Automatic Generation of SystemC Models from Component-based Designs for Early Design Validation and Performance Analysis

Zhonglei Wang
Lehrstuhl für Integrierte Systeme
Technische Universität München
Arcisstraße 21, 80290 München, Germany
Zhonglei.Wang@tum.de

Wolfgang Haberl, Stefan Kugele
Institut für Informatik
Technische Universität München
Boltzmannstraße 3, 85748 Garching, Germany
{haberl, kugele}@in.tum.de

Michael Tautschnig
Institut für Informatik
Technische Universität Darmstadt
Hochschulstraße 10, 64289 Darmstadt, Germany
tautschnig@cs.tu-darmstadt.de

ABSTRACT

In this paper we present an approach of generating SystemC executable models from software designs captured in a new component-based modeling language, COLA, which follows the paradigm of synchronous dataflow. COLA has rigorous semantics and specification mechanisms. Due to its well-founded semantics, it is possible to establish an integrated development process, the artifacts of which can be formally reasoned about and are dealt with in automated tools such as model checkers and code generators. However, the resulting models remain abstract and cannot be executed immediately. Therefor SystemC offers executable models of a component-based flavor. Establishing an automated translation procedure from COLA to SystemC thus allows for design validation and performance analysis during early design phases. We have validated our approach on a case study taken from the automotive domain.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Performance, Design, Languages

Keywords

COLA, Code Generation, Simulation, SystemC

1. INTRODUCTION

System features based on software are becoming the most important factor in system designs in many branches of industry, such as automotive, avionics and others. With the ever-growing importance and complexity of embedded control systems, this trend places new demands on design methodologies. In this context the component-based approach to software and system designs has been established as the most appropriate approach to tackle the challenge.

COLA [6], the Component Language, is a component-based modeling language especially targeted at embedded control systems. It offers a graphical representation to specify the functional behavior of the modeled application. Being a synchronous formalism, COLA follows the *hypothesis of perfect synchrony* [1]. Basically, this asserts that computations and communication take no time. Components in a synchronous dataflow language then operate in parallel, processing input and output signals at discrete instants of time. This discrete uniform time-base allows for a deterministic description of concurrency by abstracting from concrete implementation details, such as physical bus communication, or signal jitter. To take the resulting models to the designated hardware platform, automated code generation is supported by our established toolchain.

Still, not only execution on the target hardware is sought after, but also *model-level simulation* plays an important role in the design process. As we know, specification mistakes made in the first phase of the development cycle are quite frequent. These specification mistakes and the errors that occur during the development are the most expensive things to handle, if they can be identified only after the generated code run on the target hardware. Therefore, we aim at a simulation framework for rapid prototyping, early design validation and performance analysis. SystemC is the one that appears to be the most suitable for our purpose. As an SLDL (system level design language) SystemC has become a standard in system level design [4]. It allows for efficient modeling of both hardware and software and enables simulation of the entire system including its communication architecture [5, 7, 9].

Since SystemC matches the common syntactic elements of many component-based modeling languages, including components, ports and hierarchical composition, SystemC code may be generated automatically from COLA models. The generated SystemC models can also match the COLA semantics described in [6]. Thus, the two modeling approaches

WOSP'08, June 24–26, 2008, Princeton, New Jersey, USA.

are well incorporated to establish an integral development process, which benefits from the advantages of the both approaches.

The rest of the paper is organized as follows: Section 2 gives an overview of related work. Section 3 gives a brief introduction to COLA. Following this, Section 4 contains a detailed description of the translation from COLA to SystemC. Section 5 presents the ways to perform simulation using SystemC. In Section 6 a case study from the automotive domain is considered to validate the proposed approach. Section 7 concludes this paper.

