DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 2018

Hardware construction with SystemC

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

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

Abstract — **Hardware construction is a structural design approach that involves algorithmic generation of circuit netlist using high-level programming language. Languages that support this approach are known as hardware construction languages (HCL). Among the best-known HCLs are Chisel and Bluespec, adopted both in industry and in academia. On the other side, C++ and SystemC are known in context of high-level synthesis (HLS), a process that involves automatic transformation of untimed behavioral models into RTL. Clearly, these two approaches are complementary and can be potentially implemented in a single language and synthesis flow. We explored opportunity to bring hardware construction support to SystemC, identified missing pieces in library and tools, and implemented a robust elaboration tool that supports full power of C++ language for synthesizable designs.**

Keywords—SystemC;hardware construction; hardware generation; high-level synthesis;

I. INTRODUCTION

Structural hardware modeling is often perceived as a manual process of assembling and wiring a complete system from various building blocks. Those building blocks can be as primitive as NAND gates, or as complex as complete CPU cores, but structural coding mostly remains the same: for each module designer writes a code to instantiate a module and connect its IO ports. As a number of building blocks grows this process becomes very tedious and error-prone. That is why hardware designers often write scripts and other software tools to generate structural HDL code automatically from concise specifications. This works especially well when structure of generated hardware is regular and IO interfaces are standardized. Examples of such tools are arithmetic circuit generators (FFT generators for example) and SoC IP assembly tools. Importance of automatic circuit generation has been recognized and some hardware description languages have built-in support for this approach. Hardware generation capabilities are different in each language. Verilog and VHDL have very primitive support for circuit replication using generate statement. On the other side, Chisel [1] was specifically designed for circuit generation and is advertised as a hardware construction language (HCL) to emphasize this feature.

In this paper, we explore an opportunity to bring hardware construction support to SystemC language. SystemC, like many other hardware description languages can be used to describe both structural and behavioral aspects of electronic circuits. The execution of SystemC application consists of elaboration phase followed by simulation phase. Elaboration results in creation of module hierarchy, port bindings and process instances. Elaboration code consists of module constructors and elaboration phase callbacks [2]. Arbitrary C++ code can be executed during elaboration; every structural aspect of the design can be parameterized. Thus, SystemC has all abilities of hardware construction language. However, modern SystemC synthesis tools are mostly concerned about behavioral aspect of design and supported hardware construction features are very limited. In this paper, we identify missing pieces and show how to integrate hardware construction into SystemC synthesis flow.

Structure of the paper is the following:

- First, we introduce a small code example to explore hardware generation in SystemC.
- Then we discuss required support for hardware construction in synthesizable subset and library.
- Finally, we present a prototype of a tool that extracts generated netlist from SystemC executable model.

II. HARDWARE GENERATOR IN SYSTEMC: PRACTICAL EXAMPLE

To understand a field of hardware construction in SystemC let us consider a toy example. Suppose we need to generate a SoC subsystem that consists of multiple memory-mapped slave devices selected by address bus, as shown on Figure 1.

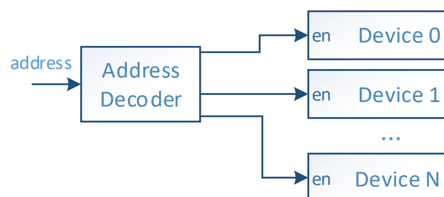

Figure 1 Generated system

We want system to be configurable by specifying number of devices, their types and address ranges in a text configuration file. Format of the file is the following: first line specifies number N - number of devices, each of next N lines contains device specification in the following order: start address, end address, device type, device name. Example of file is shown on Listing 1.

```
3
0x0000  0x0fff  apb_i2c   i2c_0
0x1000  0x1fff  apb_uart  uart_0
0x2000  0x28ff  apb_i2c   i2c_1
```

