UML AS A SYSTEM LEVEL DESIGN METHODOLOGY WITH APPLICATION TO SOFTWARE RADIO

SUN ZHENXIN

NATIONAL UNIVERSITY OF SINGAPORE

2004

UML AS A SYSTEM LEVEL DESIGN METHODOLOGY WITH APPLICATION TO SOFTWARE RADIO

SUN ZHENXIN
(B. Comp.(Hons), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

First and foremost, I would like to thank my supervisors, Dr. Wong Weng Fai, Dr. P.S. Thiagarajan and Mr. Santhosh K Pilakkat, for giving me the time, insights, and invaluable guidance throughout this research project. I am also grateful to their encouragement during my work.

I would also like to take this opportunity to thank my partner, Ms. Kathy Nguyen Dang. She is wise and intelligent, and always can get new idea to solve the problems we met in the project. Without her, I would not be able to finish the project. I would also like to take this opportunity to thank Dr Winson Zhu Yongxin. He was always around to assist me in solving technical problems and gave me good suggestions.

A great deal of thanks is due to all the people in Embedded System Lab for being so friendly and help during my master study. Special thanks to my girlfriend Xiao Wei for her understanding and encouragement during this period of time. She has always reminded me to stay optimistic and never give up.

Table of Contents

List of Figures

List of Tables

Summary

With raising the complexities of embedded systems, designers have been searching for new methodology that can handle given complexities with relatively low cost. Unified Modeling Language (UML) is a powerful specification language used widely in software development, as it improves product quality and productivity significantly. Embedded system designers also use UML notations to capture the system level design requirements and do the early stage testing. However, UML-based models are still far away from the implementation code. In order to address this problem, we have developed a framework for deriving SystemC implemented design from the class, statechart and component diagrams in order to capture the structure and the behavior of embedded systems. With a code generator, the system level implementation can be directly derived. Designer is able to generator different implementations from top-level to lower level. On each level, the executable models will help designer to test and verify their design. These form a top-down design flow, and at the end of the flow, synthesizable SystemC models will be generated, which is one step from hardware chips.

With this methodology, a bridge between the design and the system level implementation can be built. It can help designers with coding writing, model testing and co-designing. UML provides very powerful graphical notations to capture the system specifications. At the early stages, the system model can be easily and clearly defined. At the same time, designers will be able to focus on the structure and behaviors of system and avoid the process of manual translation of code.

In the design progress, testing and verification can be done in multi-level of abstraction. As we build executable model from UML executives, the generated code can be used to

test and verify the models. In addition, testing code can be inserted in the models and they will be automatically in turn inserted into the generated model code.

This methodology can also help in the software/hardware co-design. SystemC can be used to simulate both software components and hardware components. Therefore, at early stages, designer build the integrated system model without perform the partitioning. After synthesizable model being built, design can then decide the partitioning. Hardware portions will be further translated into gate-level model by using CoCentric tools, which is just one step from the hardware chip. The derived SystemC specifications can be further compiled into system hardware implementation by using CoCentric compiler. One of our case studies, Software Radio design shows many significant advantages of this methodology.

We defined a set of rules and use class and statechart diagrams to capture the structure and behavior of embedded systems. Components diagrams are used to model clock setting for each components. Facts shows that the Software and hardware components as well as their communication channels can be modeled using native UML-notations. A code generator, called UML2Code, has been built to generate SystemC implementation from executable UML model. Three levels of SystemC specifications can be generated for testing and simulation purpose. With a promoted design flow, designer is able to get the synthesizable RTL code at the end. With some restriction in the coding style, the generated code can be further compiled to gate level models by Synopsis CoCentric Compiler. We did rigorous tests on our code generator. The results shows that the generator is reliable and generated code are compact and relatively effective. We explored the application of our methodology on the development of Software Radio. One of the applications, Digital Down Converter, has been built to show the advantages of the methods in the Software Radio design.

I: Introduction

1.1 Motivations

Recently, the importance and significance of embedded system keeps increasing and the complexity of the embedded system is growing as the range of applications becomes larger. [1] Thus more and more complex systems require the design be built in less time and with relatively low cost. Due to this situation, there is increasing demand on more reliable and cost-efficient design and develop methodology. On the other hand, the development methodologies in various design organizations are quite ad hoc. System specification may be mis-captured during the design stage due to the lack of formal method, and model exchange and the spread will be difficult. Furthermore, manual translation from model to code is tedious and error-prone.

On the other hand, traditional co-design approach required the software/hardware partitioning being done at the early stage. [5] After testing different partitioning, the optimized resolutions can be found. But each refinement requires reprogramming the components as well as the interfaces. This is very costly and time consuming.

Several studies have been proposed to address the problems by considering using MDA approach for software/hardware con-design.[2][3] Most of them were trying to model the functional and structural requirements at high level, and they have obvious limitations because of lack of the behavioral information.[4]

Our project is trying to apply MDA (Model Driven Architecture) approach for entire system development. More precisely, full implementation of software and hardware components as well as their communication interfaces can be produced directly from the system specifications. We use UML language as the PIM (platform independent model). The Unified Modeling Language (UML) has been widely used in software developments. UML is a language for specifying, visualizing, constructing, and documenting the artifacts of systems. [6] It provides convenient notations that allow the developer to capture structural and behavioral details using an object oriented design methodology. It has been proved that UML can improve product quality and productivity significantly. These are also major concerns in embedded system design. However, UML-based descriptions are still far from implementation level specification. While these specifications need to be refined to a more detailed level. SystemC, a C++ based modeling language, is a ideal choice which allows designers to work at a sufficiently high level of abstraction when expressing and verifying designs, yet enabling the linkage to hardware implementation and verification.[7]

The aim of this project is to build a UML Profile for System Level Design that will allow

- Leveraging of the informal expressiveness of the UML to capture the requirements at the early stages (E.g. Usecases, Activity, collaboration diagrams etc.)
- Early modeling of requirements using formal executable UML Models, without committing to system partitioning to Hardware/Software(Using class, object and state diagrams and SystemC), ie., produce an executable behavioral simulation model.
- Follow the core OO Method of step-wise iterative refinement to take the behavioral UML simulation models into implementation at the executable UML Models by providing code generation into appropriate languages for each

level/phase of refinement. (E.g. SystemC behavioral for algorithmic level, SystemC RTL, SystemVerilog, VHDL, Verilog RTL and C/C++ for Implementation model).

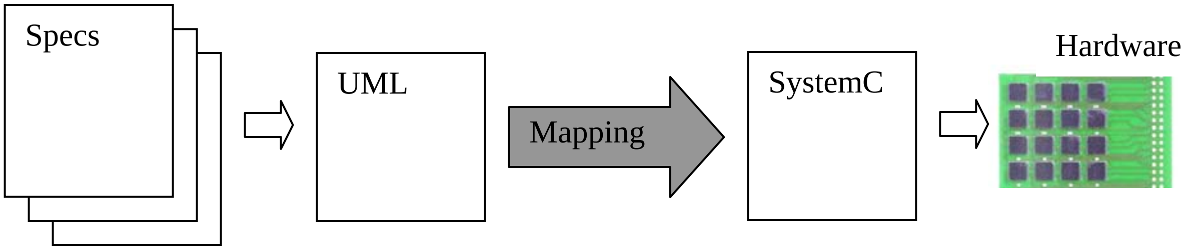


Figure 1 Mapping between Specifications and hardware

With this methodology, a bridge between the design and the system level implementation can be built. It can help designers with coding writing, model testing and co-designing. UML provides very powerful graphical notations to capture the system specifications. At the early stages, the system model can be easily and clearly defined. At the same time, designers will be able to focus on the structure and behaviors of system and avoid the process of manual translation of code.

In the design progress, testing and verification can be done in multi-level of abstraction. As we build executable model from UML executives, the generated code can be used to test and verify the models. In addition, testing code can be inserted in the models and they will be automatically in turn inserted into the generated model code.

This methodology can also help in the software/hardware co-design. SystemC can be used to simulate both software components and hardware components. Therefore, at early stages, designer build the integrated system model without perform the partitioning. After synthesizable model being built, design can then decide the partitioning. Hardware portions will be further translated into gate-level model by using CoCentric tools, which is just one step from the hardware chip.

1.2 Achievements

During Master study, we explored a methodology which uses UML-notations to do the system level design. The major achievements are as follows:

- *System Specifications capturing*

  We defined a set of rules and use class and statechart diagrams to capture the structure and behavior of embedded systems. Components diagrams are used to model clock setting for each components. Facts shows that the Software and hardware components as well as their communication channels can be modeled using native UML-notations.

- *SystemC code auto-generation*

  A code generator, called UML2Code, has been built to generate SystemC implemtnation from executable UML model. Three levels of SystemC specifications can be generated for testing and simulation purpose. With a promoted design flow, designer is able to get the synthesizable RTL code at the end. With some restriction in the coding style, the generated code can be further compiled to gate level models by Synopsis CoCentric Compiler.

- *Code Generator testing and evaluations*

  We did rigorous tests on our code generator. Testing include the correctness, efficiency, code length, code quality and other important aspects. The results shows that the generator is reliable and generated code are compact and relatively effective.

- Case studies on Application: Software Radio

  Software radio is new technology in embedded system. It is highly software/hardware co-related. With traditional design methodology, the time-to-marcket is quite long and the cost is high. We explored the application of our methodology on the development of Software Radio. One of the applications, Digital Down Converter(DDC), has been built to show the advantages of the methods in the Software Radio design.

