Model-driven SoC Design Via Executable UML to SystemC*

Kathy Dang Nguyen Zhenxin Sun P.S. Thiagarajan
Weng-Fai Wong
School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543, Republic of Singapore
{kathyngu,sunzhenx,thiagu,wongwf}@comp.nus.edu.sg

Abstract

We present a system level description mechanism based on UML-notations from which one can automatically extract SystemC code. Our modelling framework is based on a restricted set of UML diagram types and some standard extensions influenced by the communication infrastructure of SystemC. The system specifications are developed using the UML-compatible tool, Rhapsody [18]. We then translate the internal representation of the design generated by Rhapsody into SystemC code. The extensions we have implemented using the stereotype feature of the Rhapsody tool pull up the communication infrastructure and timing feaures of SystemC to the UML-level. Consequently, we can describe executable platforms at the UML-level as well as translate UML-based application descriptions to SystemC level.

1. Introduction

System level design methods seem inevitable given the technological trends and the accompanying economic pressures. In the recent past, a broad consensus has emerged regarding the basic principles that should govern system level design methods. The main ones are:

- The design methodology should support component-based design accompanied by substantial component-reuse.
- There should be an intermediate representation level with clean *executable semantics*, at which both the application and the platform on which the application is to be realized can be captured and related.
- Behaviors described at the intermediate level, should clearly separate the computational aspects from the communication features.
- This intermediate representation should be able to serve as a common design document for the software and hardware teams which can then independently work towards a detailed implementations.

Given the wish-list above, two crucial choices to be made concern the high level system description language and the intermediate representation language. We claim that a modeling language based on *UML (Unified Modeling Language)* notations for high level system descriptions and *SystemC* as the intermediate representation language constitute sound choices. The main goal of the paper is to substantiate this claim.

UML is now widely accepted in the software engineering community as a common notational framework. It supports object-oriented designs which in turn encourage component-reuse. It can be used to provide multiple views of the system under design with the help of a variety of structural and behavioral diagrams. It allows standard ways of extending the language to meet the demands of specific application domains. Though it was originally created to serve the software engineering community, UML is also becoming an attractive basis for developing system descriptions in the (real time) embedded systems domain [13]. In fact, many of the enhancements to the UML 2.0, the new emerging standard, are geared towards easing the task of specifying complex real time embedded applications.

On the other hand, SystemC allows both applications and platforms to be expressed at sufficiently high levels of abstraction while at the same time enabling the linkage to hardware implementation and verification. Furthermore, SystemC - viewed as a programming language - is a collection of class libraries built on top of C++ and hence is naturally compatible with the object-oriented paradigm that UML is based on. Though SystemC is at present mainly

*This work is supported in part by A*STAR Grants 022/106/0042 and 022/106/0043.

geared towards hardware descriptions, the enhanced version in the making [9] will support software module descriptions and run time features including scheduling. Hence SystemC has the potential to provide a full-fledged description of an execution platform which can serve as the target of a co-design methodology. Thus SystemC is a viable intermediate representation language.

One might wish to consider SystemC itself to serve as the high level system description language. However, at the application level one would like to have visual notations for interacting with the end-users to capture requirements. It is also important to be able to use standard models of computation (MOCs) at the initial design stages. Further, one may not wish to concretely specify the communication mechanisms and instead leave it to be defined by the underlying operational semantics of the MOCs being deployed. Finally, design reuse with the help of minor modifications to an existing component as well as formal verification are easier to carry out at higher level of abstraction than what is offered by SystemC. Hence we propose a top layer of system descriptions using UML notations.

Given these two choices, our initial goal has been to build a flexible and fully automated translation mechanism using which one can transform UML-based system descriptions to SystemC code. A crucial step here is to develop a *coherent subset* of UML-notations since UML offers a bewildering variety of diagrammatic notations and it is up to the user to decide the combined roles of these various diagrams. We select here the so called executable subset of UML, namely class diagrams and state diagrams. The other diagram types may well be useful for capturing user requirements and for documenting important features of the design, but they are unlikely to contribute to code generation and hardware synthesis. One important exception is sequence diagrams. As we discuss later, they do have an important role to play in system level designs but we do not make use of them at present.

The linkage between the UML-layer and SystemC-layer that we have constructed serves a dual purpose. On the one hand, we use it for transforming applications described at the UML-layer to SystemC code for initial simulation. On the other hand, our translation mechanism also enables us to *pull up* the platform description mechanisms to the UML-layer. In this usage, we could consider both the executable platform description and the application models to be available at the UML-layer where we can do formal verification and analyze an abstract version of the mapping problem. Using our translator, a designer can then translate these two descriptions down to the SystemC level for more detailed simulation and move towards a detailed implementation. Thus motivated, a substantial part of our work at the UML-level consists of incorporating SystemC-compatible entities.