Listing 1 Configuration file example

We will implement a generator for this system in C++/SystemC. For this demonstration, exact functionality of each device is not important and they are implemented as black box modules with only a single input port - `en`. We will assume that there are only two possible device types: `apb_i2c` and `apb_uart`.

```
struct apb_i2c : sc_module {
    sc_in <bool>   en{"en"};
    apb_i2c(sc_module_name) {}
};

struct apb_uart : sc_module {
    sc_in <bool>   en{"en"};
    apb_uart(sc_module_name){}
};
```

Listing 2 Source code for blackbox device modules

A code for address decoder is shown on Listing 3. We represent address range of each slave device by `address_range` type (1). Module `address_decoder` (2) has input port for address (3) and vector of output ports `slave_select` (4). Since we do not know number of devices and their address ranges at compile-time, we need to pass this information during elaboration. Here we pass it in `addr_map` (5) parameter of module constructor. Size of `addr_map` defines number of `slave_select` ports (6), each vector element represents an address range for device. The module has a single SC_METHOD process that implements address decoding logic (8).

```
template <typename AddrT>
struct address_range {   (1)
    AddrT start_addr;
    AddrT end_addr;
};

template <typename AddrT>
struct address_decoder : sc_module {  (2)

    sc_in <AddrT>            address{"address"};  (3)
    sc_vector<sc_out<bool>>  slave_select{"slave_select"};  (4)
```

```
    address_decoder(sc_module_name,
                    std::vector <address_range<AddrT>> addr_map)  (5)
        : address_map(std::move(addr_map))
    {
        slave_select.init(address_map.size());   (6)

        SC_HAS_PROCESS(address_decoder);
        SC_METHOD(slave_select_method);
        sensitive << address;
    }

    const std::vector <address_range<AddrT>>  address_map; (7)

    void slave_select_method() {  (8)
        for (size_t i = 0; i < address_map.size(); ++i) {
            slave_select[i] = false;
            if (address >= address_map[i].start_addr &&
                address <= address_map[i].end_addr)
                slave_select[i] = true;
        }
    }
};
```

Listing 3 Source code for address decoder

Source code for top-level module `test_system` (9) shown on Listing 4. In that module we assemble a system of slave devices connected to address decoder. Module consist of address bus (10), vector of `slave_select` signals (11), dynamically allocated decoder module (12), and vector of dynamically allocated device modules (13). Actual process of hardware construction takes place in module constructor (14). First, we read number of slaves from file (15) and initialize `slave_select` vector size (16). Next, for each device (17) we read its type and address map from file (18), instantiate it, and bind `en` port (19), (20). Finally, we instantiate address_decoder and bind its ports (21).

```
template <typename AddrT>
struct test_system : sc_module {  (9)

    sc_signal<AddrT>              address{"address"};  (10)
    sc_vector<sc_signal<bool>>    slave_select{"slave_select"};  (11)
    address_decoder<AddrT>        *decoder;  (12)
    std::vector<sc_module*>       devices;  (13)

    test_system (sc_module_name, std::string config_file) {  (14)
        ifstream ifile(config_file);
        ifile >> hex;
        unsigned N_DEVICES;
        ifile >> N_DEVICES;  (15)
        slave_select.init(N_DEVICES);  (16)
        std::vector <address_range<AddrT>>  address_map;

        for (unsigned dev_id = 0; dev_id < N_DEVICES; ++dev_id) {  (17)
            unsigned start_addr, end_addr;
            std::string device_type, device_name;
            ifile >> start_addr >> end_addr
                  >> device_type >> device_name;  (18)
```

```
            if (device_type == "apb_uart") {  (19)
                apb_uart * adev = new apb_uart(device_name.c_str());
                adev->en(slave_select[dev_id]);
                devices.push_back(adev);
            }
            else {  (20)
                apb_i2c * bdev = new apb_i2c(device_name.c_str());
                bdev->en(slave_select[dev_id]);
                devices.push_back(bdev);
            }

            address_map.push_back({start_addr, end_addr});
        }

        decoder= new address_decoder<AddrT>("decoder",address_map);(21)
        decoder->address(address);
        for (size_t dev_id = 0; dev_id < N_DEVICES ; ++dev_id)
            decoder ->slave_select[dev_id](slave_select[dev_id]);
    }
};
```

Listing 4 Source code for top-level module

As we see from this example, arbitrary C++ code and libraries can be used to generate SystemC models. Next, we discuss what is required to make this code synthesizable.