In past few months, some experiments have been done to study the translation from UML-notation, in more details, Class diagrams and State diagrams, to SystemC models. An auto-generator code generator, called UML2Code, has been built to perform the translation. From executable UML-model, the generator can produce executable SystemC model, which can be used for test and simulation purpose. With some restriction in the coding style, the generated code can be further compiled to gate level models by Synopsis CoCentric Compiler. The detailed implementation of the generator will be discussed in section 3. We have done some interesting examples using our promoted design methodology. Software Radio, as a new technology, is highly software/hardware co-related and costly in the design and testing progress. We found that the designer can benefit a lot by using our design methodology on development of Software Radio. We have developed a small example, a Digital Down Converter for GSM.

Through the experiments, a direct mapping between system description and target system level model was established. It raised several very interesting points and encourages us to do more future work on it.

1.3 Thesis Outline

The rest of paper is organized as follows. In section 2, we describe the background information such as UML and SystemC. In section 3, several related works will be discussed. In section 4, we show how we implement the translation and what we have achieved in the experiments. In section 4, we show PingPong example and its testing result, and show a software radio example: A Digital Down Converter. In section 5, we summarize the results and draw some reasonable conclusions, and then some interesting future work will be discussed.

II: Background

2.1 MDA (Model Driven Architecture)

The MDA is a new way of writing specifications and developing applications, based on a platform-independent model (PIM). A complete MDA specification consists of a definitive platform-independent base UML® model, plus one or more platform-specific models (PSM) and interface definition sets, each describing how the base model is implemented on a different middleware platform. A complete MDA application consists of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that the application developer decides to support.

The MDA defines an approach whereby you can separate the system functionality specification from its implementation on any specific technology platform. That way you can have an architecture that will be language, vendor and middleware neutral. For creating MDA-based applications, the first step will be to create a Platform Independent Model (PIM), which you should express in UML. Such a PIM can then be mapped to a Platform Specific Model (PSM) to target platforms like the CORBA Component Model (CCM), Enterprise JavaBeans (EJB) or Microsoft Transaction Server (MTS). Standard mappings should allow tools to automate some of the conversion. Such a PSM, again expressed in UML, can then be actually implemented on that particular platform.

The architecture encompasses the full range of pervasive services already specified by OMG, including Directory Services, Event Handling, Persistence, Transactions, and Security. Most importantly, MDA enables the creation of standardized Domain Models for specific vertical industries. The benefit of such an architecture is obvious! You will be able to build new MDA-based applications using platforms/middleware of your choice. Also, such an approach makes it easier to integrate applications and facilities across middleware boundaries. Since you have defined the architecture once, not only can you integrate legacy and present applications but you can also integrate those of the future!

As usual OMG will be releasing only the specifications, and vendors are expected to develop tools based on these specifications. Some of the specifications have already been released while many others are yet to be developed. For more details please visit http://www.omg.org/mda

2.2 UML

Unified Modeling Language (UML) relies on OO paradigm and well-known software engineering principles as information hiding by data abstraction and reuse of software components by inheritance and generality are supported. UML is a visual modeling language for specifying, visualizing, constructing and documenting software systems and business processes. It was created in 1994 as fusion of the object-oriented method Booch, OMT and OOSE.[6] The UML specification defines a formal Object-oriented Analysis& Design metamodel, which includes diagram types for static, behavioral, usage, and architectural aspects of software systems. Differing to its predecessors, the UML specification only specifies syntax and semantics of its notation, but does not determine how to apply its elements within a development process. [17] Therefore, several different modeling methods use the UML notation and additionally consider domain specific details.

While UML is well-suited for modeling software systems in general, it lacks support for some aspects important to embedded real-time systems, e.g. modeling of timing constraints, signals, and independent components.[8] Therefore, different proposals to extend the UML for modeling real-time systems have been made. The Object Management Group (OMG) proposes to extend the UML by building UML “profiles” that contain the needed extensions. [16] However, this extension mechanism is currently not part of the standard and it is still discussed how to realize it. In parallel the leading CASE tool vendors implement proprietary extensions to the UML. Rational and Telelogic adapt UML for modeling embedded real-time systems by combining it with the modeling languages from the real-time (ROOM) and telecommunication domains (SDL), while I-Logix stays with Standard-UML, but provides a very powerful implementation of statecharts. [9]

2.3 SystemC

SystemC[7], a system level modeling language based on C++. It provide library supporting system level design. It has been attracting more and more attention. SystemC has desirable properties for system level design. Besides, SystemC use most of C++ grammar, and this allows user to learn it in a very short time. Even those who have no experience with programming in SystemC can read and understand the code.

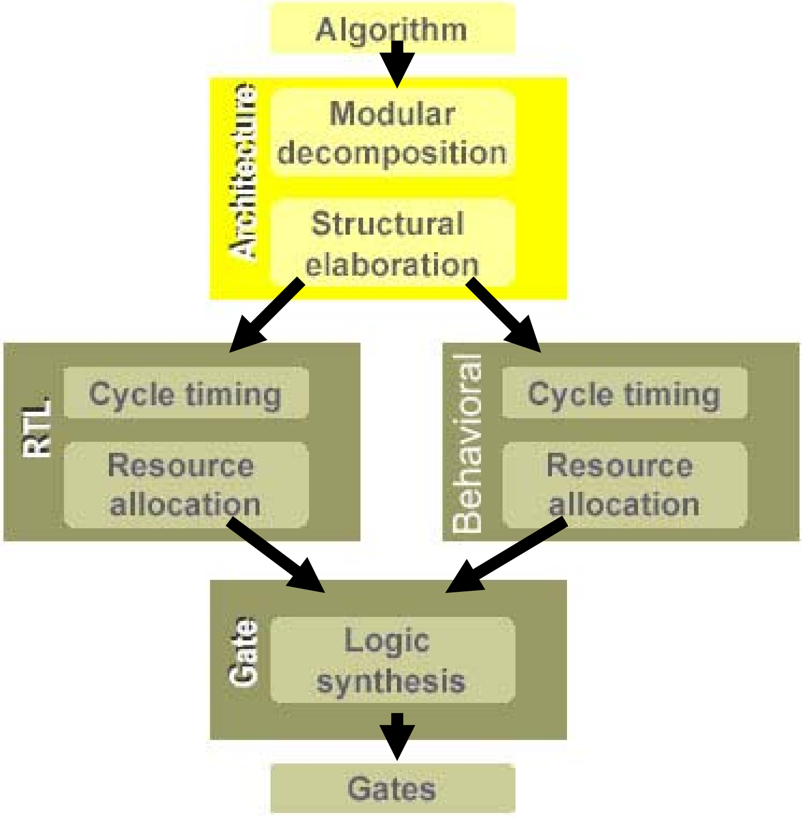


Figure 2 Different Levels of SystemC model

Using the SystemC library, user can model as system at various levels of abstraction. At the highest level, only the functionality of system may be modeled. For hardware implementation, model can be written either in a functional (behavioral) style or RTL (register-transfer level) style. [9] The software part of a system can be naturally described in C++. Interfaces between software and hardware and between hardware blocks can be described either at the transaction-accurate level or at the cycle-accurate level. More over, different parts of the system can be modeled at different levels of abstraction and these models can co-exist during system simulation. C++ and SystemC classes can be used not only for the development of the system, but also for the test-bench. SystemC consists of a set of header files describing the classes and a link library that contains the simulation kernel. Any ANSI C++ compliant compiler can compile SystemC, together with the program. During linking, the simulation kernel of the SystemC library is used. The resulting executable serves as a simulator for the system designed.

SystemC provides an ideal platform for developing embedded systems. Software and hardware parts can both be specified using the same language and verified using a common test-bench. The hardware parts may be refined up to RT level and implemented by using synthesis tools. The hierarchical modeling features of SystemC are supported by the hierarchical specification model. This facilitates not just a structured design, it also enable IP reuse. The FSMs can also be organized in a hierarchical manner, implementing a Hierarchical control flow.

III: Related works

This section briefly discusses other projects that have investigated integrating formal and informal approaches to systems development, where multiple modules are used to describe a system.

3.1 YAML

Most of the effort we have seen in the UML-SystemC translation was to generated skeleton SystemC code from, in particular, class diagrams and object diagrams. An example of this kind of projects is YAML.[10] YAML uses UML notations to model hardware and allows user to input information about objects and relationships into a UML class diagram (for behavioral hierarchy) and object diagram(for structural modeling). YAML generates the C++ code for the design, using information input by the user to the UML class and object diagrams.