In the current stage of our work, we are mainly concerned with the *transaction level modeling* (TLM) layer of SystemC. At this level, the communication infrastructure is specified using function calls and hence the performance numbers reported will generally not be cycle-accurate. This we feel is acceptable if the goal is to rapidly obtain a design document at the SystemC level that describes both the application and the platform. Naturally, many other design steps will have to be constructed in order to realize a viable design flow. We feel that our modeling framework and the translator will provide a sound basis for constructing the missing steps.

We have formulated our UML-based modeling environment using the Rhapsody tool [18]. It supports state diagrams with concurrency and hierarchy (in other words, statecharts [11]). It also provides access to the XMI [17] representation of the design which is faithful and facilitates the translation process.

In order to support UML-based platform descriptions, we have incorporated stereotypes in the Rhapsody environment to capture the communication primitives of SystemC such as interfaces and channels. It is worth noting that the new version of Rhapsody [6] offers communication primitives with a similar flavor as first class entities.

We have also incorporated the clock sensitivity features and other timing aspects of SystemC at the Rhapsody level. Consequently we can describe real time applications accurately while being able to include timing constraints in the UML-based platform descriptions.

As mentioned earlier, we use just class diagrams and state diagrams at present. We feel that the simple sequence diagrams in the Rhapsody version that we have been using are not very useful, except for specifying test runs. However, in UML 2.0, these diagrams have been extended with the powerful features of choice, iteration and conditions. One of our immediate future goals is to use these extended sequence diagrams to specify expressive test benches, translate them into SystemC and use them for verification.

The need for system level design methods has been discussed more eloquently and in greater depth elsewhere[15, 16, 10]. The role of SystemC in this context has also been explored in detail [15, 10]. What UML may have to offer towards system level design methods for real time embedded systems has been studied from a number of perspectives as reported in [13]. For basic material on SystemC, the UML (especially the forthcoming UML 2.0 standard) and the Rhapsody tool, we refer the reader to [21, 10], [22, 4] and [18, 6] respectively. Our programme, initiated here, could also have been based on system description languages such as SpecC , Rosetta or SystemVerilog [7, 1, 8]. Our preference for SystemC over these related languages has been mainly influenced by accessibility and familiarity.

An earlier effort that translates UML to SystemC is YAML[19]. However, YAML uses UML merely to capture the *structural* aspects of the system under design. In contrast, our approach provides for the full-fledged use of state diagrams - including C++ code associated with the actions - and hence can capture *system behaviors*, in particular, concurrency, at the UML-level. YAML also does not take advantage of TLM modeling in SystemC.

In the next section, we recall the main features of SystemC. In section 3 we explain our scheme for using the Rhapsody tool to develop designs. In the subsequent section we first discuss the major details of our implementation. We then present some examples and results to illustrate the main aspects of our translator. The final section concludes with remarks on future work.

2. SystemC preliminaries

Here we briefly describe the basic features of SystemC. For more background information, please see [10] and [21]. SystemC is a library built entirely on C++. It separates computation and communication by having modules and processes for computation; interfaces and channels for communication. Modules are the basic building blocks for partitioning a design. A module hides its data and algorithms from other modules. A module may have one or many processes which can run concurrently. Modules communicate through channels. There are two types of channels: primitive channels and hierarchical channels. Primitive channels are, in some sense, state-less while hierarchical channels can have internal states and control flow associated with them. As the name suggests, hierarchical channels can contain other channels, modules or processes. Interfaces specify the signature of the operations provided by channels. A module accesses a channel through a port whose type is one of the interfaces implemented by the channel.

A key feature of SystemC 2.0 is that communication can be modeled at a higher level of abstraction often referred to as *transaction level modeling (TLM)*. It is hard to pin down this notion precisely. Intuitively, communication between components is described through method calls, without any synchronization. Here, "transaction" stands for the exchange of data between two components of a system. This level emphasizes what data are transferred and from which locations but not the detailed implementation based on a specific protocol. Thus, communication among components is abstracted from the details of the implementation of the communication architecture and this enables component-reuse. In addition, simulation at this level can usually be carried out at high speed. For a more detailed description of TLM, see [5].

Behavioral synthesis of SystemC descriptions is still an unfinished story. On this front, one tool we have been able to access is the CoCentric SystemC Compiler tool. It synthesizes a SystemC behavioral hardware module into an RTL description or a gate-level netlist. Unfortunately, severe restrictions are placed on the SystemC code that can be synthesized.

