York, USA, pp.357–362.

[6] Kevin Marquet and Matthieu Moy. 2010. PinaVM: A systemC Front-end Based on an Executable Intermediate Representation. In Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT '10). ACM, New York, USA, pp.79–88.

[7] Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

[8] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. 2005. Pinapa:An Extraction Tool for SystemC Descriptions of Systems-on-a-chip. In Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05). ACM, New York, USA, pp.317–324.

[9] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. Foundations and Trends in Programming Languages 2, 1 (2015), pp.1–69.

[10] Jannis Stoppe and Rolf Drechsler. 2015. Analyzing SystemC Designs: SystemC Analysis Approaches for Varying Applications. Sensors 15, 5 (2015), pp.10399–10421.

[11] Jannis Stoppe, Robert Wille, and Rolf Drechsler. 2013. Data extraction from SystemC designs using debug symbols and the SystemC API. In IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2013, Natal, Brazil, 2013, pp.26–31.

[12] J. I. Villar, J. Juan, M. J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe. 2011. Python as a hardware description language: A case study. In 2011 VII Southern Conference on Programmable Logic (SPL), pp.117–122.