III. HARDWARE CONSTRUCTION SUPPORT IN SYSTEMC LIBRARY AND SYNTHESIZABLE SUBSET

As we have shown in our example, structure of the design can be created dynamically during elaboration phase of execution. During this phase, module constructors and elaboration-time phase callbacks are called. Thus, hardware construction in SystemC is all about elaboration-time programming. SystemC synthesizable subset [3] however does not distinguish elaboration and simulation phases and imposes the same constraints on process functions and constructor code. In practice, the elaboration-time language subset offered by modern SystemC synthesis tools is often more limited than the language subset supported for processes. In our example we have used `vector`, `sc_vector` and file streams – none of them is currently supported by synthesizable subset.

To enable hardware construction in SystemC, elaboration and simulation phases should be clearly separated in synthesizable subset. During elaboration, arbitrary C++ code and libraries can be allowed, since we do not need to translate them into RTL. Only results of elaboration matter for synthesis.

Now consider interface between elaboration code and synthesizable processes code. This interface is represented by variables that are referenced from processes. The variables that are not used during simulation have no effect on behavior of system and structure of generated hardware. The only process in our example design is `address_decoder::slave_select_method` (8). This process is not synthesizable in terms of current synthesis subset because it references two dynamic data structures: `vector` for `address_map` (7) and `sc_vector` for `slave_select` (4). If however, we replace `address_map` with `static const` array and `slave_select` with constant-sized array like shown on Listing 5 this code becomes synthesizable. Now sizes of arrays and values of address ranges are compile-time constants. This is a common requirement for synthesis – all data structures in design should have a fixed size so they can be represented in generated hardware as arrays of modules, ports, fixed-sized RAMs or register files. If we want to generate ROMs or LUTs, all values should be compile-time constants.

```
sc_out<bool>  slave_select[3];
static const address_range<AddrT>  address_map[3] = {{0x0000, 0x0fff},..};
```

Listing 5 Synthesizable arrays for address decoder

If we look again at original source code we notice that while `address_map` and `slave_select` sizes are not compile-time constants, they are constant during simulation runtime. Once elaboration of the model finishes, sizes of these vectors cannot be changed. Values of `address_range` are also constant during simulation. This brings us the idea of elaboration-time constant – a value that is defined during elaboration, but constant during simulation. Such a value can be considered a constant for the purpose of hardware generation.

In our example we've used existing datatypes to model elaboration-time constness: we've used `sc_vector<>` that can be only initialized once with an `init()` method and `const std::vector<>` that can only be initialized in constructor member initializer list. While that worked for demonstration purposes, such an API can be too restrictive and inconvenient in practice. Instead, future SystemC standard and synthesis subset can consider modeling a notion of elaboration-time constness explicitly in code by introducing new datatypes. For example, we can define two new types: `sc_elab_const<T>` to model a value defined during elaboration and `sc_elab_vector<T>` to model a vector, which size is defined during elaboration. Variables of these types should be mutable during elaboration, for example, we can emplace new elements into `sc_elab_vector`. During simulation, any attempt to mutate them should trigger runtime error.