3. UML modeling

We use two types of UML diagrams for modelling, namely, class and state diagrams. Class diagrams are predominantly used to describe the component structure of a system while state diagrams describe the behavior of the components. Besides the standard UML notations, we have also added some features using the stereotype extension mechanism in order to lift some SystemC entities up to the UML level.

3.1. Class diagrams

We of course use the class hierarchy to describe the computational entities via their methods and their data types. However, class diagrams are also used in a crucial way to give an overview of a system in terms of components and how the components are connected to each other.

We will use the simple bus model available in the SystemC package [21] as a running example. In this model, there are three possible masters, namely a blocking, a non-blocking and a direct master, a bus and two memory slaves. A master initiates transactions on the bus to access the memory. Figure 1 shows a fragment of the class diagram of this example.

We now turn to the use of class notations to define and distinguish between the various features of SystemC lifted up to the UML-level using the stereotype mechanism. This is an extension mechanism of UML that allows one to define virtual subclasses of UML meta-classes with new meta-attributes and additional semantics. Using this, we can define a class as a module, an interface, a primitive channel or a hierarchical channel. In addition, we support declaration of ports for modules to access channels. A port can be declared as an attribute of a module.

Classes can be related by the following UML relations:

- Generalization (or inheritance): when a channel implements an interface, it inherits that interface. Moreover, an interface, channel and module can inherit another interface, channel and module respectively.
- Aggregation/composition: modules and channels may be hierarchical.
- Association: classes that exchange messages with each other are associated to one another. We model messages by UML events with or without arguments. Fur-

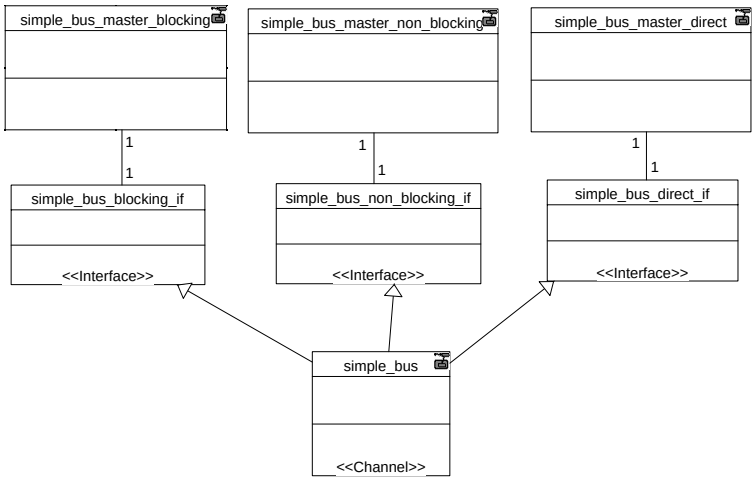

Figure 1. A class diagram

thermore, a module may have an association relationship with an interface when it accesses a channel through this interface.

In the case of the simple bus example, the masters access the bus through three different interfaces. The bus is a hierarchical channel which implements the three interfaces.

3.2. State diagrams

State diagrams describe the behavior of a class. A state can be a simple state or a composite state. A composite state may be concurrent (often called an AND state). A composite state which is not a concurrent state is called an OR state. Being in an AND state means being in all of its sub-states. Being in an OR state means being in exactly one of its sub-states.

Modelling concurrency is an important part of a system specification and this is achieved with the help of AND states. Figure 2 shows a state diagram of a master which is a combination of the three masters described above. This is a derived version of the simple bus model in the SystemC package where we have combined the three masters into one master to illustrate the ability to model concurrency. The AND state `Master` has three sub-states, each of which is an OR state which in turn has a set of simple leaf states (that have no internal structure). Each state is associated with a set of actions on entry and actions on exit. These will be executed when the object enters and leaves that state respectively. A transition connects a source state and a target state. The label of each transition includes a trigger event, a guard and a sequence of actions. Events may be parameterized. A guard is a boolean expression. When an object is in a state and an event of an outgoing transition of that state occurs, the corresponding guard is evaluated. If the guard is true, the transition is taken, the actions are performed and the object moves into another state. Otherwise, the object stays in that state and the event is simply discarded.

The action associated with a transition can be a C++ expression or a function call whose body (in the form C++ code) is to be provided by the user. The action could also correspond to sending an event to another state diagram (describing the behavior of a different class). In addition, the action could be calling an interface method through a port. Moreover, in the actions, we support specification of clock sensitivity or delays in terms of clock cycles or time units through C++ macros. This gives the designers an option to have either un-timed or timed models. Furthermore, this allows users to provide annotations of timing information for performance estimation and architectural exploration. For TLM level implementations, we do not restrict the C++ code associated with the actions in anyway. There will however be severe restrictions when the target is behavioral level SystemC code. We will return to this point in the next section.