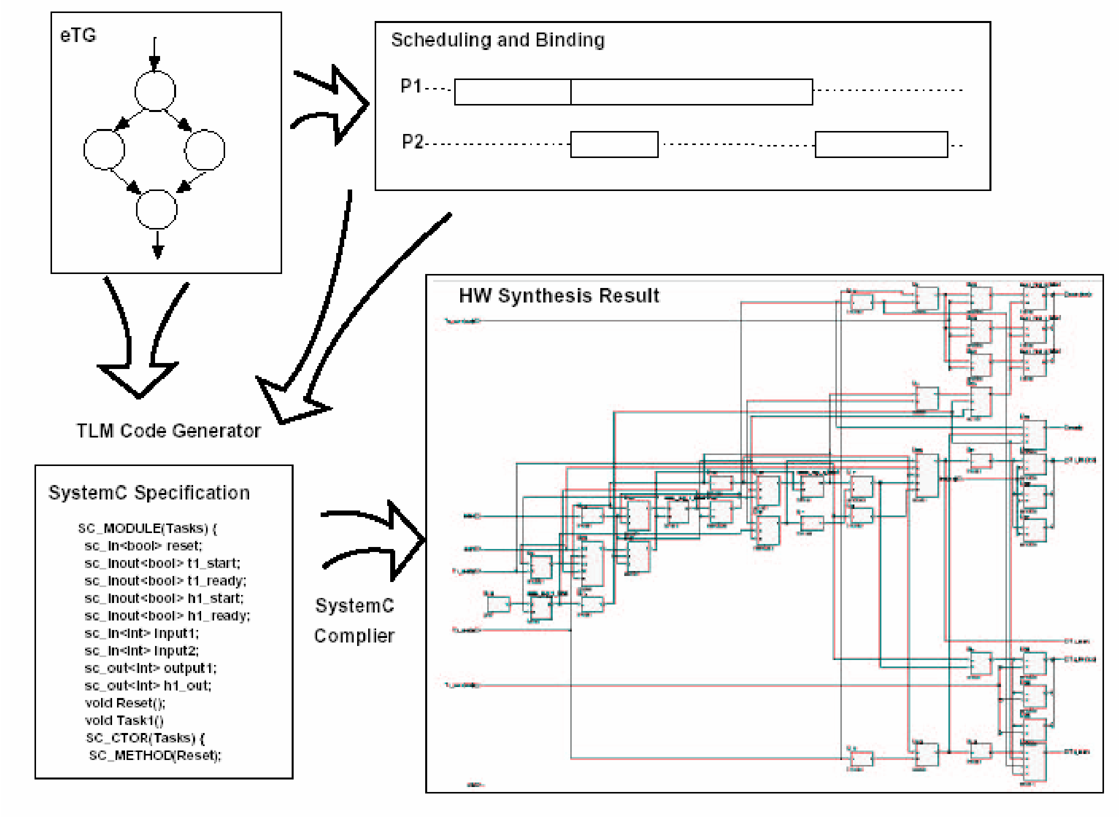
YAML provides a user friendly graphical interface to model systems under the guidelines of UML, using the SystemC and ICSP C++ class libraries. User can specify the details of SystemC and ICSP classes into the UML front end. The code generated by YAML conforms to the syntax of ICSP and SystemC classes and can be directly compiled and simulated.

The major advantage of using YAML is the ability to avoid the complex syntactic details involved in using the C++ libraries. User can generate the SystemC + ICSP code from YAML, after specifying the various details in the class and object diagram. YAML has been used to model various designs including a DLX compatible processor pipleline. The DLX pipeline code consists of around 2000 lines of C++ code. Most of which was generated automatically by YAML. It raises good results in generating and simulating the models.

YAML gives some ideas of modeling the system functional and structural information. However, it lacks of the behavioral requirement support, and hence cannot capture the requirement of control information.

3.2 Auto-generation of SystemC model from Extended Task Graphs

To model the behavior of hardware, Klaus proposed to use extended task graph.[11] Task graph gives a accurate definition of time and different model of computation are emphasized. Task graphs are a widely-spread means for the specification of embedded systems behavior. Task graphs have a well-defined execution semantic and a temporal order and other abstract modeling characteristics. A task graph represents operations and data dependencies between them. Its main features are both the modeling of control flow and a hierarchical structuring of functionality. (Figure 3)



```
SC_MODULE(Tasks) {
  sc_in<bool> reset;
  sc_inout<bool> t1_start;
  sc_inout<bool> t1_ready;
  sc_inout<bool> h1_start;
  sc_inout<bool> h1_ready;
  sc_in<int> input1;
  sc_in<int> input2;
  sc_out<int> output1;
  sc_out<int> h1_out;
  void Reset();
  void Task1()
  SC_CTOR(Tasks) {
   SC_METHOD(Reset);
```

Figure 3 An example of SystemC generation using eTG

The methodology was successfully applied to complex specification consisting of more than 200 tasks. Besides scheduling, the complexity is linear in the number of tasks and allows to handle such complex systems very easily. The produced code is quite readable using well-defined signal names derived from the specification and, as mentioned earlier, the code is synthesizable.

3.3 RoseRT to SystemC translation

Another team in our department is exploring a similar method to translate from UML to SystemC.[5] A RoseRT wrapper of SystemC has been built to produce SystemC code from restricted RoseRT design. Despite its intention as a tool for general purpose software development, RoseRT has close similarities to SystemC. Capsules in Rose RT communicate via ports and protocols just as modules communicate via ports and channels in SystemC. A capsule undergoes a state transition when a specified trigger signal arrives whereas in SystemC this corresponds to an incoming signal on one of the ports specified in the sensitivity list of an SC_METHOD. RT2SystemC translator exploits these similarities to identify and extract important sections of the C++ code generated by the RoseRT tool.

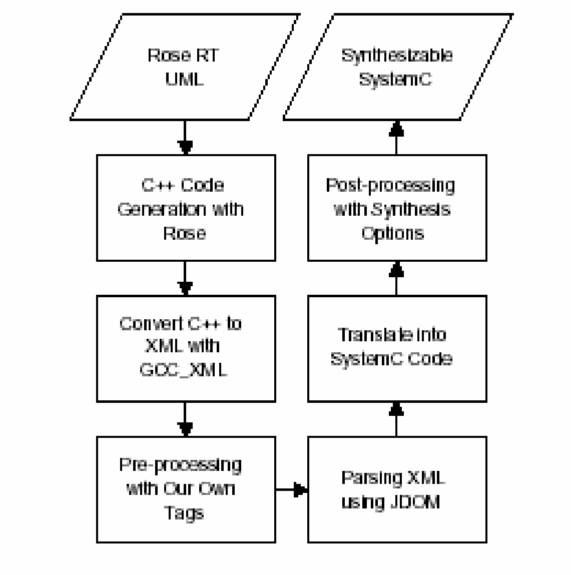

Rose RT UML
Synthesizable SystemC
C++ Code Generation with Rose
Post-processing with Synthesis Options
Convert C++ to XML with GCC_XML
Translate into SystemC Code
Pre-processing with Our Own Tags
Parsing XML using JDOM

Figure 4 Translation flow of RT2Code

RT2Code translation starts from UML model in RoseRT, and then the rose generated C++ code are further compiled into XML documents. RT2SystemC generator uses the XML documents as input, and generated synthesizable SystemC code. There are interesting similarities as well as differences between their translation and our approach. (Figure 4)

There are still some limitations in the translation. Firstly, the generation is based on the generated code of RoseRT, and it is very software dependent. Furthermore, RoseRT lacks of complex statechart support, therefore, the behavioral functionalities of larger system may not be well captured. However, there are many interesting similarities between their project and ours. By studying their project, we can add in more value to our approach.

IV: Implementation of UML2Code Translator

4.1 Introduction

In this section, the experiments on the UML to SystemC will be discussed. Currently, we have finished the work from UML model to the lower level SystemC model translation. The class diagrams and statechart diagrams are used to capture the system design information. In later part of this section, detailed implementations are discussed.

4.2 General Translation flow

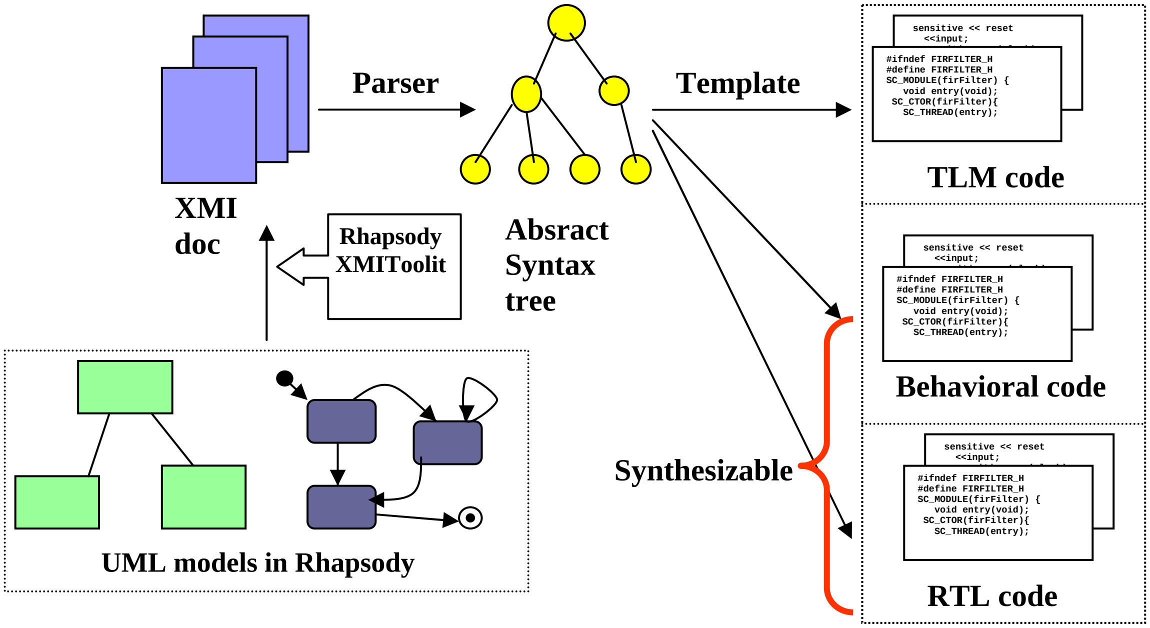


Figure 5 Translation flow of UML2Code generator

Figure 5 shows the translation flow of UML2Code. We model the system in UML with Rhapsody. The corresponding XMI documents will be generated based on the completed design model using Rhapsody XMITookit, which are the inputs of our translator.

XMITookit preserved all the model information, and it is a textual representation of the UML model. The generator will parse the XMI documents and build the internal representation of the model. The syntax tree will be further processed and some relations will be computed and stored. The syntax tree will hold all the data required to build different levels of code. By using different templates, the same model can be used to generate code in different levels: TLM level, Behavioral level and RTL code. Among the generated code, Behavioral code and RTL code can be further synthesized into hardware net-list.