Another C++ feature that can be utilized for hardware construction is ability to define types of polymorphic objects at runtime. In our demo example, we define a type of each slave device during elaboration. Array of devices is modeled as `std::vector<sc_module*>`. Here `sc_module*` is a pointer, but raw pointers in C++ do not assume ownership of dynamically allocated object, instead we can use `std::unique_ptr<T>` from C++ standard library that has this semantics. If we combine it with `sc_elab_vector<T>` introduced in previous paragraph, we can express our design intent explicitly in code:

```
sc_elab_vector<std::unique_ptr<sc_module>> devices;
```

Here `devices` is a vector of modules, size of vector and type of each device is defined during elaboration.

This concludes our overview of features required to enable hardware construction in SystemC. Let us summarize them:

1. Synthesizable subset should distinguish elaboration and simulation phases of execution. Arbitrary C++ can be allowed during elaboration.
2. Interface between elaboration and simulation phases is represented by variables referenced from processes. Values, sizes and types of these variables can be defined during elaboration.
3. For the purpose of hardware construction synthesizable subset should support types that allow to express following semantics:
   a. Elaboration time constant
   b. Elaboration time sized vector
   c. Owning pointer for objects of polymorphic type.

In the next section, we explore a possible implementation of SystemC elaboration tool that will support all described concepts.

IV. DYNAMIC ELABORATION FOR SYSTEMC SYNTHESIS

Most of existing SystemC HLS tools use static analysis methods at elaboration phase. Usually they consume abstract syntax trees (ASTs) and control/data flow graphs (CDFGs) produced by C++ compiler front-end as shown on Figure 2. With static analysis, however, it is hard to support every C++ feature, so HLS tools impose restrictions

on supported language subset. Modern SystemC synthesis tools prohibit C++ standard library, have limited support for dynamic allocations and polymorphism. This limited language subset is simple enough for HLS tools to extract structural information by analyzing constructor bodies. In contrast, hardware construction languages usually utilize dynamic approach: they execute elaboration code and use code reflection information to extract generated design structure. We propose to combine dynamic and static approaches for SystemC synthesis: use dynamic approach for elaboration and static approach to handle behavioral code of process functions as shown on Figure 3. Such a hybrid approach to SystemC elaboration was already applied before by Pinapa [4] and PinaVM [5] SystemC front-ends.

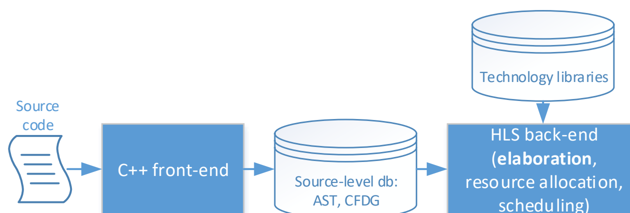

Figure 2 Structure of modern SystemC HLS tool

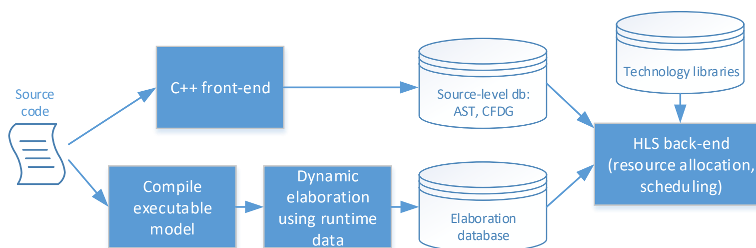

Figure 3 HLS flow with dynamic elaboration

Unlike many modern programming languages, C++ has no built-in support for runtime reflection. Instead, information about datatypes and their memory layout had to be obtained in some other way. Pinapa [4] obtains information about classes and data member offsets from GCC AST. Tool links with simulator executable to extract structure of design from memory. PinaVM [5] gets information about datatypes from LLVM IR. It utilizes JIT compilation to execute elaboration phase and sequences of instructions that compute data member offsets. Pinapa and PinaVM provide both structural and behavioral representations of design. Couple of research projects utilize debug information as a source for reflection. GDB-MI interface was used in [7] to extract SystemC design hierarchy. Windows PDB debug symbols were used in [7] to implement TLM model visualization.