2. RELATED WORK

To achieve time-to-market reduction and meet various safety requirements, model driven development processes dominate the embedded control system industry. There are dozens of modeling languages developed in either industry or academia. The Unified Modeling Language (UML) and MATLAB/Simulink have become industrial standards. Both modeling languages provide graphical representation and include means for component-based modeling. The modeling concepts of COLA are similar to them, but with a rigorous formal semantics.

Since synchronous dataflow languages are an increasingly popular tool for description of embedded control software, before COLA there are various efficient implementations of synchronous languages that already exist (e. g.Esterel [2], Lustre [3]). COLA combines many good features of these existing languages. It is aimed at not only modeling logical architecture but also including hardware models and describing the technical architecture in a consistent formalism.

For the purpose of performance analysis, several previous works report approaches to combining high-level modeling languages and SystemC. Previous work found in [8] proposes the translation from UML class diagrams and object diagrams to SystemC models. UMLSC [10] is a tool that translates UML state diagrams to SystemC. There are other similar approaches. We do not give an exhaustive list here. These approaches do not support modeling down to the implementation level, and therefore, only SystemC skeleton code can be automatically generated. Implementation details must be manually added to make the applications work as expected.

3. OVERVIEW OF COLA

For brevity, we give only an overview of the modeling concepts here. Most of the syntax elements of COLA are introduced in Section 4. For a more detailed description of the COLA language cf. [6].

The key concept of COLA is that of *units*. They can be composed hierarchically, or occur in terms of *blocks* that define the basic (arithmetic) operations of a system. Each unit has a set of typed *ports* describing the interface, which form the *signature of the unit*, and which are categorized into input and output ports. Units can be used to build more complex components by building a *network* of units and by defining an interface to such a network. The individual connections of (sub-) units in a network are called *channels* and connect an output port with one or more suitably typed input ports.

In addition to the hierarchy of networks, COLA provides a decomposition into *automata* (i. e., finite state machines, similar to UML Statecharts). If a unit is decomposed into an automaton, each state of the automaton is associated with a corresponding sub-unit, which determines the behavior in that particular state. This definition of an automaton is therefore well-suited to partition complex networks of units into disjoint *operating modes*, whose respective activation depends on the input signals of the automaton.

An example of modeling an adaptive cruise control is used for illustration of the proposed modeling principles. The top-level network, i.e., the COLA system representing the ACC model, is shown in Figure 1. The main components are the user interface (net_ui), which realizes the control actions, and the display (DEV_A_DISPLAY), the computation of the actual speed (net_rotation), the distance sensing (net_radar), the connection to the engine (DEV_A_MOTOR), and the main control code (net_acc_on_off). In the COLA model, interfaces to hardware are marked by naming the blocks DEV_A_ for actuators and DEV_S_ for sensors.

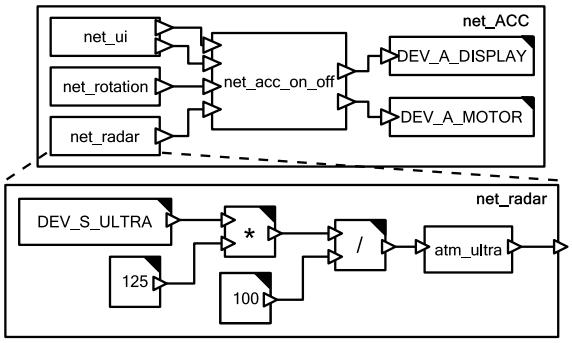

Figure 1: ACC top-level view and decomposition of the component net_radar

As an example, we present the decomposition of net_radar in Figure 1. The network is implemented by constants and basic arithmetic operations on data provided by the ultrasonic sensor (DEV_S_ULTRA). The interface of the network consists of a single output port, whereas all sensor specific data manipulation is encapsulated within this network. The characteristics of the employed hardware require further computation, performed within an automaton (atm_ultra). In a similar manner, the other components of the ACC can be decomposed.

