CINFRA: A Quick System Modeling Approach

Vladimir Ivanov
State University of Aerospace Instrumentation
St. Petersburg, Russia
vladimir.ivanov@ieee.org

Swaminathan Ramachandran
CircuitSutra Technologies Pvt. Ltd.
Bengaluru, India
r.swaminathan@circuitsutra.com

Abstract—**In this paper we deal with System-on-Chip (SoC) system (transaction and higher abstraction) level modeling. The traditional approach based on libraries of custom IPs and SystemC/TLM-2.0 framework is saddled with issues, which prevents wider dissemination of system modeling technology in the industry. Incompatible interfaces of custom IPs give rise to integration issues. Current SystemC/TLM-2.0 framework requires highly skilled developers having software and hardware expertise. We propose a new approach that looks promising and overcomes the above-mentioned drawbacks. The essence of our approach is an infrastructure library that hides the complexity of SystemC. System level models are assembled from infrastructure elements with functional cores expressing hardware capabilities. Implementation results and proof-of-concept are presented.**

I. Introduction

With ever-increasing complexity, SoC requires adequate means and tools for modeling. Means includes models and languages for system specifications. One of the most popular means is SystemC [1], [2]. SystemC is a system level modeling language, based on C++ that has gained a lot of traction in the fields of system level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis. It is a set of C++ classes and macros, which provide an event-driven simulation interface enabling a designer to simulate concurrent processes. SystemC processes can communicate in a simulated real-time environment, using signals of any (built-in or user-defined) data types.

Transaction level modeling in SystemC involves communication between SystemC processes using function calls. TLM-2.0 focuses on modeling of on-chip memory-mapped buses. TLM-2.0 has a layered structure, with the lower layers being more flexible and general, and the upper layers being specific to bus modeling. TLM-2.0 has kicked off a thriving third party ecosystem for development of reusable and standard IPs that can be shared across teams and companies.

The flexibility and generality of C++ in modeling hardware across multiple levels of abstractions and desired cycle accuracy, via templates and libraries, has some downsides. The hardware engineers and system architects are put off by the syntactic complexity (template heavy) and debug (into library internals) of SystemC. There is a noticeable amount of boilerplate code that needs to be written to handle common scenarios. In addition, the tool vendors have made limited investment into the language, whose reference implementation is available from Accellera site [2].

Tools provide instruments for model development including IDEs, which aggregate GUIs, libraries, compilers, linkers and execution environment or, in case of SystemC, simulation engine. Synopsys "Platform Architect MCO"© represents a typical tool in this area. The tool provides an environment for SystemC/TLM-2.0 models' development at different levels of accuracy (untimed, loosely timed, cycle accurate). It gives a powerful means for system architecture performance optimization including power optimization. One of the important parts of such tools is the library of predesigned elements. Libraries usually include typical hardware elements such as clock generators, registers, buses etc.

Despite the powerful means and tools in system model development, there are substantial drawbacks as listed below, which prevents its wider dissemination in the industry:

- Semantic complexity of SystemC/TLM-2.0 for hardware engineers and as a result, system models are typically developed by highly trained software developers who must have both hardware and software expertise;
- Long development cycles for model development despite hardware libraries usage, due to a labor-intensive development process, testing and verification;
- Integration of IPs taken from hardware libraries is not straightforward due to rigid interfaces. Quite often, selected IPs bring the required functionality, but due to incompatible interfaces they cannot be integrated in a model project;
- Software projects are isolated within on-shelf tools. Usually commercial tools do not export system model program codes outside their tool environments.

We have attempted to address all these issues in this paper. In the following sections, we describe a system modeling methodology, which dramatically simplifies and speeds up system model development, while at the same time maintains the requested level of accuracy and simulation quality.

The paper consists of five sections. After the current introduction, the essence of the proposed methodology is described in Section II. Section III presents generic CINFRA library, which is a simple library of infrastructure elements and templates. This library is the cornerstone of the proposed methodology and allows developing system models with any level of accuracy and complexity. We present some implementation results in Section IV and conclusions in Section V.