Figure 2 shows a concurrent state `Master` consisting of three states: `Master_direct`, `Master_blocking` and `Master_non_blocking`. We describe the behavior of the process associated with the `Master_blocking` state. First, it goes from the initial state to `mb_to_read` state. When the event `finish_bw` arrives, the process performs the action `mb_do_read` and goes to state `mb_waiting_read`. Note that there is no guard for this transition. In state `mb_waiting_read`, the process waits for the event `finish_blocking_read` and when it arrives, does `mb_after_read` then goes to state `mb_to_write`. Other states and transitions can be interpreted similarly.

In UML, objects can also communicate through events

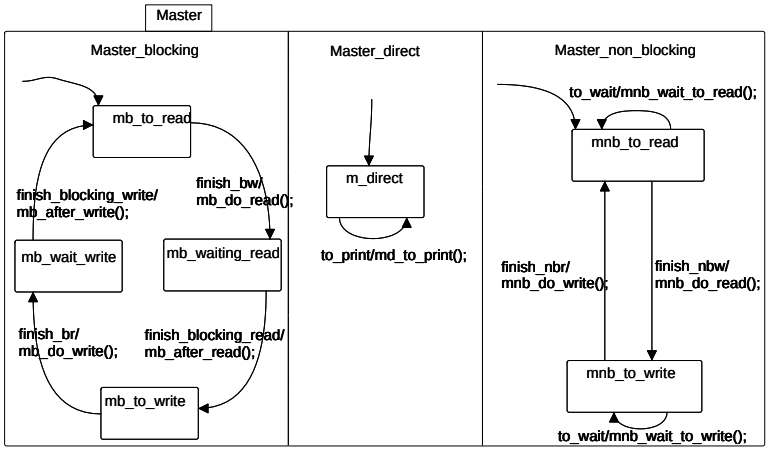

Figure 2. An AND state

that may carry arguments. In our framework, users can model their systems in terms of UML classes (or objects) that communicate through events. They can also have classes corresponding to modules or hierarchical channels to communicate by means of interface methods.

We currently require users to declare a top level class to instantiate objects. Object diagrams could have been used instead to do this, but it is not supported at this time. We also allow for only one level of nesting in our state diagrams with AND states at the top layer and OR states at the second layer. However it will not be difficult to extend our state diagrams to allow more than two levels of hierarchy.

4. Implementation

We use Rhapsody which supports the main features of UML that we need. Moreover, Rhapsody has a toolkit which can generate XMI [17] as an intermediate representation. This representation contains all the information about the model that we need for code generation.

The XMI toolkit of Rhapsody is used to generate XMI document from the graphical models. We then use our XMI parser to gather information from the XMI document to build an abstract tree as an input to a template engine. SystemC code is generated from templates using Velocity [23] which is a template engine that generates code from predefined templates. With the help of Velocity, we decouple the parsing of XMI document from the code generation step so that changes in the XMI parser do not affect code generation process. Further, in the later part of the workflow, we only need to work with the templates to generate code without having to deal with the verbose code of parser. With this approach, by merely changing the templates we can generate code for different levels of abstraction. Figure 3 shows the workflow of our translator.

We support the initialization of multiple instances of a type (module, primitive channel or channel). However, they cannot be created dynamically because at present SystemC does not support dynamic instantiation; the structure of a system is assumed to be determined at elaboration time.

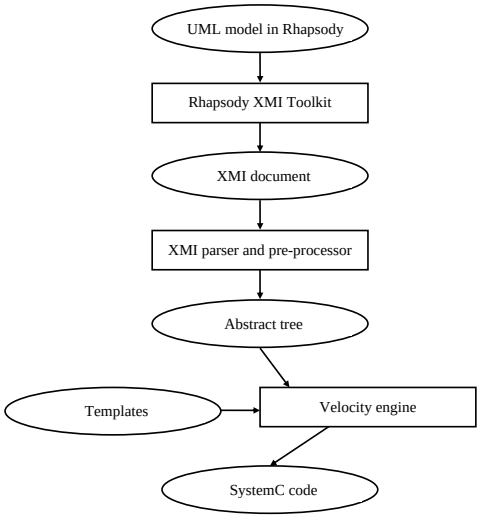

Figure 3. Our implementation workflow

4.1. Translation to TLM level

SystemC code at the TLM level is ideal for simulation as details of the low level communication infrastructure are not present. In our design flow, users do not have to specify any SystemC components at UML level. They can work with classes or objects, state diagrams and model communication between objects by events with or without arguments. Such events will be implemented through SystemC primi-

tive channels. Each module has a primitive channel to receive events sent by other modules. This primitive channel essentially acts as a buffer for incoming events to that module. When a module has an association relationship with another module, it can send messages to that module. A port is declared in its code body to access the other module's primitive channel. Thus, the primitive channels implement two interfaces: the interface for sending events and the interface for receiving events. The SystemC code generated by our translator will be at the TLM level since the senders and receivers just call functions of the primitive channels, regardless of whether or not the events have arguments associated with them.