4.2.1 Capturing the system Specifications

We use Rhapsody to do the system design. Rhapsody is a powerful design tool develpled by I-logic. I-Logix' Rhapsody® is a Unified Modeling Language (UML) application development platform for pervasive computing. Rhapsody allows the software engineer to model software designs graphically (utilizing UML) and simulate interaction and communication between software components. Subsequently, production-quality C, C++, and Java code is automatically generated as the design evolves. Furthermore, Rhapsody provides very powerful Statechart diagram support. It supports hierarchical states and the statechart are used to generate executable codes. Graphical animation allows design-level debugging on the host before testing the software on the target.

We capture the system specification by using Class diagram and StateChart Diagram. In the UML models, the class diagrams specify the structural information. An embedded System consists of software and hardware components as well as their communication interfaces. The software and hardware components are modeled as classes. Each classes will hind functionalities and implementation from other. The coefficients of the components are modeled as attributes of the classes. Furthermore, the operations in class are used to model the functions of the UML

4.2.2 Convert the UML model to XMI documents

It is quite difficult to process the graphical notation directly. Rhapsody provides XMI document generator called XMITookit. It can generate the text representation of the UML models. The internal representations of the UML in different platforms are quite ad hoc. But, XMI is accepted as a universal text standard representation of the UML. Therefore, we use XMI as generator input instead of graphical notation to achieve a tool-independent developing methodology. There are a few advantage of doing this. Firstly, XMI is a text document, and it has a standard format, therefore, it can be easily parsed and processed. Secondly, XMI is XML-based meta-language, and it is quite extensible.

4.2.3 UML2Code

UML2Code consists of 3 parts: Parser, Transformer and velocity engine, and each of them will process the data generated from it frontier. The translation can be divided into 3 stages: Parsing, Post-processing and code generation. Figure 6 shows a flow diagram of the UML2Code generator.

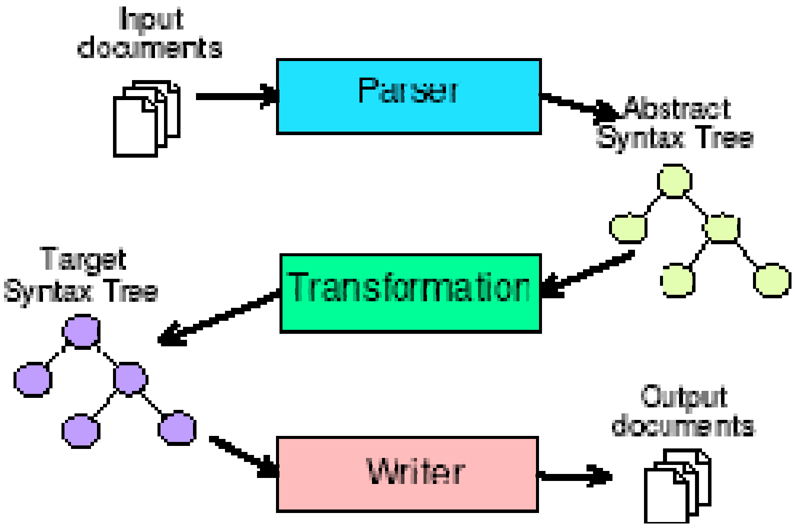


Figure 6 Translation flow of UML2Code

Parsing

Parser is used to process the XMI document and extract the information of models holding in the models. JDOM is used to parse the XMI into tree structure. By used XPATH, a XML searching engine, we are able to find the traverse the tree and find the properties of the XMI tag. Figure 7, shows the tree structure of XMI document, the parser will parse the tree and build another syntax tree base the extracted information.

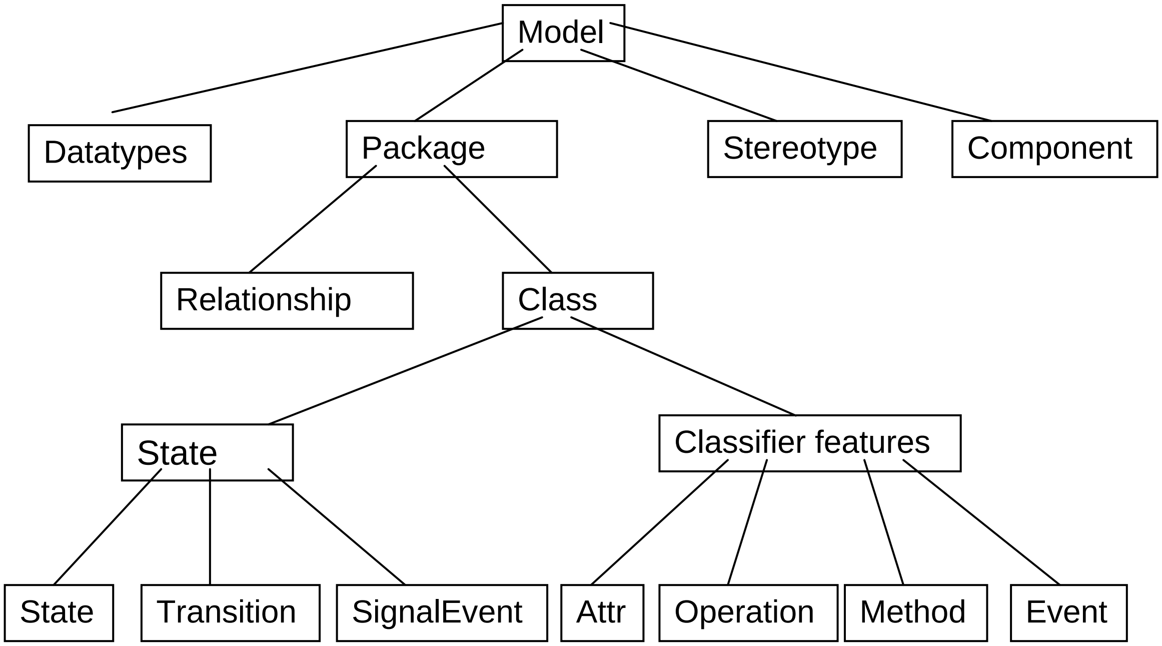


Figure 7 The internal structure of XMI documents

After parsing stage, abstract syntax tree will be built. In the generated tree, model will contains many classes and packages. Each class will hold the reference to the root state of its Statechart. And the clock information will be store in the model class.

Postprocessing

After the information of the model been extracted from the XMI documents, the data will be further process to compute some information which can not be directly got from model, such as output events.

Code Generation

To make our translator flexible, we make use of template technology. The templates are separated development from the data set. By using the same dataset, velocity engine can generated different output code while using different template set. As we was planning to generate codes in 3 levels of abstraction, this design save a lot of effort while translation from one level of code to another. On the other hand, any changes applied to the template will not affect the processing the model data. We can do iterated refinements on the templates only.

4.2.4 A proposed Design Flow

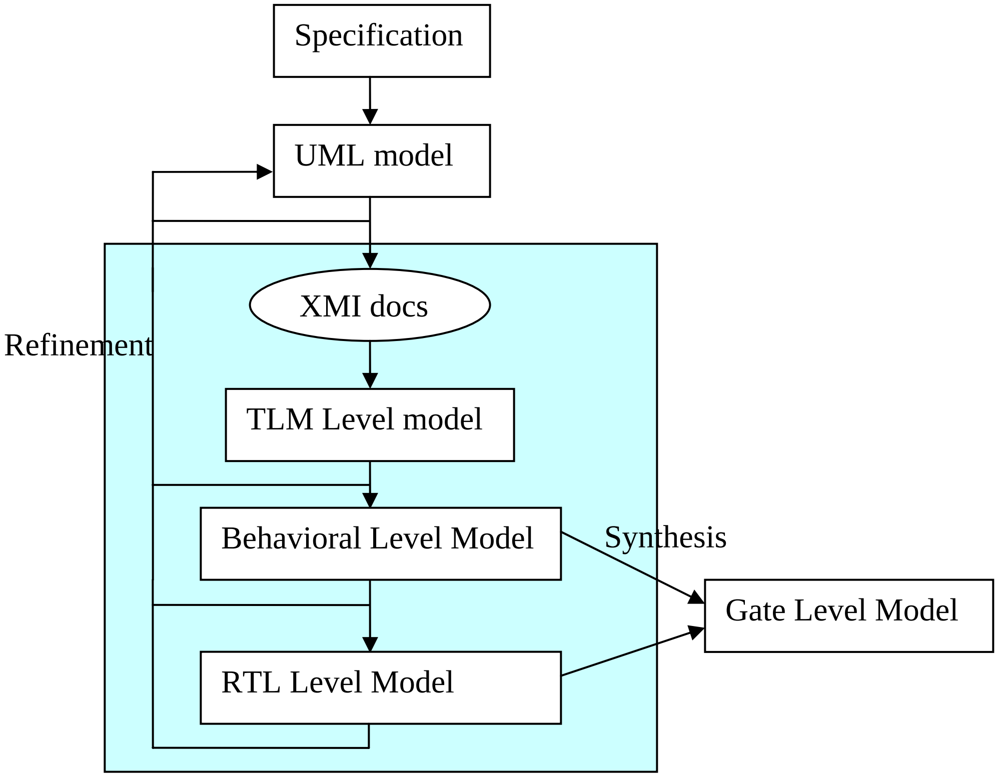


Figure 8 Code generation flow

Figure 8 shows a proposed code generation flow is illustrated. At the very beginning, the UML models are built based on the system specification. Verification and refinement can be done in several levels to test whether the requirements are captured. The system is first translated into TLM level model, which can be used for fast translation. And then the behavioral level model can be generated, which can be further compiled into gate level models.