4. COLA TO SYSTEMC TRANSLATION

There are obvious affinities between COLA units and SystemC modules in terms of their structure. Both contain input ports, output ports and an implementation of the intended functionality. A higher level element may be composed hierarchically of several such elements which are connected by channels. The simulation semantics of SystemC is also able to retain COLA semantics. In the rest of this section we describe how COLA semantics is followed by SystemC and present in detail the mapping between COLA and SystemC elements.

4.1 Synchronous Dataflow in SystemC

Being a synchronous formalism, COLA asserts that computation and communication occur instantly in a system, i.e., take no time. The COLA components then operate in parallel at discrete instants of time, only constrained by the causality induced by data dependencies. This behav-

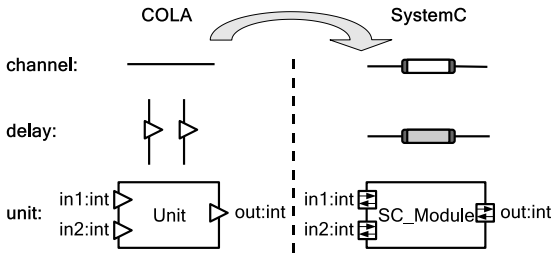

Figure 2: SystemC modules corresponding to the COLA channel, delay and unit

ior of COLA designs can be efficiently modeled in SystemC by means of its delta-cycles (i.e. zero-delay) mechanism. Nevertheless, effort must be spent on mapping communication from COLA models onto SystemC. Since SystemC is a modeling language in a discrete event-driven paradigm, the primitive channels in SystemC have specific events associated with updating of the channels. In COLA, however, communication has no notion of events. Channels propagate data from source ports to destination ports without delay. Therefore, data going through paths of possibly different lengths arrive at the destination ports of a unit at the same time.

As the first but not the best solution, we model COLA channels using *one-stage FIFO channels*. In the rest of the paper we call the one-stage FIFO channels *FIFOs* for short. A FIFO has an event associated with the change from being empty to having data written to it, and another with the change from being full to having data read from it. In order to fulfill the causality requirement of COLA models, the FIFOs are accessed by *blocking* reads and writes. A blocking read will cause the calling process to suspend, until data is available, if a FIFO is empty. Likewise, a process will also suspend, if it accesses a full FIFO with a blocking write. In this way, the SystemC process that realizes the functionality of a COLA unit can perform computation only after all the FIFOs bound to its input ports are full.

4.2 Translation Rules

In this sub-section, we take a closer look at individual COLA elements. We give a description of each element and illustrate the mapping of the COLA elements to the respective SystemC elements.

4.2.1 Channels and Delays

Graphically, a COLA *channel* is simply represented by a line (cf. Figure 2) connecting the ports in question. As discussed previously, we map a COLA channel onto a SystemC FIFO, which is represented by a tube with the two ends in dark gray depicting interfaces of the FIFO.

In the current version of COLA the *delay* is the only element whose behavior is dependent on time. It is used to delay the dataflow for one tick. Intuitively this is a representation of memory, which is initialized with a given default value. At the first evaluation of the delay, the default value is written to the output port and the value present at the input port is used as the new internal state. In all further steps, the internal state is written to the output port and again the input port value is stored as the new internal state. A COLA delay is represented by two vertical lines drawn in parallel. It has exactly one input and one output port, represented by the triangles on the vertical lines. Modeled by a SystemC FIFO that is initialized with a default value before simulation, the semantics of the delay can be preserved. In this paper, we call such a filled FIFO a *DFIFO*. The body of its graphical notation is colored in light gray, as shown in Figure 2.