II. Proposed system modeling methodology

A. Model declarative specification

Usually a “system model” of hardware assumes a complex entity consisting of many interacting elements, which in turn can also be treated as a system. Such hierarchical decomposition can be continued until some level where all elements are atomic and non-decomposable. The term “model” exactly reflects the level of decomposition (abstraction) where we want to stop decomposition. For example, in Register Transfer Level, the model includes registers, digital signals and logical operations carried out on those signals. More abstract Transaction Level Model (TLM) hides communication details, and instead of signals considers transactions for data transfer. Below, in the paper, “system model” will be assumed to be of TLM or higher abstraction models. “System model” term will be used for such types of models.

We specify hardware models as formal algorithms following [3]. Nevertheless, due to hardware nature, it needs to be kept in mind that it should be a declarative specification. The basis of the system modeling methodology is an adequate underlying model of computation (MoC) [4]. Description of system hardware models based on MoC brings us at least two advantages, which are vital for complex systems:

- mathematically rigorous hardware elements behavior and their interactions;
- ability for formal verification.

Hardware components of TLM models are reactive. It means that a corresponding component starts work when it receives all input data. The component generates output data after processing all input data. This reactive behavior motivates us to use the dataflow model of computations (Dataflow Network Processes, [3], Section B.4) for specification of system models. Let us also note that this specification is purely declarative.

Fig. 1 shows an example of a dataflow model consisting of two terminal operators A, B and data 1, 2, 3, 4 among them. An operator A starts work (or *fires*) when both input data 1 and 2 are full. The operator A generates an output data 3. The data 3 is an input data for an operator B. Thereafter, the full data 3 enforces firing the operator B that, in its turn, generates an output data 4.

Operators A and B might be not terminal but rather complex operators. In such cases, they play role of wrappers for internal terminal operators and their firing is defined by internal components of corresponding operators.

Our strategy in building the system modeling methodology is the following.

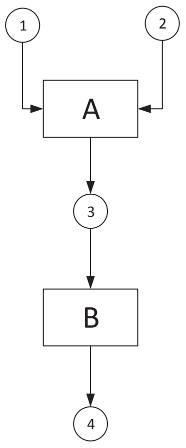

Fig. 1. An example of dataflow model

Keeping in mind the mentioned dataflow MoC, we are going to use SystemC to build infrastructure templates and objects, which will behave according to this MoC and provide related hardware functionality. TLM-2.0 supplies the standard communication capabilities. One of the obstacles on the way is the perceived complexity of SystemC/TLM-2.0. While SystemC has found a home as a system level modeling language, it still has work to do to endear itself to people coming from non-software background. One way to deal with this is to provide a framework that they are already familiar with and tucks away the complexity until they need to deal with it. What is the familiar paradigm? Passing input/output values as function parameters and capturing the processing logic within the function. This provides a simple mind-map for consuming input values, processing it and generating output.

The framework also takes care of providing the appropriate input/output SystemC ports and TLM memory buses to connect with other SystemC/TLM-2.0 compliant models out of the box. The framework provides this through a simple declarative mechanism, which captures the intent, and takes care of transforming the same to corresponding implementation at compile time.

We introduce three types of infrastructure templates for building system models.

- COMPONENT – terminal element bringing hardware functionality; it is atomic and the functional core is executed when all input values are presented.
- BLOCK – complex element consisting of COMPONENTs and other BLOCKs; it does not have its own functionality; all block functionality is expressed in components.
- SUBSYSTEM – complex element consisting of COMPONENTs and BLOCKs with specific communication capabilities, it reflects a complex cluster of hardware elements.

General methodology of system model development consists of the following actions.

- Specification of the model as a dataflow model in terms of operators, complex operators and data flow amongst them.
- Mapping operators onto COMPONENTs and complex operators onto BLOCKs or SUBSYSTEMS.
- Development of functional cores for corresponding COMPONENTs or referring to functions from an existing functional library.
- Writing program code by declaration of model COMPONENTs, BLOCKs, SUBSYSTEMs and elements from CINFRA library.
- Binding declared modules by using corresponding macros from CINFRA framework or specification of a text netlist.
- Testing.