In this Flow, the model testing and verification can be down at all the stages. Designer can enjoy a Top-down development progress.

4.3 Capturing System specification

4.3.1 System Structure and Class Diagrams

UML class diagrams are used to describe the various object classes and their relationships and associations, including inheritance and aggregation.

Classes

Classes are used to model the system components and communication interfaces. A class will have attributes and operations. For each attributes, it will have type, publicity and static status. For functions, it will have return type, arguments with types and names, as well as the publicity and static status.

To module different elements, we make use of stereotype of class. There is a mapping from SystemC elements to the UML stereotypes. To model the details of the SystemC design elements, we introduced three extensions using UML stereotypes mechanism. Table 1 shows the mapping between SystemC elements and UML stereotypes.

SystemC elements	UML stereotypess
Modules	Normal class
Interfaces	<interface>
Primitive channel	<pri_channel>
Hierarchical channel	<channel>

Table 1 Mapping from SystemC elements to UML stereotypes

Relations

The aggregation is used to model the containing information of components. If A has aggregation relation with B, then B will be modeled as component of A. Furthermore, we use association to model the relations between components. A component A has association with component B is A will call B' functions, ie., send message to B. In the example, (Figure 9) Ping is associated with Pong with undirected association, thus, Ping and Pong will communicated with each other through message sending.

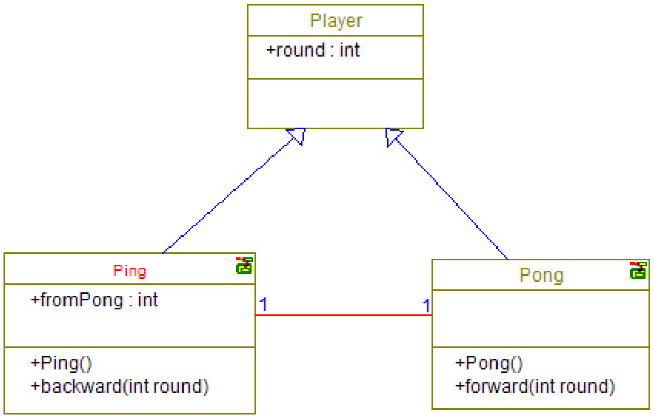


Figure 9 An example of class diagram

Figure 9 is an example of class diagram. In the example, Ping and Pong are two classes, and they are all inheriting Player class. In the translation, class diagram are used model the components' functionality and the communication relations among them.

The relations among the classes are used to model the communication relationships between the components. These relations include association and aggregations. If two components have message/signal exchange, an association will be placed in between. The direction of the association indicates the direction of the communication.

Top Class

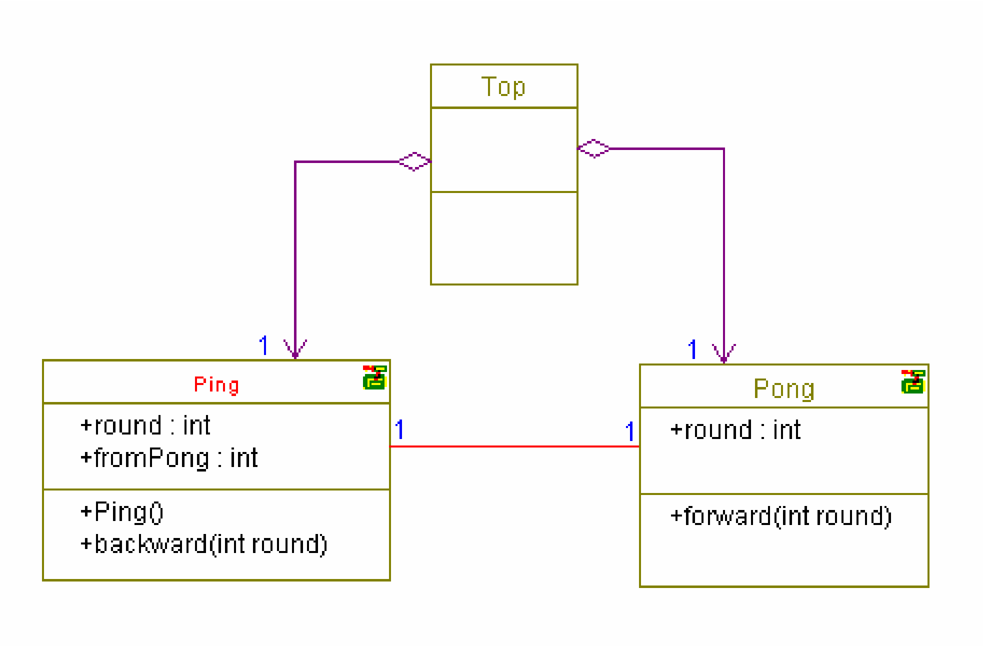


Figure 10 Class diagram of PingPong

To produce executable code, a driver class, named Top, has to be created to create all the module instances and initialize the simulation. Following is a example of main class.

```
TOP_CREATE_OBJECT(itsPing, Ping, "ping");

TOP_CREATE_OBJECT(itsPong, Pong, "pong");

SET_OBJECT(itsPong, itsPing, itsPing);

SET_OBJECT(itsPing, itsPong, itsPong);

#ifdef _SystemC

        sc_initialize();

#endif

TOP_START_BEHAVIOR(Ping, itsPing);

TOP_START_BEHAVIOR(Pong, itsPong);

#ifdef _SystemC

        sc_start();

#endif
```

4.3.2 Clock and Component Diagram

Clock setting is an essential point in the embedded system design. In general case, different hardware components will be deployed on different clocks. Therefore, handle the clock setting will be a “must” for the designer. The clock rate will affect the speed of the hardware, how the component communicate, Cost, power assumption and other important issues. On the other hand, in synthesis code, the processes will be sensitive to a certain clock. The clock rate will directly determine the behaviors.

In our project, we tried to model the clock in a simplified way. In SystemC, the default clock period is 1ns. We use 1ns as unit period, and the all the clock period must be multiple of 1ns.

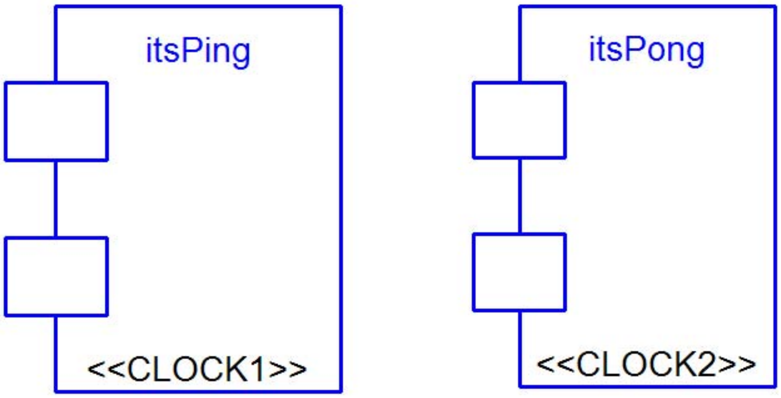


Figure 11 Example of Component Diagram of PingPong

It is quite naturally that the clock setting can be described as component property, therefore, we use component diagram to model clock settings. Here, we again make use of stereotypes. In the diagram, the components are the module instances and the stereotypes of the components are set as CLOCK X, where $X \in [1,2,...]$. In Figure 11, it shows a simple clock setting for PingPong. There are 1 instance of Ping and 1 instance of Pong. The clock period of pinger is set to be 1ns, while ponger's is 2ns. For those component without clock setting, a default clock with period 1ns will be assigned. The following is SystemC code for creating clocks.

```
Sc_clock base_CLK; //default clock

sc_clock CLOCK2(“CLOCk2”, 2, 0.5, 0, false); //clock with period 2

sc_clock CLOCK10(“CLOCk10”, 10, 0.5, 0, false);
```

4.4 Translation from UML to SystemC

In this section, we will introduce detailed implementation of the translator in TLM, Behavioral and RTL level.

4.4.1 TLM translation

Transaction Level model is a level where all the communication between components in the system is done through method calls, without synchronization, even between software and hardware components. Therefore, TLM level codes do not contain any information about clock-accuracy and timing issues. Here, transaction means the exchange of data. This level emphasizes what data are transferred from which component, and the communication among components is separated from details of implementation.

The basic elements of SystemC are modules and processes for computation, interface and channels for communication. Modules are the basic building blocks for partitioning the design.[5] Each module hides its data and operations from other modules. Modules hold the reference to other modules and interfaces. Furthermore, the modules communicate with each other through functions calls. The data will be passed as parameters. The synchronization details will be hidden. A module consists of one or more processes. Processes are the means to model the concurrent behavior.

4.4.1.1 Translation of Class Diagrams

A normal class will be translated into a *sc_module*. At the same time, the attributes and operations will be translated into attributes and methods in the module. The signature of the attributes and functions, such as the visibility (public/private), static status, will be kept without changing.