4.2.2 Units and Blocks

Units are the abstract computing elements within COLA. Each unit defines a relation on a set of input and output values. A unit is denoted by a box labeled with the identifier of the unit (cf. Figure 2). Its ports are represented by triangles on the border. When a COLA unit is mapped onto a SystemC module, its input and output ports correspond exactly to the respective input and output ports of the SystemC module and its functionality is described by a *thread process*. The generated SystemC code that represents the COLA unit depicted in Figure 2 is given in Figure 3.

```
SC_MODULE(module_name){
     sc_fifo_in<int> in1;
     ... // other ports
     ... // channels
     // the thread process that realizes the functionality
     void func_imp(){
          while(1){
               int temp_in1 = in1.read(); // blocking read
               int temp_in2 = in2.read();
               ...
               out.write(...); // blocking write
          }
     }
     SC_CTOR(module_name){
          ... // body of constructor:
          SC_THREAD(func_imp); // declare the process
     }
};
```

Figure 3: A SystemC module from a COLA unit

Blocks are units that cannot be decomposed further. They define basic computational behaviors. Examples of blocks include arithmetic operations, logical operations, or constants. The graphical notation of a block is distinguished from those of the other units by drawing a black triangle in its upper right corner (cf. Figure 4). For each block we do not generate a SystemC module but generate one code line in the process of the module that includes the block. If several blocks are interconnected, the generated code is *inlined*. Figure 4 shows an example of a composite unit that is composed of a set of blocks. The generated code describing the blocks' behavior is:

```
fifo_c5 = fifo_c1*(-1)/20;
```

As inlining is applied here, the FIFOs that interconnect the blocks are not specified.

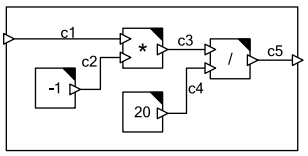

Figure 4: A network composed of blocks

4.2.3 Sources and Sinks

In COLA *sources* and *sinks* are categorized into blocks. We discuss them separately because of their speciality. They are blocks describing input- and output interfaces, through which a COLA system communicates with sensors and actuators, respectively. They are the only COLA elements, the

generated SystemC modules of which should be extended with detailed behaviors. Figure 5 shows the SystemC skeleton code generated for the source DEV_S_ULTRA (cf. Figure 1) which interfaces ultrasonic sensor for distance sensing. Code can be added to describe the data pattern that specifies when and which data is *measured* from the sensor.

```
SC_MODULE(dev_s_ultra){
    sc_fifo_out<int> ultra;

    void dev_s_ultra_imp(){
        while(1){
            int temp_ultra;
            /* please add code here */
            ultra.write(temp_ultra);
        }
    }
    SC_CTOR(dev_s_ultra)
    {
        SC_THREAD(dev_s_ultra_imp);
    }
};
```

Figure 5: Generated SystemC skeleton code for DEV_S_ULTRA

4.2.4 Networks and Automata

A composite COLA unit can be either a *network* or an *automaton*. A network contains sub-units connected by channels. It is used to describe data flow.

Control flow is modeled using automata in COLA designs. The states of automata are also referred to as *operating modes*. Each state of an automaton represents an operating mode and is associated with a behavior. For each state a sub-unit is given to realize the state-associated behavior and computes the output of the automaton. There is only one active sub-unit in an automaton, namely the sub-unit which corresponds to the enabled state of the automaton. The passive sub-units freeze, i.e., retain their current state. The transitions between the states are guarded by *predicates*.

The mapping of a COLA automaton to a SystemC module follows the semantics described above. In the SystemC module, computation and communication are divided into several paths. Each path is associated with a state of the automaton to be modeled. Based on predicates, the flow of data is redirected to the sub-module implementing the enabled state. Figure 6 illustrates an example of a COLA automaton with two states, two input ports and one output port. The activation of the path depends on whether the input p equals *1* or not. All the input data are then forwarded to the active sub-module.