B. *Component template*

COMPONENT is a sc_module declaration. It shown in Fig. 2. The Component is the basic unit to capture the hardware functionality. The Component, by default, awaits inputs on all its input ports, and then fires the functional core while passing all the input data as input Params<data-type>&… to the function. The functional core then executes the functionality and populates the output in Params<data_type>&… which is then forwarded to the corresponding output ports. Params<> may be thought of as a vector/map which has a 1-1 correspondence to its corresponding port. Execution (and pipeline) delay can also be (optionally) inserted after the functional core execution.

For e.g. if we need to model a simple Max2 Component with a reset/in, two inputs int/in and one output int/out, it may be declared as follows:

```
   // Simple functional core to calculate max
   void max2(Params<int> &in, Params<int> &out) {
         out[0] = (in[0] > in[1]? in[0]: in[1]);
   }
   // Max2 Component
   COMPONENT (Max2, PORT (rst_t, 1, IN), PORT (int, 2,
IN), PORT (int, 1, OUT)) {
         COMPONENT_CTOR (Max2) {
                 // Associate this Component with Max2
                 // functional core
                 this->set_func_core(&max2);
         }
   };
```

The communication between Components is standardized and this allows for flexible exchange of information based on the need.

Each port is declared as a triad – PORT (data-type, size, IN/OUT), where

- data-type is any C++ built-in type, SystemC type or User-defined data type.
- block(data-type) changes the port to a blocking port-type from the default non-blocking port type.
- evt_t is a data-less port type that signals events.

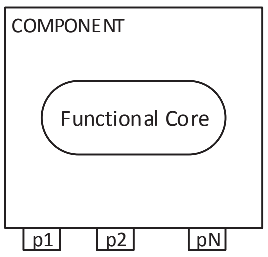

Fig. 2. A COMPONENT template

- wr_tlm_t & rd_tlm_t is a stand-in for TLM2.0 write/read sockets.
- size is the number of ports, default size is 1.
- IN/OUT determines the direction of the port. INOUT is not supported now.

A Component or Block supports all PORT types, whereas a SubSystem supports a subset of PORT types.

Callbacks are associated with each PORT/IN type to handle the incoming signals. Drive functions are associated with each PORT/OUT type. Default implementation is provided, which users can override, if needed.

C. *BLOCK template*

BLOCK is a sc_module declaration. Fig. 3 illustrates BLOCK concept.

The Block is a structural mechanism to put together simpler Components/Blocks to model more complex logic. BIND() is a function that is used to dynamically discover and connect the given pins, instantiating a channel in between as needed. The Block, unlike the Component, can execute operations when only a part of data are available for computations. For this purpose, we use a specific component - mux2, having two inputs and one output. The mux2 executes data time-division multiplexing and is triggered by only one input. A dataflow model of mux2 is shown in Fig. 5.

The following example illustrates the Block concept. Let us suppose we need to model a Max4 Block (using 3 simple Max2 Components) we could declare and bind it as follows:

```
   // Max4 Block
   BLOCK (Max4, PORT (rst_t, 1, IN), PORT (int, 4, IN),
   PORT (int, 1, OUT)) {
         // Declare 3 Max2 Components.
         Max2 m0, m1, m2;

         BLOCK_CTOR(Max4)
         : m0(“m0”), m1(“m1”), m2(“m2”)
         {
         BIND(“rst_0”, “m0.rst_0, m1.rst_0, m2.rst_0”);
         BIND(“in_0”, “m0.in_0”);
         BIND(“in_1”, “m0.in_1”);
         BIND(“in_2”, “m1.in_0”);
         BIND(“in_3”, “m1.in_1”);
         BIND(“m0.out_0”, “m2.in_0”);
         BIND(“m1.out_0”, “m2.in_1”);
         BIND(“m2.out_0”, “out_0”);
         }
   };
```

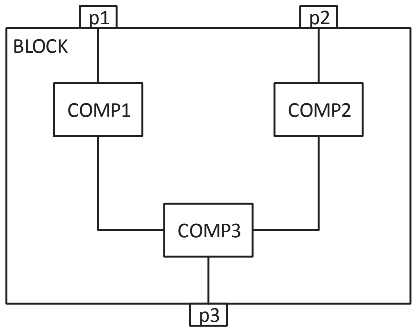

Fig. 3. An example of a BLOCK template

D. SUBSYSTEM template

SUBSYSTEM is a sc_module declaration. Fig. 4 illustrates SUBSYSTEM concept.