Relations

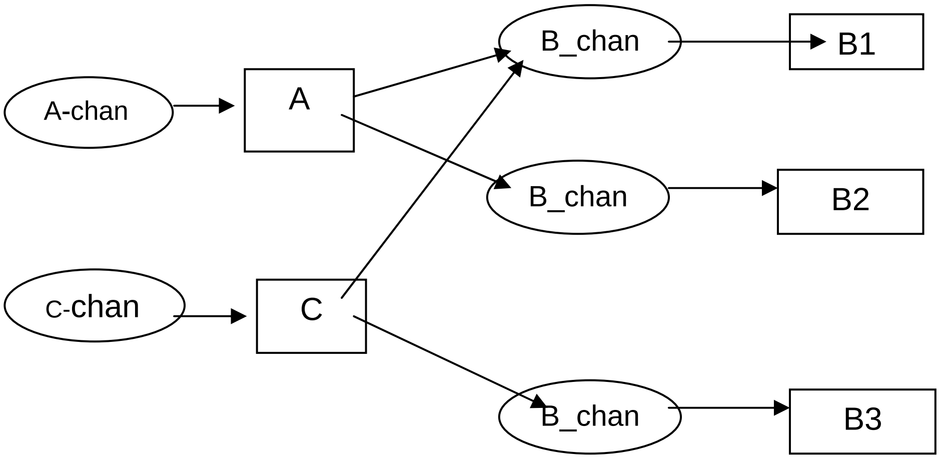


Figure 12 Port connection

Generated modules communicate with each other using sc_port. Each input events will be mapped to a sc-port. As Figure 12 shows, for each module, a sc_channel will be generated which will attached to the module. And sc_channel will provide interface of all the ports. All the communication to the module has to be performed through the channel. The generated modules can access the reference the channels through the ports, which belong to the associated modules. As showed in the Figure 12, A can access channel ttached to instance B1 and B2, while C can access channel of B1 and B3.

Inheritance

Inheritance plays a very important role in the software design. Our algorithm also provides the support of inheritance as well as multiple inheritances. In the UML model, if one class is inheriting from another class, the inherited attributes and operations will be copied down as if they are local attributes and operations. However, the polymorphism will be discarded in case that they wont have any inheritance appear in the generated class.

```
Class module_name :  public sc_module {
             sc_port<module_name> this_port;
             classname_chan *this_channel;
            //associated object;
            //associated objects channels
            //associated objects ports;
            int state;
            //attributes
            //inherited attributes
SC_HAS_PROCESS(module_name); //define the process
Ping(sc_module_name name,moduel_chan *chan) : sc_module(name)
{
   this_channel = chan;
   //initialization code
   state = initial_state;
   SC_THREAD(entry);
}
private:
   void entry();
//declaration of the local opertions
};
```

Template 1 TLM Template for header file

4.4.1.2 States Diagram Translation

General Translation

The statechart formalism has been introduced by Harel[12]. A statechart design essentially consists of states and transitions like a finite automation. In order to model depth, a state can be refined to contain sub-states and internal transitions. Two such refinements are available: AND-states and OR-states that give a state hierarchy. At the bottom of the hierarchy, basic-states are not further refined. If the system specified by a statechart resides in an OR-state, then it also resides in exactly one of its direct sub-states. Staying in an AND-state implies staying in all of the direct sub-states and models concurrency. When a state is left, each sub-state is also left, and this can be used to model preemption. Sub-states of an AND-state may contain transitions, which can be executed simultaneously. The different parts of an AND-state may communicate by internal events, which are broadcast all over the scope of events. [6] Statechart diagrams in UML allow for guards on transitions, propagated transitions, actions on transitions, actions on state entry, activities that last as long as a states, actions on exit. [13]

States

Dynamic behavior of a UML class expressed in terms of state transition diagrams of simple states is represented in TLM level as a process. In particular, TLM level process will use non-clocked process called *sc_thread*. A local variable called *state* is used to hold the current state id, and it is assigned to the value of initial state id during the initialization stage. The processes keep moving among the states until final state is reached. When the process entering a new state, it first perform the actions_on_entry. And then the reaction will be performed. In most of time, the processes stay in one state and waiting for some events. Upon receiving an event, the process will perform the guard action and change the value of state accordingly. When it exits the state, action_on_exit will be performed.

Transitions

State transitions are translated into variable assignment to *state* in main loop. Each transition corresponds to assigning a new state id to the *state* variable. The assignment will be done after the action_on_exit actions are taken.

Events

Sending/Receiving events within states diagram are translated as function calls. As we stated above, each module has a channel attached. For each input event, there will be function call for it. The parameters of the function call will be the parameters of the event. A status flag is used to indicate the event is activated. When processes are waiting for guard events, the values of the status flags are monitored. The changes to the status variables activate the transitions. When the process receives events, all the guard will be check and certain actions will be performed if the guard is true. When the guard is not true, then the process stays in the same state and waits for new events coming.

Pseudo state

Default initial states are mapped to a value that the variable state are initially assigned to. When the process starts, the process always starts running from the initial state. Final state simply cause the thread to end, when it is reached, the process will stop execution.

OR- state

In OR-state, the process always stays in exactly one sub-state. OR-states are mapped into a process running infinite loop, where a *state* variable will keep the state information. The value of the state is one of the sub-state id of OR-state. The loop will terminate when the state change to final-state.

The following is the pseudo-code for this type of basic states:

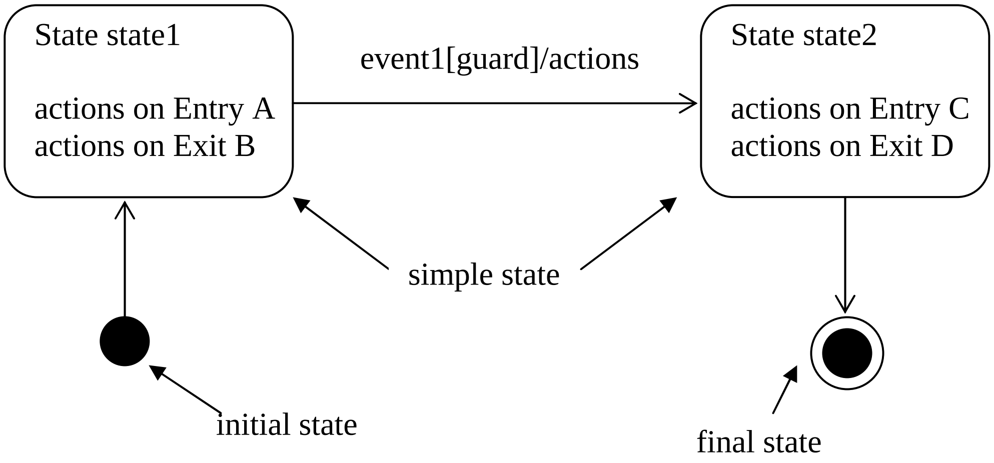


Figure 13 A example of OR-State diagram

```
While(state!=final_state)
{
    switch(state)
    {
        Case state1:
            1. Action on entry A
            2. wait for trigger events
            3. if guard is true
                    do action of transition
                    do action_on_exit B
                    state=state2(new_state_id)
            4. break
        case state2:
        …
    }
}
```

Template 2 Template of the example AND-state

AND-state

AND-state is used to model the components with concurrently executing processes. To simplify the design, we only allow one level of sub-state. A modeled component cannot have nested states and the process is static. Each sub-state of the AND-state is treated as a normal OR-state. Each sub-state of AND-state is mapped to a process running a thread.

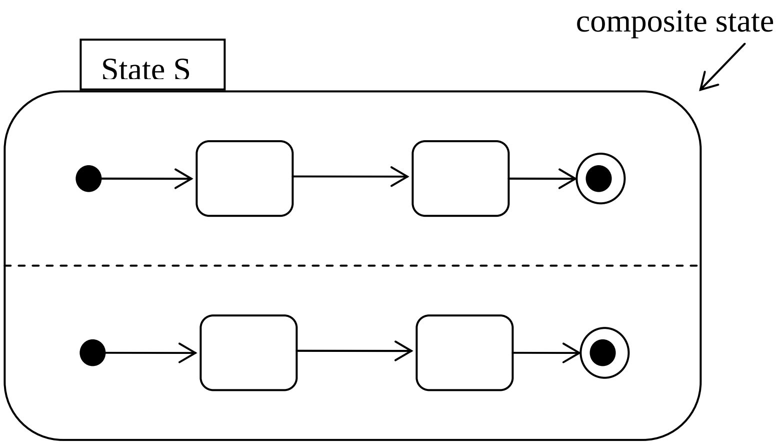


Figure 14 An example of AND-state

4.3.1.3 Top Level Class Translation

In the module, there is a class name Top, which will initialize the execution of the whole module. Top will initialize necessary modules and channels. Modules are connected to the modules it communicated with by holding the references of their channel. After all the connections are built, the simulation will be initialized with statement sc_start().

4.4.2 Behavioral Level translation

4.4.2.1 Introduction of Behavioral level

Behavioral synthesis as supported by synopsys tools consists of automation that enables design at a higher level of abstraction by synthesizing an RTL implementation from a behavioral description. Behavioral synthesis transforms untimed or partially timed functional code into fully timed RTL implementations. Because the micro-architecture is automatically generated, the designer can focus on designing and verifying the module functionality. This behavioral design flow increases designer productivity, reduces errors, and speeds verification. [19]

The behavioral synthesis process incorporates a number of complex stages including lexical processing, algorithm optimization, control/dataflow analysis, library processing, resource allocation, scheduling binding of functional units and registers, and output processing.

This process starts with a high-level language description of the desired behavior of a module including I/O behavior and computational functionality. A number of algorithmic optimizations such as constant folding and common sub-expression elimination are performed to reduce the complexity of the result. The description is then analyzed to determine the fundamental operations required and the dataflow dependencies between them.

Behavioral Level coding style

Behavioral level code is more restricted than TLM level. It has to take clock information into consideration. Besides, the synthesizable requirement makes the code even harder. Generally, the behavioral code has following differences compared to TLM level code.[20]

- Clock thread rather than thread
- Only signals are allowed for communication
- Cycle balance in the conditions and loops

The only process type allowed for behavioral synthesis is the *sc_cthread* one. This is basically a process that is sensitive to clock only and to no other signal. In addition, a reset signal must be declared by means of the SystemC watching statement.