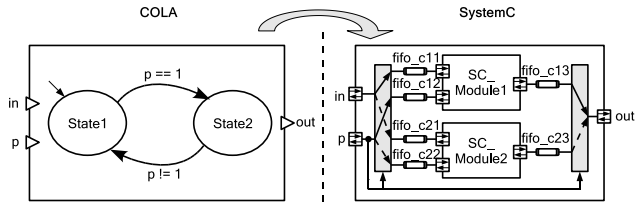

Figure 6: Mapping from a COLA automaton to a SystemC hierarchical module

5. TIMING SIMULATION USING SYSTEMC

Once an automated mapping as described above has been established, simulating realistic scenarios requires a proper definition of external stimuli (environmental/user events). SystemC offers the flexibility to either specify when and which data is *measured* or simulate the interactions between the system and its environment. In this way the functionality of the system can be validated without knowing the target platform.

For the time being, the hardware platform related issues have not been taken into account. The whole system is modeled as concurrently running modules connected by FIFOs which implement the point-to-point communication scheme. Both computation and communication are modeled at the untimed level.

Once the target platform is known, performance analysis of the whole system can be performed, starting at a high level of abstraction and refining it stepwise. The first timing simulation step aims at determining the approximate temporal behavior of the system by annotating timing information to software components and replacing the untimed FIFOs with timed FIFOs to take communication time into account. Such a timing simulation can help making early decisions regarding task allocation and hardware/software partitioning. COLA units are annotated with their respective timing information and stored in a repository, to facilitate reuse.

In the further simulation steps, the target platform is also modeled in SystemC at different levels of abstraction. The software component modules are then bound to the processor models, while the communication can be further refined by replacing the timed FIFOs with SystemC hierarchical channels modeling the desired bus structure. In such a simulation framework, scheduling strategies can be easily integrated. A detailed description of performance modeling is out of the scope of this paper. For this purpose, the approaches proposed in [5, 9] can be followed.

6. CASE STUDY

We now show how the proposed simulation approach can be applied, using a case study of modeling an adaptive cruise control (ACC). This example is an imitation of the concerns and requirements of automotive design, and does not represent a real set of control algorithms for an actual product or prototype.

The intended functionality of the ACC is described as follows. When the ACC is turned on, the speed and distance regulation is activated. This includes the measurement and comparison of the pace set by the user and the actual measured car velocity. If the desired user speed differs from the actual speed, the value for the motor control is corrected accordingly. This regulation is used as long as no object is detected within a safety distance ahead of the car. We chose 35 units for our example. If the distance drops below this threshold, the actual speed is continuously decreased by 5 percent. The minimum distance allowed is set to 15 units. If the actual distance is below 15 units, the car must perform an emergency stop.

According to this specification, the system is modeled using COLA, as shown in Figure 1. The design is then simulated at model level for both functional validation and timing analysis (Figure 7). The test data are defined before simulation, except for the test data of the rotation sensor. As shown in the figure, we add a module rotation_gen to simulate feedback between motor and rotation in order to generate more realistic test data. rotation_gen delays the motor's speed and converts it into appropriate rotation data, which

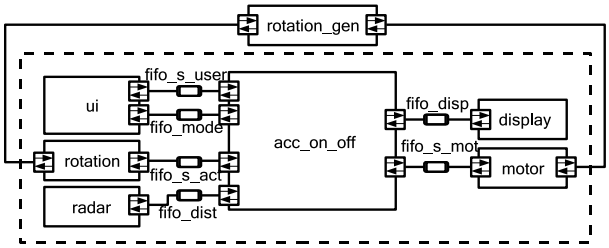

Figure 7: The ACC modeled using SystemC

will be forwarded to the rotation sensor.

In Figure 8 the results of simulating the SystemC model are displayed. The simulation is run for 500 steps in this example. The ACC is enabled permanently during the simulation. The diagram features the intermediate data as well as the output value. The motor speed (s_mot) is the only output value shown in the diagram. Intermediate data include the desired speed (s_user), the actual speed (s_act) and the distance (distance) that are generated by ui, rotation and radar, respectively. s_user is increased or decreased by 1 in each simulation step, controlled by two *triggers* in ui. In the diagram, s_user is increased from 0 to 30 during the first 30 steps. distance is defined arbitrarily in the example, i.e., decreases linearly during the first 300 steps and increases during the last 200 steps. s_act is calculated interactively using the data fed back from the motor. The delay in increase of s_mot visible in the diagram results from the soft start functionality of the ACC. s_mot is reduced some steps when distance is lower than 35 and increases smoothly again after distance exceeds 35 units again. As can be seen, the functional behavior of the designed system can be simulated well using SystemC.