Users can also specify SystemC components, such as interfaces and channels through stereotypes, ports, time delay, and clock sensitivity through C++ macros. These are translated to SystemC accordingly.

There are three types of SystemC processes: `sc_thread`, `sc_cthread` and `sc_method` [10]. Each OR state in a state diagram is translated into a `sc_thread` of the corresponding module. Thus, a state diagram such as the one in Figure 2 will be translated into three `sc_thread`s of the same module. The reason we chose `sc_thread` over `sc_method` and `sc_cthread` is that an `sc_thread` can be suspended during execution to wait for events and is not necessarily sensitive to every clock edge.

4.2. Translation to behavioral level

Behavioral level SystemC code has to have the details needed for the generation and synthesis of the hardware implementation of the described system. As such, it is necessary to place certain restrictions on the design. The restrictions we place at Rhapsody level on the C++ code supplied by the user are the ones demanded by the Synopsys tool. Further, at the Rhapsody level, users have to declare a class called `Top` to initialize all the instances. This is because the method `new()` used to create instances is not synthesizable. Users may however initialize multiple instances of the same class. The translator will create the corresponding modules and connect them according to their specified relationships.

The code to be synthesized at this level has to comply with the coding convention of the Synopsys tool. Restrictions are placed on the data types, constructs, instructions and SystemC classes [19]. Due to these restrictions, an OR state is translated to an `sc_cthread` which is only sensitive to an active clock edge. Communication at this level is done only through signals. According to the Synopsys coding convention, an interface method call from a module to a channel cannot be synthesized. We find this restriction rather severe and have implemented a fix to work around this restriction. The translator detects function calls and replaces them by signals that are then sent to trigger local function calls. The transfer of function arguments and return values are also translated into an exchange of signals. Although at UML level, there may be interfaces and channels, the translator converts channels to modules and ignores interfaces since only modules can be synthesized. The translator adds an `sc_cthread` to each channel module to receive triggers for local function calls. The UML events and their arguments are also translated into signals. Thus, the actual implementation is complicated by the need to detect module-module and module-channel connections and connect the corresponding signals at elaboration time.

5. Examples and results

5.1. A Simple Bus

This is a benchmark example of SystemC at the TLM level which has been described partially in the previous section. Here we use it to demonstrate how one might model a fragment of a platform at UML level and translate it into SystemC. This model uses all the four stereotypes mentioned above, namely modules, interfaces, primitive channels and hierarchical channels. In particular, the three masters are modules that access the bus through three ports using three different interfaces. The bus is a hierarchical channel which implements the methods of the three interfaces.

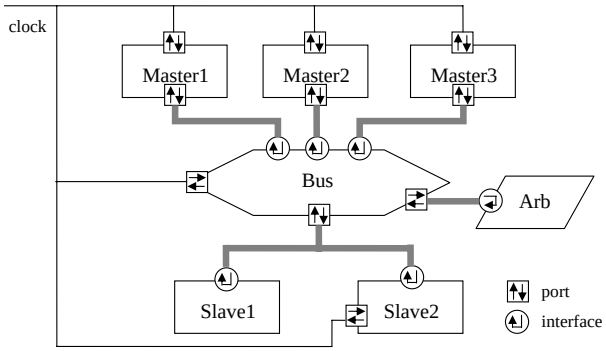

Figure 4. Block diagram for the simple bus example [10]

For faster simulation speed, the arbiter and the fast memory are modelled as primitive channels to decrease the number of threads and thus, decrease the context-switching time. The bus accesses these primitive channels through the arbiter interface and the fast memory interface, respectively. The `Top` class initializes all the objects of the system, in this case one instance for each module and channel. Figure 5 shows the class diagram of this example.