The simulation semantics of the wait statements inside a *sc_cthread* is to suspend the execution until the next active clock front. From the point of view of the synthesis, these statements define the operations that have to be performed in a single clock cycle or, equivalently, they are used to define the temporal behavior of the system with per cycle accuracy.

There are two types of scheduling modes in behavioral level: cycle-fixed and superstate-fixed scheduling modes. Cycle-fixed scheduling requires that I/O behavior matches behavioral description, cycle by cycle. That is all the bocks of instruction between two wait statements are scheduled to be executed in exact one cycle. For example, if a read and a write is separated by two wait(), then exactly two cycles are needed. Superstate-fixed scheduling mode requires that relative order of I/O operation is preserved based on behavioral description, but not the exact cycle in which they happen. The scheduler can split sequences of instructions to be executed in a single clock cycle over many cycles. For example, if a read and a write is separated by two wait(), then at least two cycles are needed, and there could be more than two cycles been taken to finish the statements. In our translation, the superstate-fixed scheduling mode will be used.

An important set of constrains imposed to the behavioral descriptions to be synthesized are related to the position of the wait statements with respect to control structures such as loops and conditional statements. Generally, if one branch has a conditional (if…else, switch…case, or ?: operation) has at least one wait statement, then place at least one wait statement in each branch. [19]

The following rule has to apply for the balance of clock

- Place at least one wait statement in every loop except unrolled for loops.
- Place at least one wait statement between successive writes to the same output.
- Place at least one wait statement after the reset action and before the main infinite loop. Do not include either a conditional branch or a rolled loop in the reset behavior description.
- If one branch of a conditional has at least one wait statement, place at least one wait statement in each of the other branches, including the default branch and implicit else conditions.
- Place at least one wait statement after the last write inside a loop and before a loop continue or an exit.

4.4.2.2 Class Diagram Translation

Behavioral level SystemC models the system in functional and behavioral. SystemC provide a subset of the semantics, and by following certain coding rules, the resulting code can be synthesized using ConCentric tools. The Synopsis CoCentric tool can use the synthesizable code to produce the gate level models, which is just one step to hardware. Different from the TLM level translation, the code in the operation bodies must be synthesizable. Generator will not modify the code much during the translation. Besides, there are some unsupported primitive types, such as floating point types, and the attributes is not allowed to use such types. Thirdly, pointers and pointer operations are not allowed in the synthesizable code, although it is very commonly used in the C/C++ context.

Classes

Similar to the TLM level translation, each class is mapped into a module. All the modules will have incoming and outgoing ports. In SystemC language, there are three types of primitive type of port, namely, *sc_in<type>, sc_out<type> and sc_inout<type>*. *Sc_in* is a type of read only port. *Sc_out* is used for write only operation. *Sc_inout* are used for two-way communication.

Stereotypes are kept in the behavioral level translation. <pri_channel> and <channel> are mapped to module. In difference to from the TLM level, process in behavioral level communicates with each other using signals. *Sc_channel* and *sc_interface* is not synthesizable, and pointer operations are not allowed in the behavioral code. All the interface function calls in the channel are mapped to signal sending/receiving. Function call consists of function name, parameters, and return types. A Boolean port with name <function name>_ready is used to indicate the function is being called. Each parameters is translated into a signal with type same as the parameter. Return value is also sent by signal with type being the return type, and a flag signal is used to notify the return value is ready. On the sender side, one Boolean outgoing port and a few other ports for sending the parameters, and an incoming port (if return is not void) is used to get the return value. On the receiver side, one Boolean incoming port and a few other ports for receiving the parameters, and an outgoing port (if return is not void) is used to send the return value. On the function calls, the flag signal is set to be true, and at the same time, the parameters are sent out through the ports. After that, sender will run an infinite loop and wait for return value to be ready. Receiver are running infinite loop and waiting for the call, and when the flag is set to be true, the values of the parameter will be retrieved. When return value is ready, the *return_ready* signal will be sent out and the sender will get the return value.

The module can be considered as a black box and only incoming and out going signals (interfaces) are visible(Figure 15). Behavioral level modules hide the details of implementation and functionality from each other, and they only provide interface for sending/receiving signals.

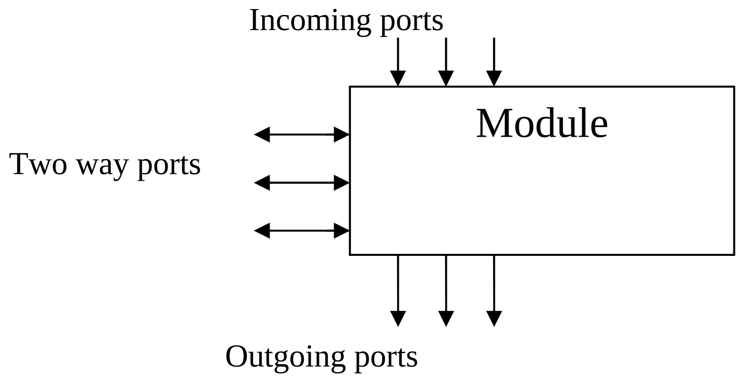


Figure 15 Black box view of module

In behavioral level code, clocked thread, sc_cthread, is used to define the processes. Synchronization issues have to be considered during the translation. Currently, there is no clock definition in the UML model. Therefore, we generate the clock for all the modules. In SystemC, sc_clock is a special signal used to model the clock. And a global clock is used to synchronize all the modules. In other words, all the incoming clock ports, *sc_in_clk*, are connected to the same *sc_clock* signal. All the processes are sensitive to the positive edge of the clock by default. Besides, a global reset signal is connected to all the reset port, and the processes will be reset if the reset signal is set to be true. The above designs are auto-generated, and in the further work, some improvement will be proposed to handle more complex situation. Template 3 shows a general structure of the header file of behavioral level module.

```
Public SC_MODULE(moduel_name) {
sc_in<bool> reset;
sc_in_clk CLK;
//code for define incoming and outgoing ports
SC_SCTR(module_name)
        {
                //Initialization code
                SC_CTHREAD(entry, CLK.pos());
                watching(reset.delayed==true);
        }
private :
   void entry();
}
```

Template 3 template for Behavioral level module class

Association and Inheritance

Associations are mapped into connections. Connection means that the signals that connect with two module. As we mentioned, modules can be treated as black boxes, and the signals are used to connect the ports. Association relations are used to model the connections in high level. Therefore, each associations maps to a connection. During the initialization stage, the connections are made by referencing to the associations.

For inheritance, behavioral translation use the same method to handle it as the TLM level. The ancestral operation and attributes are copied to the current class. The module works as if there is no super class attached.

4.4.2.3 State Diagram Translation

General Translation

The translation in behavioral level has no much difference from TLM level translation. The execution of generated modules uses FSM to model. Each process is running an infinite loop. An operation called entry contains the main loop for switch the state. By switching the states and waiting for certain events, process will perform the correct behaviors.

State

Same as TLM translation, a state variable called *state* is used to store the current state information. When the value of *state* changes, the state of process changes as well. The clocked processes are sensitive to the clock, and running an infinite loop until final state is reached. Furthermore, wait balance is carefully generated.

Events

Events are mapped to signals. As *sc_event* is not synthesizable, and it is replaced by signal sending. Each event contains events name and parameter. Correspondingly, a flag signal with Boolean type is used to indicate the event is ready. While, each parameter need same type of signal to send/receive.

4.3.2.4 Top level class Translation

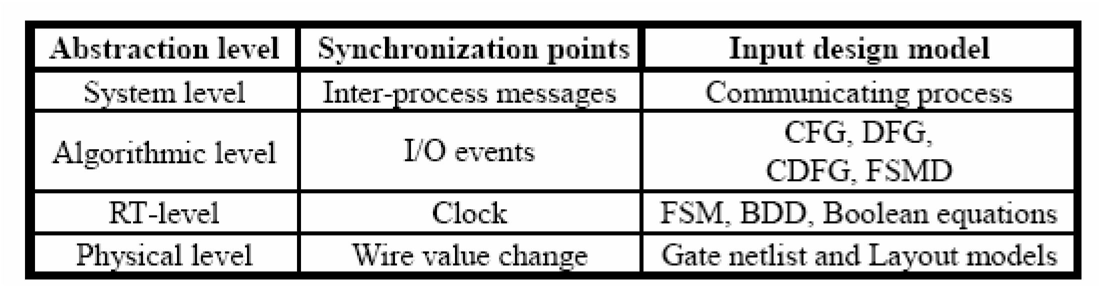
Top level class is used to initialize the while simulation. It plays three roles: create objects, build connection among object and start the simulation. Based on the object diagram, the new instances of classes are created in Top. To connect the modules, we need to connect the corresponding ports. Signals are created based on the communication events. Furthermore, senders' and receivers' ports are connected through these signals. After all connections are built, initial signals are sent to the modules to initialize the module. Finally, sc_start is called to start the simulation.