Unfortunately, no one of existing dynamic SystemC elaboration tools is actively maintained. To experiment with hardware construction we have implemented our own tool to extract SystemC design structure from runtime data and generate elaboration database. In our tool, we use GDB debugger Python API [8], as it provides simple high-level interface to debug data. From this database, we generate complete Verilog netlist, but without `always@` process bodies. Unlike previous research in this area, our focus is solely on hardware generation. We believe that existing HLS tools can be easily enhanced by replacing static analysis of constructor bodies with a similar dynamic elaboration tool.

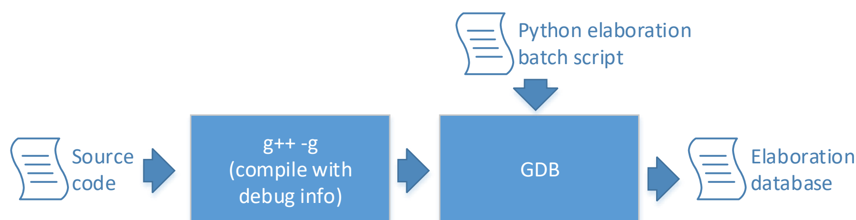

Figure 4 GDB-based SystemC elaboration flow

Next, we describe some implementation details of our Python script for GDB that extracts elaboration database for SystemC model.

Serialization format for elaboration database is based on Google Protocol buffers [9]. Protocol buffer compiler is a convenient tool that automatically generates binary serialization format and library for various programming languages based on a human-readable text schema. In our case, we generate database from Python and can consume it in C++ HLS tool.

Short outline of protocol buffers schema we use is shown on Listing 6. We hope that language is simple enough to be understood by unprepared reader. All type names used in design are stored in array (22), thus every type has a unique ID – type index in this array. We store messages describing all objects instantiated in design in another array (23), so every object has unique ID as well. There are multiple kinds of object messages in our schema, but we classify them into three groups (27): primitives, records and arrays. Primitives are types like integers, pointers and pointer-like objects (sc_ports and sc_exports). Records are collections of named fields. Arrays are collections of indexed elements. For each object we store its unique ID (24), ID of its type (25), and ID of its parent object (26) (that can be either array or record). Records and arrays store IDs of their child objects (28), (29). Top-level module has ID = 0. Elaboration database can be visualized like a tree, with top-level module as a root and primitive objects in leaves.

```
message SCDesign {
    repeated string types;    // list of C++ type names used in design (22)
    repeated Object objects;  // list of objects in design, objects[0] == Top module (23)
}

// Object in memory, member of design hierarchy
message Object {
    required uint32 id;       // Unique ID of object, starts from 0 (24)
    required uint32 type_id;  // Unique ID of object type name, starts from 0 (25)

    // ID of parent object.
    repeated uint32 parent_id; (26)

    enum ObjKind { (27)
        PRIMITIVE = 1; // Object is design "primitive", like integer value, port, pointer..
        RECORD = 2; // Object is struct or class, a container of data members
        ARRAY = 3;  // Object is array
    }

    required ObjKind kind; // object kind

    optional Primitive primitive;
    optional Record record;
    optional Array array;
    ...
}
message Record {
    repeated uint32 member_ids = 1; (28)
...
}
message Array {
    repeated uint32 element_ids = 3; (29)
...
}
...
// Raw pointer or port
message Pointer {
    optional uint32 pointee_id; // ID of pointee object (30)
}
//... other types of primitives and aggregates
```

Listing 6 Outline of protocol buffers schema used for serialization

Pointer and pointer-like primitives store an ID of their pointee (30) objects. This transforms our design tree into a directed graph.