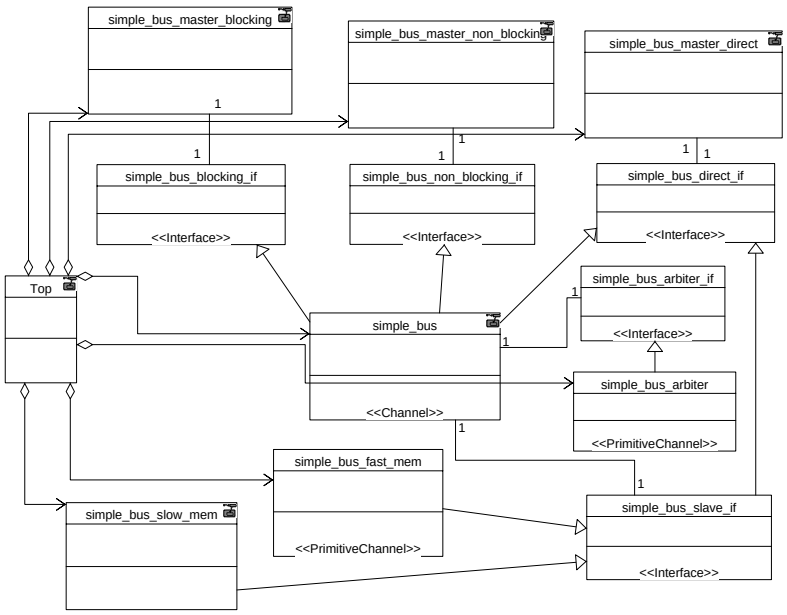

Figure 5. Class diagram of the simple bus example

We first captured this UML-level model using the Rhapsody 4.2 tool and then translated it into TLM SystemC code automatically using our translator. We then simulated the resulting SystemC code using the standard SystemC simulation kernel. When measuring performance, we did not initialize the direct master, because it is only used for debugging. The experiments were performed on Linux Red Hat 9.0 running on CPU Intel Xeon 2.8GHz. We measured the number of CPU clock cycles for 1,000 transactions using the Pentium's `rdtsc` instruction. With the original code provided in the SystemC public distribution, we obtained a speed of 81K transactions per second. In comparison, with our automatically generated code from the UML model, we obtained a speed of 41K transactions per second. One reason for the slower simulation speed of our generated code is the use of `sc_thread` for all processes. The original model has the bus and the slow memory implemented as `sc_method`s. Due to the need for context switching, `sc_thread`s run slower than `sc_method`s. We intend to look into this and other optimization issues in our future work.

5.2. The micro polymerase chain reaction controller

This is a simple real-time embedded control system. Polymerase Chain Reaction (PCR) is a thermal cycle reaction used for the rapid *in vitro* multiplication of DNA samples [14]. The μ-PCR chip realizes a miniaturized version of this process where a small quantity of the DNA sample is placed in each chamber of the chip and the PCR reaction is achieved by controlling the thermal power supplied to the the chambers according to an input temperature profile. A schematic diagram is shown in Figure 6. We will not describe here the PCR bio-chemical process in detail but instead focus on the functional model of the real time controller. This unit is driven by the temperature profile (which specifies the control objective) and feedback received from the chip regarding the current temperatures of the chambers. In the present version of the plant model, the effects of inter-chamber influences are ignored as a simplification. Hence there is essentially one independent controller for each chamber. This controller periodically reads the temperature (converted into a voltage value via an analog-to-digital converter) of the chamber. With the help of the estimator (the control law) it then computes the output voltage required for that chamber to maintain the temperature according to the temperature profile of the current PCR thermal cycle. This voltage is then converted back into an analog value (via a digital-to-analog converter), which is then used to control the heating element of that chamber.

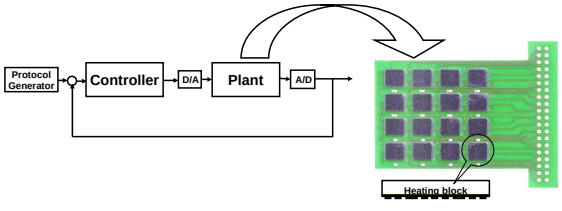

Figure 6. μ-PCR block diagram

In this example, the profiler which keeps the temperature profile and the estimator which keeps the control laws were modelled separately from the controller so that we can re-configure the temperature profile and control law easily. They were modelled as primitive channels in order to get better simulation speed at the TLM level. The communication among the modules are cycle-accurate, in the sense the status of a module's input and output are specified at each clock cycle. Yet another real time aspect of this example is the timing diagram associated with the A/D converter. For this example, we have synthesized the behavioral level SystemC code generated via our translator using the CoCentric compiler tool of Synopsys.