A SubSystem is a structural element like a Block except that it has restrictions on the kind of ports it permits at the boundaries (only Reset, Clock and Tlm ports permitted).

For e.g. if we would like to support a pipeline operation on a given input TLM stream, we could do it as follows:

```
SUBSYSTEM (Image, PORT (rst_t, 1, IN), PORT (wr_tlm_t, 1,
IN), PORT (wr_tlm_t, 1, OUT)) {
        PipeOp0 p1;
        PipeOp1 p2;
        PipeOp2 p3;

        SUBSYSTEM_CTOR (Image)
        : p0("p0"), p1("p1"), p2("p2")
        {
        BIND("rst_0", "p0.rst_0, p1.rst_0, p2.rst_0");
        BIND("in_0", "p0.in_0");
        BIND("p0.out_0, p1.in_0");
        BIND("p1.out_0", "p2.in_0");
        BIND("p2.out_0", "out_0");
        }
};
```

E. Communications

The Communication and Computation part of the Components are strictly separated as it was described in [5]. The framework provides a standard set of communication mechanisms for notifications and exchanging information. They are listed as follows:

1) evt_t: data-less notifications.

2) rst_t: Specialized evt_t with custom behavior to reset the module to init_state.

3) pos_clk_t/neg_clk_t: Clock types triggered at positive and negative edges.

4) T: In-built or user defined data type that is implemented using sc_in/sc_out ports. This provides non-blocking communication interface.

5) block(T): Like T, except that the transaction blocks unless there is space to write/data to read. This may be used for flow-control.

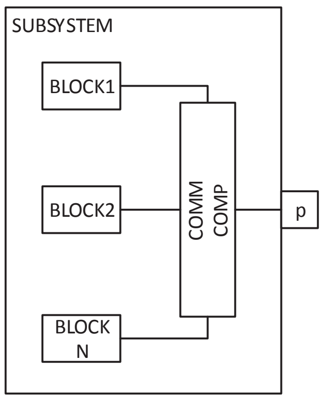

Fig. 4. An example of a SUBSYSTEM template

1. wr_tlm_t/rd_tlm_t: Write/Read TLM sockets. Write and Read TLM transactions are handled separately for ease of modeling. tlm_generic_payload is wrapped with a simple structure (mreq_t) to hide its complexity.

Currently INOUT transactions and TLM/Debug transactions are not supported.

III. CINFRA LIBRARY

A. Infrastructure elements

Many infrastructure-plumbing elements were developed to enable easy connectivity of Components. Some of them are listed below:

- mux2/fmux – different multiplexers: mux2, multiplexer with 2 input ports and 1 output port, fmux, configurable multiplexer with variable number of input ports and user defined functional core;
- dmux2/fdmux – different demultiplexers: dmux2, demultiplexer with 1 input port and 2 output ports, fdmux, configurable demultiplexer with variable number of output ports and user defined functional core;
- fwd – forwarder copies data from the input port to configurable number of output ports;

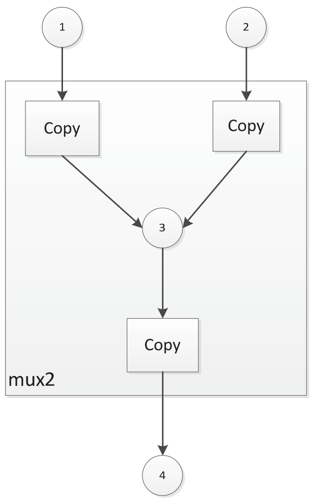

Fig. 5. A dataflow model of the multiplexer mux2

- sel – selector selects data from the input port according to a user defined rule in the functional core and copies selected data to the output port;
- splitter – splitter routes data from configurable number of input ports to configurable number of output ports according to user defined rules;
- terminator – terminator terminates input or output ports;
- const – constant value generator.