Algorithm of Python GDB script implements following steps:

- Set breakpoint after elaboration phase, before start of simulation.
- Run executable, stop at breakpoint.
- Get pointer to top-level module from simulation context.
- Recursively traverse all design objects.
  - Create message descripting each object.
  - Put memory address range for every object into interval tree. This will be used later to resolve pointer-pointee links.
- For each pointer or pointer-like object in design find its pointee by address in interval tree.

One limitation of C++ is that it leaves no information for debugger about dynamic allocations (`operator new`). For example, if we write `int * p = new int[3]` debugger can't identify number of integers in array. Another issue is that raw pointers do not have ownership semantics, so we can point to same dynamic object from pointers in different SystemC modules. As a workaround for these issues, we impose following constraints on coding style:

1. Pointers should always model ownership. Multiple raw pointers to same dynamic object are not supported.
2. To model dynamically-sized arrays `std::vector` should be used instead of explicit `operator new[]`.

Reference Accellera SystemC kernel [10] deletes information about hierarchical bindings during elaboration. For example if we bind input port to another input port, information about those bindings will not be stored. This information however is important to preserve port names in generated RTL, so we modified SystemC kernel to retain this data.

To validate our elaboration tool we have used a set of HLS SystemC designs we have implemented in recent years. Those designs were written for commercial HLS tool and do not utilize elaboration-time programming features described in this paper. The largest of them is shared memory, a highly parameterizable memory hierarchy design template that supports on-chip and off-chip bus protocols, caching, various types of memory banks, multiple design topologies. Largest instance consists of 587 module instances and 27739 primitive objects. Those are relatively small numbers for RTL or gate-level designs, but in HLS world, such a design can be considered large. Runtime of Python GDB elaboration script is 14 minutes. For comparison, synthesis for this module in HLS tool usually takes about 3 hours.

Presented approach has couple of drawbacks. Some compilers do not produce DWARF debug information, for example Microsoft C++ compiler. Performance of elaboration will not be sufficient for low-level RTL and gate-level designs with millions of design primitives. During development, we also discovered that sometimes type names generated by recent g++ and clang compilers in DWARF debuginfo and mangled RTTI symbols do not match. As a result dynamic type identification in GDB does not work correctly. Most recent compiler that does not have this issue is g++ 6.4.

V. CONCLUSIONS

Bringing hardware construction to SystemC makes it a truly universal electronic system-level design language that supports all modern modeling styles. We have shown how dynamic elaboration can be applied for SystemC language. Our approach removes limitations on language subset supported during elaboration. Highly parameterizable designs, such as a bus fabrics and hierarchical memory subsystems can benefit a lot from proposed

approach, increase code reuse and eliminate a lot of manual, error-prone work. We hope that this approach can be adopted by HLS vendors, removing existing limitations in the established tools.

REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, “Chisel: Constructing hardware in a scala embedded language.” in Proceedings of the Design Automation Conference. ACM, 2012, pages 1216–1225

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

[3] SystemC Synthesis Subset 1.4.7 webpage. http://accellera.org/downloads/standards/systemc

[4] Matthieu Moy, Florence Maraninchi, Laurent Maillet-Contoz, “Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip” in EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software. ACM, 2005, pages 317-324

[5] Kevin Marquet, Matthieu Moy, “PinaVM: a systemC front-end based on an executable intermediate representation” in EMSOFT '10 Proceedings of the tenth ACM international conference on Embedded software. ACM, 2010, pages 79-88

[6] H. Broeders and R. van Leuken. “Extracting behavior and dynamically generated hierarchy from SystemC models” in Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE, pages 357 –362, June 2011

[7] Jannis Stoppe, Robert Wille, and Rolf Drechsler. “Data extraction from SystemC designs using debug symbols and the SystemC API” in VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on, pages 26–31. IEEE, 2013

[8] GDB Python API webpage. https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html

[9] Protocol Buffers webpage. https://developers.google.com/protocol-buffers/

[10] Accellera reference SystemC implementation webpage http://www.accellera.org/downloads/standards/systemc