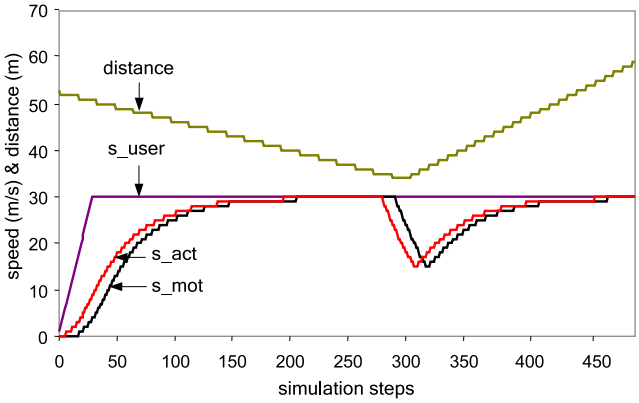

Figure 8: Simulation of the modeled ACC

Further, the timing simulation with a PowerPC processor as target is performed. We assume that the whole application is mapped to this single processor. The individual worst case execution times of the top-level components are given in Table 1. So far, we have not considered more platform related issues, such as communication delays of receiving sensor data, allocation and scheduling, for this simple application. However, the same simulation methodology is also applicable for more complicated applications.

7. CONCLUSIONS

In this paper we presented an approach for integrating SystemC based model-level simulation into a component-based system development process. Such a process benefits from the advantages of both COLA and SystemC. COLA allows for modeling embedded control systems from logical architecture down to technical architecture and from specification down to implementation within a single formalism. The reason for selecting SystemC as simulation framework is the fact that it is widely used for design space exploration and performance analysis of embedded systems and considered as a standard in system level design. Our approach allows for automatic generation of SystemC code from COLA models.

Table 1: Timing Simulation Results

application components	ET (cycles)
net_ui	235
net_rotation	75
net_radar	141
net_acc_on_off(acc off)	96
net_acc_on_off(acc on, distance $<$ 35)	698
net_acc_on_off(acc on, 15 $<$ distance $\leq$ 35)	568
net_acc_on_off(acc on, distance $\leq$ 15)	342

8. REFERENCES

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[2] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[4] Institute of Electrical and Electronics Engineers. IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, 2006.

[5] M. Krause, O. Bringmann, and W. Rosenstiel. A SystemC-based software and communication refinement framework for distributed embedded systems. In *Proceedings of the 13th Workshop on Synthesis And System Integration of Mixed Information Technologies*, Nagoya, Japan, 2006.

[6] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, S. Rittmann, and M. Wechs. COLA – The component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München, Sept. 2007.

[7] H. Posadas, F. Herrera, V. FernÁndez, P. SÁnchez, E. Villar, and F. Blasco. Single source design environment for embedded systems based on SystemC. *Design Automation for Embedded Systems*, pages 293–312, 2005.

[8] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh. Yaml: a tool for hardware design visualization and capture. In *Proceedings of the 13th international symposium on System synthesis*, pages 9–14, 2000.

[9] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf. Task-accurate performance modeling in SystemC for real-time multi-processor architectures. In *Proceedings of Design, Automation and Test in Europe Conference and Exposition (DATE 2006)*, pages 480–481, Munich, Germany, 2006. EDAA.

[10] C. Xi, L. JianHua, Z. ZuCheng, and S. YaoHui. Modeling SystemC design in UML and automatic code generation. In *Proceedings of the 2005 conference on Asia South Pacific design automation (ASP-DAC '05)*, pages 932–935, 2005.