B. *Declarative syntax*

The declarative syntax is enabled using C++ Template Meta Programming. Some of the important elements in it are described below.

1. ToPort: This is used to convert a duet of data-type and direction to a SystemC port of appropriate type using template partial specialization.

```
template<typename T, direction_t D>
struct ToPort {
using type = sc_core::sc_out<T>;
};
template<typename T>
struct ToPort <T, IN> {
using type = sc_core::sc_in<T>;
};
template<>
struct ToPort <evt_t, IN> {
using type = sc_core::sc_in<bool>;
};
template<>
struct ToPort <evt_t, OUT> {
using type = sc_core::sc_out<bool>;
};
```

2. Port: This is used to capture the user defined Port declarations and communicate it to the underlying class.

```
template<typename T, size_t N, direction_t D>
struct Port {
using port_data_type = T;
static const size_t port_size = N;
static const direction_t port_dir = D;
};
```

3. Variadic template base class to capture port declarations. This information is then used to declare SystemC ports, instantiate corresponding port-buffers, evaluate trigger condition for FunctionalCore, generate pre-callbacks for ports etc.

```
template<typename FunctionalCore, typename ...Ts>
struct Component
: public sc_core::sc_module {
// See how declarative syntax can be used to drive
// SystemC port declarations
std::tuple< sc_core::sc_vector< typename ToPort<typename
Ts::port_type, Ts::port_dir>::type >... > ports;
/* … */
}
```

C. *Common extensible core framework in C++11*

The framework provides a simple and consistent way to declare a Component/Block/SubSystem. The heavy lifting of transforming this declarative structure to C++ code is accomplished using the C++ Variadic Template feature and Template Meta Programming. Component<>, Block<> and SubSystem<> classes are defined as generic base classes with mandated functionality. Each user-defined COMPONENT/BLOCK/SUBSYSTEM inherits this functionality and expands on it as required. Since the bulk of the functionality is already captured and well-tested in the base classes, the amount of code to be written by the user is drastically reduced (order of 2x-10x).

D. *Text based netlist connectivity and configuration*

The framework supports dynamic discovery of SystemC port types and has knowledge of how to connect common port types (and instantiate appropriate channels, if needed). The user can extend this for user-defined types as well. While the dynamic discovery may slow down the init-time of the system, this is miniscule compared to the runtime of a typical simulation. In addition, this helps cut down the compilation times by changing fix-recompile-run to fix-run cycles, for netlist connectivity errors.

Hierarchical (XML-based) configuration support is provided for initializing the init-time parameters of Components/Blocks/SubSystems. These can help to override the default values of the base classes to suit the SoC being constructed. When SystemC Configuration, Control and Interface (CCI) [7] is released, this may be updated to CCI standard without much ado.

```
<system>
    <module name="t"> <!-- top -->
        <module name="tb"> <!-- testbench -->
            <attribute   name="tv"   value="10,20,60,9,-
1,16"/> <!-- testvector -->
        </module>
        <module name="adder_0">
            <attribute  name="port_names"  value="rst_in,
int_in, int_out"/>
            <attribute name="delay" value="10 ns"/>
        </module>
        <module name="adder_1" clone="adder_0"/>
        <module name="adder_2" clone="adder_0">
            <attribute name="delay" value="15 ns"/>
        </module>
        <attribute name="connect"
            value="tb.rst, adder_0.rst_in_0,
            tb.rst1, adder_1.rst_in_0,
            tb.rst2, adder_2.rst_in_0,
            tb.a,  adder_0.int_in_0,
            tb.b,  adder_0.int_in_1,
            tb.A,  adder_1.int_in_0,
            tb.B,  adder_1.int_in_1,
            adder_0.int_out_0, adder_2.int_in_0,
            adder_1.int_out_0, adder_2.int_in_1,
            adder_2.int_out_0, tb.c"/>
    </module>
</system>
```

IV. IMPLEMENTATION RESULTS

A. *CINFRA library implementation*

The CINFRA library is developed using standard C++11 features and uses the Boost library for XML config parsing. The code has been compiled using both g++ and clang++ compilers on Windows and Linux (64-bit). The complete source code is only a few kLOC; as it leverages the SystemC/TLM2.0 infrastructure in full.