This application has been simulated at both TLM and behavior level. Following is a fragment of the code for the thread of the Controller at behavior level. For brevity we have eliminated some `wait()` statements.

```
while (true){
  switch (state){
    case 72:
      // wait for the current temperature signal
      // from the ADC
      wait_currenttemp();
      if ( read_currenttemp() == true ){
        //the guard of this transition is true
        if (true){
          //get current temperature value
          Yk = GET_PARAM(currenttemp, temp);
          state = 76;
          break;
        }
      }
      break;
    case 76:
      if (true){
        // send and receive information
        // from the profiler
        setPoint=IMC(profiler_port,mapping(timer));
        state = 80;
      }
      break;
    . . .
  }
}
```

Table 1 shows the simulation speed -in terms of transactions per second- of the Micro-PCR example on the same platform as the one used in he previous example. By a transaction we mean the period of time in which the controller senses the current voltage, computes and outputs to the plant.

Our simulation results show, as expected, that simulation speed at the TLM level is higher than that at the behavior level. The experiments also give evidence that the code we generate scales fairly well in terms of performance.

5.3. Digital down converter

For our third example, we implemented a *digital down converter* (DDC) for the *global system for mobile communications* (GSM) - a wireless communication protocol . Digital radio receivers often have fast analog to digital converters delivering vast amounts of data. However, in many cases, the signal of interest represents a small proportion of that bandwidth. A DDC is a filter that extracts the signal of interest from the incoming data stream. Our implementation closely follows the MATLAB example in Xilinx's system generator (see Figure 7).

The desired channel is translated to baseband using the digital mixer comprised of multipliers and a direct digital synthesizer (DDS). The sample rate of the signal is then adjusted by a multi-stage, multi-rate filter consisting of a cascade integrator-comb (CIC) filter and two polyphase finite impulse response (FIR) filters with a decimation factor of 2. The functions performed in the system are complex multiplication and multi-rate filtering. The overall down sampling rate of the converter is 192:1.

Each of the components is mapped into a module, and the data is sent through the chain by events (see Figure 8). The model has been translated into both TLM and behavioral levels. We could not find the source code for a similar DDC in UML or SystemC for comparison. Hence we have compared just the FIR module of our design with an FIR example provided by Synopsys. The only modification we did to the Synopsys code was to ensure that the coefficients and the bit-widths of the ports are the same as those of our FIR model. The codes were compiled into gate-level netlist using Synopsys `tc6a_cbacore` library, which targets cell-based array architectures [20]. The same timing constraints were used on the synthesis runs of both. Table 2 shows the comparisons of the final synthesized hardware. From the result we can see that our generated code uses about 33.25% more resources than the hand-coded version. We believe that this is an acceptable overhead given the fact we input the model using the Rhapsody tool with UML notations.

6. Conclusions and future work

We have presented here the backbone of a framework in which designs can be specified using UML notations. SystemC code implementing these designs can then be automatically generated. We showed three examples illustrating the use of abstraction via object-oriented modeling, real time constraints, transaction level modeling and behavioral synthesis. We see this framework as a sound basis for carrying out further research on system level design methods.

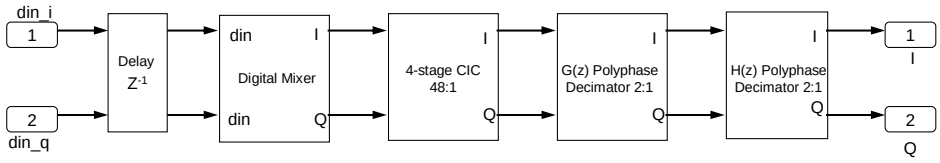

Figure 7. Digital down converter for GSM - block diagram

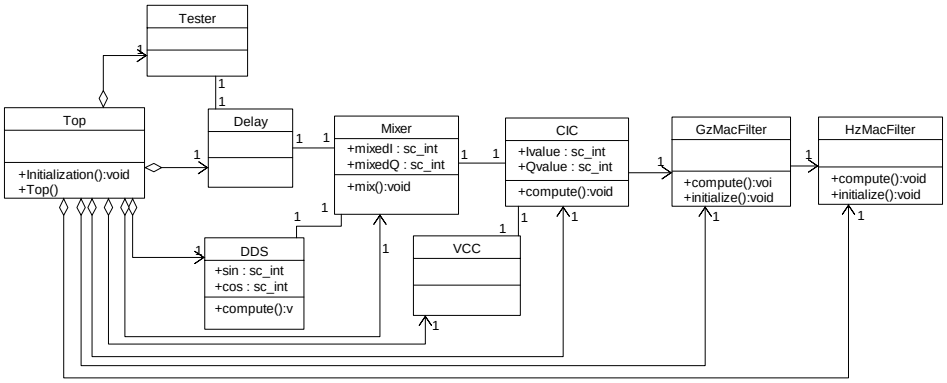

Figure 8. Digital down converter for GSM - class diagram

Table 1. Simulation speed of the Micro-PCR example

Chamber arrangement	TLM sim. (trans./sec)	Behavioral sim. (trans./sec)
2×2	12,125	5,766
4×4	2,714	1,543
5×5	1,676	785
8×8	555	148