B. System Cache Model

The CINFRA framework was used for modeling a System Level Cache which is one of the important parts of shared-memory System-on-Chip (SoC) architectures (see for example [6], Chapter 5.2). The System Level Cache (SLC) defines memory access performance of application processors in most smartphones. The functional and loosely timed SLC models (TLM) are described by the same program code, with the difference in time attributes. Time attribute for all model components is equal to zero in the functional model and they are non-zero in the loosely timed model. Metrics characterizing CINFRA efficiency for this case is shown in Table I. In the table, we compare SLC development based on SystemC/TLM-2.0 only and with CINFRA framework. We ported SystemC/TLM-2.0 SLC model into Synopsys Platform Architect© environment and compared simulation speed of ported model and model in CINFRA framework.

TABLE I. EFFICIENCY METRICS

Metrics	Value
Model development time	reduced by ~ 50%
Code length	reduced 40% - 70%
Simulation speed	increased 2 - 8 times

We did not use IP libraries for model development. CINFRA methodology requires only functional library development. Functions from this library are responsible for implementation of core hardware functionality. Component functional cores refer to such functions. In case of the SLC, the functional library includes program codes related to the cache operations. Functions can be developed in C or C++ style. Only new functions require unit testing. Other model elements are taken from well-tested CINFRA library and does not require unit testing at all. It substantially reduces testing efforts.

V. CONCLUSION

We proposed in this paper a system modeling methodology founded on algorithmic specification of a complex hardware system. It provides us with an ability to use formal verification of system models and substantial simplification of development process. Simplification is approached by development of the specific CINFRA framework that includes three basic templates and infrastructure elements allowing us to build system models with any level of complexity and accuracy. CINFRA framework is created just using pure C++11. It allows escaping to script languages for system models programming and significantly increases simulation speed of resulting models. The basic templates annotated by indexes (delay, area, power consumption etc.) supports development of performance models or models for architecture explorations. From a practical point of view, a process of system model building consists of three steps: system specification as a dataflow model, development of functional cores expressing elementary hardware functionality and mapping the dataflow model onto CINFRA framework.

CINFRA framework does not have issues related to IPs integration due to incompatible interfaces. Hardware functionality is described in functional cores that can be bound with any interface parameterized in CINFRA templates.

The proposed methodology was applied for development of the System Cache TLM model and has demonstrated its efficiency first in terms of time-to-market.

ACKNOWLEDGMENT

The research leading to these results was supported by the Ministry of Education and Science of the Russian Federation grant 8.4048.2017/4.6 from 31.05.2017.

REFERENCES

[1] D. C. Black, J. Donovan, B. Bunton and A. Keist, *SystemC: From the Ground Up, Second Edition*. New York: Springer, 2010.

[2] Accelera Systems Initiative™ official website, SystemC 2.3.1 (includes TLM), 03.11.2016, Web: http://www. accellera.org/downloads/standards/systemc.

[3] S. Edwards, L. Lavagno, E. A. Lee and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation and Synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, 1997, pp. 366-390.

[4] S. Marco, L. Lavagno and A. Sangiovanni-Vincentelli, "Formal Models for Embedded System Design," *IEEE Design & Test,* vol. 17, no. 2, 2000, pp. 14-27.

[5] CircuitSutra Technologies Pvt Ltd official website, STARC Complient Model, Web: http://www.circuitsutra.com/uploads/3/7/0/9/37098571/starc_compliant_models.pdf.

[6] J. L. Hennesy, D. A. Patterson, *Computer Architecture, A Quantitative Approach, Third Edition*. New York: Elsevier Science, 2003.

[7] Accelera Systems Initiative™ official website, SystemC CCI, Web: http://www.accellera.org/activities/working-groups/systemc-cci.