Table 2. Area statistics for FIR component implemented on cell-based array architecture

	FIR from Synopsys(S)	FIR from DDC(D)	Ratio((D-S)/S)
Number of ports	260	261	0.39%
Number of nets	18393	27942	51.92%
Number of cells	18010	27547	55.15%
Number of references	93	99	6.45%
Combinational area	30181.2	50583.7	67.60%
Non-combinational area	34560.0	36844.2	6.61%
Net interconnect area	244806.2	325033.1	32.77%
Total cell area	64741.1	87430.3	35.05%
Total area	309547.6	412461.1	33.25%

An important extension will be to incorporate test benches. Here we see the sequence diagram extensions provided in UML 2.0 as very promising. In the new standard, complex interaction patterns can be specified using choice and iteration operations on sequence diagrams. One can also attach conditions to the sequence diagrams to capture "may" and "must" assertions in the spirit of Live Sequence Charts [12]. Consequently, one can capture a variety of requirements provided by the end-users and the design teams. These extended sequence diagrams can be endowed with precise execution semantics. Hence it will be possible to specify requirements at the UML-layer and translate them into the SystemC level to generate powerful test benches.

Another important direction to pursue is the problem of mapping applications to executable platform descriptions. This is a difficult problem and a great deal of effort is required to capture generic platforms with a sufficient amount of detail. Our hope is that efforts such as [2] can be exploited to rapidly derive SystemC-based descriptions of realistic platforms. One could then design an infrastructure to solve the mapping problem with the crucial supporting feature being that both the application and the platform are executable SystemC programs at a comparable level of granularity. In this context, we also hope to leverage on the insights being gained in Metropolis project [3] which however uses a native intermediate representation language called the Metropolis Meta Model from which there is a path to SystemC.

A third important line of work would be to instrument our translator so that diagnostic information obtained via simulations at the SystemC level can be reflected back to the UML-based model level. Here, we expect test benches based on (extended) sequence diagrams at the UML-level and their translated SystemC versions to play an important role.

References

[1] P. Alexander, R. Kamath, and D. Barton. System specification in Rosetta. In *IEEE Engineering of Computer Based Systems Symposium*, April 2000.

[2] D. August, K. Keutzer, S. Malik, and A. R. Newton. A disciplined approach to the development of platform architectures. *Microelectronics Journal*, 33:881–890, 2002.

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Paserone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. 1999.

[5] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign & system synthesis*, pages 19–24, 2003.

[6] B. Douglas. Breakthroughs in software and systems engineering. White paper, I-Logix, 2004.

[7] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[8] R. Goering. Next-generation Verilog rises to higher abstraction levels,. EE Times, March 2002.

[9] T. Groetker. Modeling software with SystemC 3.0. European SystemC Users Group Meeting, 2002.

[10] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, May 2002.

[11] D. Harel. Statecharts: A visual formalism for complex systems. In *Science of Computer Programming*, volume 8, pages 231 – 274, June 1987.

[12] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[13] L. Lavagno, G. Martin, and B. Selic. *UML for Real: Design Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003.

[14] Y. Lin, C. Yang, M. Hwang, and Y. Chang. Simulation and experimental verification of micro polymerase chain reaction chip. In *Technical Proceedings of the 2000 International Conference on Modeling and Simulation of Microsystems MSM 2000*.

[15] G. Martin. SystemC and the future of design languages: Opportunities for users and research. In *The 16th Symposium on Integrated Circuits and Systems Design. IEEE Press*, 2003.

[16] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: A merger of real-time UML and co-design. In *The 9th International Symposium on Hardware/Software Co-Design*, pages 23–28, 2001.

[17] OMG. XML Metadata Interchange (XMI), version 1.0. ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf, 2004.

[18] Rhapsody home page. http://www.ilogix.com/products/rhapsody/index.cfm, 2004.

[19] Synopsys. Synopsys CoCentric SystemC compiler behavioral user and modeling guide, 2001.

[20] Synopsys. Designware building block IP user guide, 2004.

[21] SystemC home page. http://www.systemc.org, 2004.

[22] UML Resource Page. http://www.uml.org, 2004.

[23] Velocity website. http://jakarta.apache.org/velocity/.