4.4.3 RTL Translation

4.4.3.1 Behavioral and RTL Synthesis

As we go from the highest to the lowest abstraction level, diverse synchronization points and design models are used, as shown in Table 1. On the system level, we work with communicating processes that synchronize through message exchange. After partitioning [22], each process could be represented at the algorithmic level by a control/data flow graph that synchronizes through I/O events. This may also be represented using a finite state machine with datapath (FSMD) model. Behavioral synthesis takes the flow graph and produces an RTL model. This model is generally represented as a controller/datapath architecture. At the RT-level, data transfers are synchronized at clock cycle boundaries. RTL and Logic synthesis will map controller and datapaths components to a cell library to produce a gate netlist. Finally, layout synthesis will produce the final chip layout. On the physical level, wire value changes define the valid data. Abstraction level Synchronization points Input design model

System level Inter-process messages Communicating process Algorithmic level I/O events CFG, DFG, CDFG, FSMD RT-level Clock FSM, BDD, Boolean equations Physical level Wire value change Gate netlist and Layout models Table 2 - Synchronization and design models on different abstraction levels



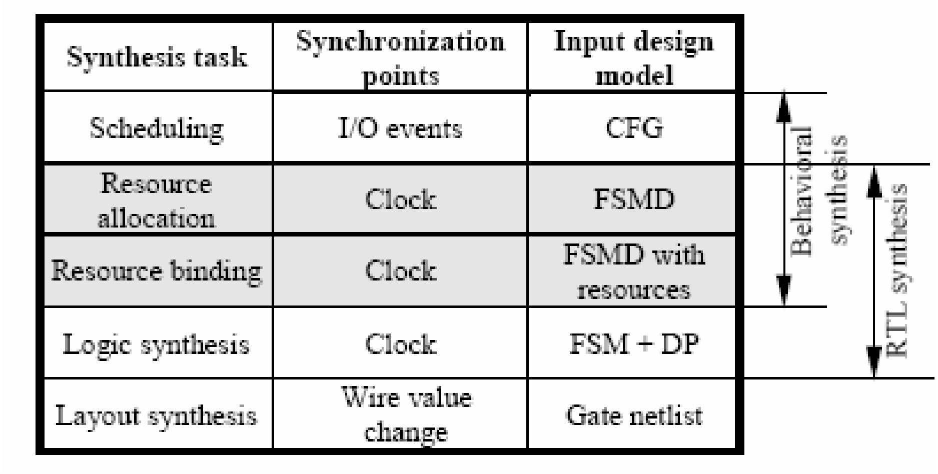
Abstraction level	Synchronization points	Input design model
System level	Inter-process messages	Communicating process
Algorithmic level	I/O events	CFG, DFG, CDFG, FSMD
RT-level	Clock	FSM, BDD, Boolean equations
Physical level	Wire value change	Gate netlist and Layout models

Table 2 Scope of Behavioral and RTL Synthesis

Table 2 details the different synthesis tasks executed by behavioral and RTL synthesis tools. Design models are also detailed for each task. This table shows clearly that there could be a functionality overlap (on gray) between behavioral and RTL synthesis tools.

Design input models for RTL synthesis goes from cycle true FSMDs to completely specified architectures. All these models have something in common: they synchronize at clock cycle boundaries, i.e., are clock cycle accurate. This level of precision is only attained after behavioral synthesis' scheduling task.

The clock cycle accurate FSMD design model is the key for behavioral and RTL synthesis integration. It can be seen in Table 2 that this model could be used for doing resource allocation and binding in the region of functionality overlap.



Synthesis task	Synchronization points	Input design model
Scheduling	I/O events	CFG
Resource allocation	Clock	FSMD
Resource binding	Clock	FSMD with resources
Logic synthesis	Clock	FSM + DP
Layout synthesis	Wire value change	Gate netlist

Behavioral synthesis
RTL synthesis

Table 3 Scope of behavioral and RTL synthesis(2)

Resource allocation and binding produce a FSMD with resources. Storage allocation and binding assign registers/memories to variables and arrays. Functional unit allocation assigns operators to operations. Interconnection allocation defines paths between storage and computation cells. Finally, a controller/datapath architecture is created. Information about resources might not be exhaustive, since we could be possibly interested in doing resource allocation and binding in two stages. For instance, complex operations could be treated by behavioral synthesis while the simple ones are transferred to RTL synthesis. In this case, we must be able to translate a FSMD with partial resource information in a format acceptable by RTL synthesis. More details on this flexible interface between behavioral and RTL synthesis will be given in the next section. Complex operations, i.e., the ones that need a data-dependent number of clock cycles to execute and multicycle operations are not allowed in the FSMD clock cycle accurate design model. There are two possible solutions to deal with this problem. The first solution is to consider complex operations as a procedure call and associate them to external functional units. Procedure calls will be used to start these external functional units and get their results [22]. Each procedure call must take only one cycle. In this case, procedure call needs to be handled by the scheduler. The second solution is to describe complex operations by procedures that use only simple operations. Then procedure calls could be expanded inline [22] and scheduled with the rest of the description [22].

4.4.3.2 RTL level Translation

We intend to generate RTL level of SystemC code. Behavioral level template was further refined and archive the RTL level template, which can be used to generated synthesizable RTL code.

Different from Behaviroal code, RTL systemC uses SC_METHOD instead of SC_CTHREAD. SC_METHOD is another kind of process. Unlike threads, SC_METHOD can not be suspended, and all the statements will be executed from the beginning of method to the end.

Similar to the Behavior level translation, each class is mapped into a module. All the modules will have incoming and outgoing ports. In SystemC language, there are three types of primitive type of port, namely*, sc_in<type>, sc_out<type> and sc_inout<type>*. *sc_in* is a type of read only port. *sc_out* is used for write only operation. *sc_inout* are used for two-way communication. Beside ports definitions, the SC_METHOD process will be sensitive to some signal, eg. clock.

```
public SC_MODULE(moduel_name) {
sc_in<bool> reset;
sc_in_clk CLK;
//code for define incoming and outgoing ports
SC_SCTR(module_name)
        {
                //Initialziation code
                SC_METHOD(entry);
                sensitive_pos<<CLK;
        }
        private : void entry();
}
```

Template 4 RTL template for header file

The main changes are applied to the Finite state machine. Different from the Behavioral code, the process in RTL level was activated by the signals. Therefore, we do not need the loops for waiting the events. Besides, the *wait()* statements cannot be used in SC_METHOD. A method called entry will sensitive to clock and perform the actions when the process is activated. Moreover, the initialization statement cannot be placed at the beginning of the method. The initialization and the reset action will be put as the default action for the FSM when the reset is on. The following is the template to create *entry* method for RTL models.

```
void entry (){
if(reset.read()){
    switch(state)
        {
                for each case:
                case state_id:
                        //Action on entry
                        //wait for trigger events
                        if guard is true
                            do action of transition
                      do action_on_exit
                       state=newstate;
                Case …
                Default:
                      if(reset.read()==false){
                           state = 61;
                            round = 5;
                     }
         }//end of switch
   }//end of if
}//end of entry
```

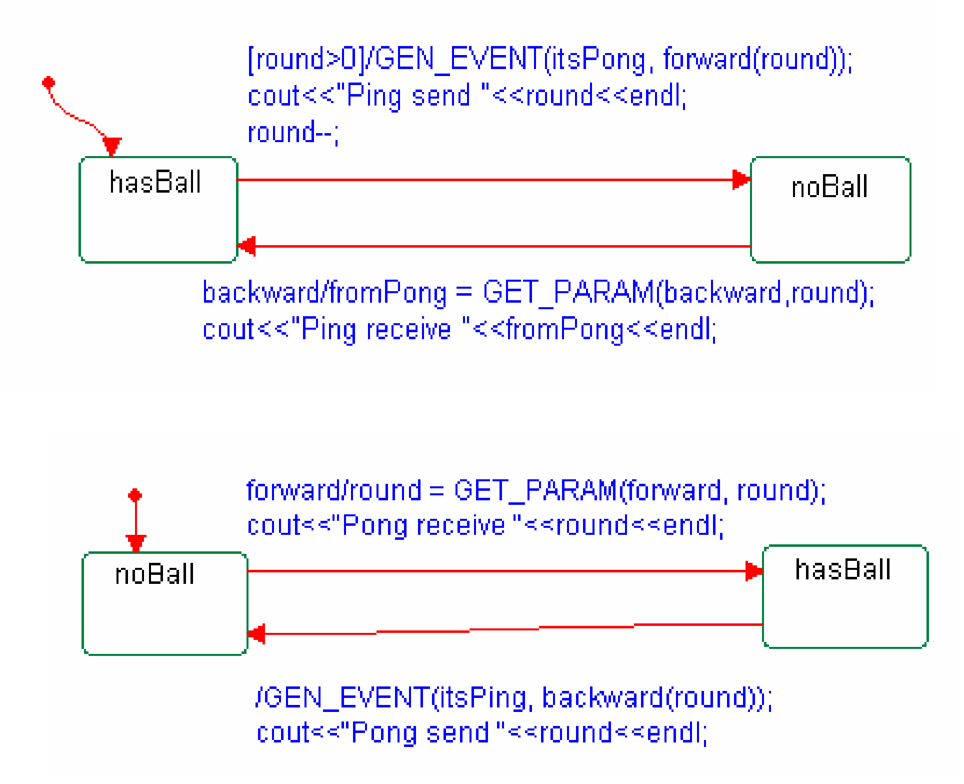
Template 5 RTL template for method entry

V: Examples and Case Studies

In this section, we show 2 examples that we have implemented.

5.1 PingPong

PingPong are taken from Rhapsody demos. It shows a small communication system consists of two components, namely: Ping and Pong. Ping is trying to send the ball to Pong, and when Pong receives, it simply sends it back. Round is used to control the rounds that they send the ball. Figure 16 shows the statechart diagrams of Ping and Pong. In the Diagram, GEN_EVENT is used to generate events, and *forward* and *backward* are two events in this diagram. Both of them have a parameter, called round.



```
[round>0]/GEN_EVENT(itsPong, forward(round));
cout<<"Ping send "<<round<<endl;
round--;
```

```
backward/fromPong = GET_PARAM(backward,round);
cout<<"Ping receive "<<fromPong<<endl;
```

```
forward/round = GET_PARAM(forward, round);
cout<<"Pong receive "<<round<<endl;
```

```
/GEN_EVENT(itsPing, backward(round));
cout<<"Pong send "<<round<<endl;
```

Figure 16 StateChart of Ping and Pong

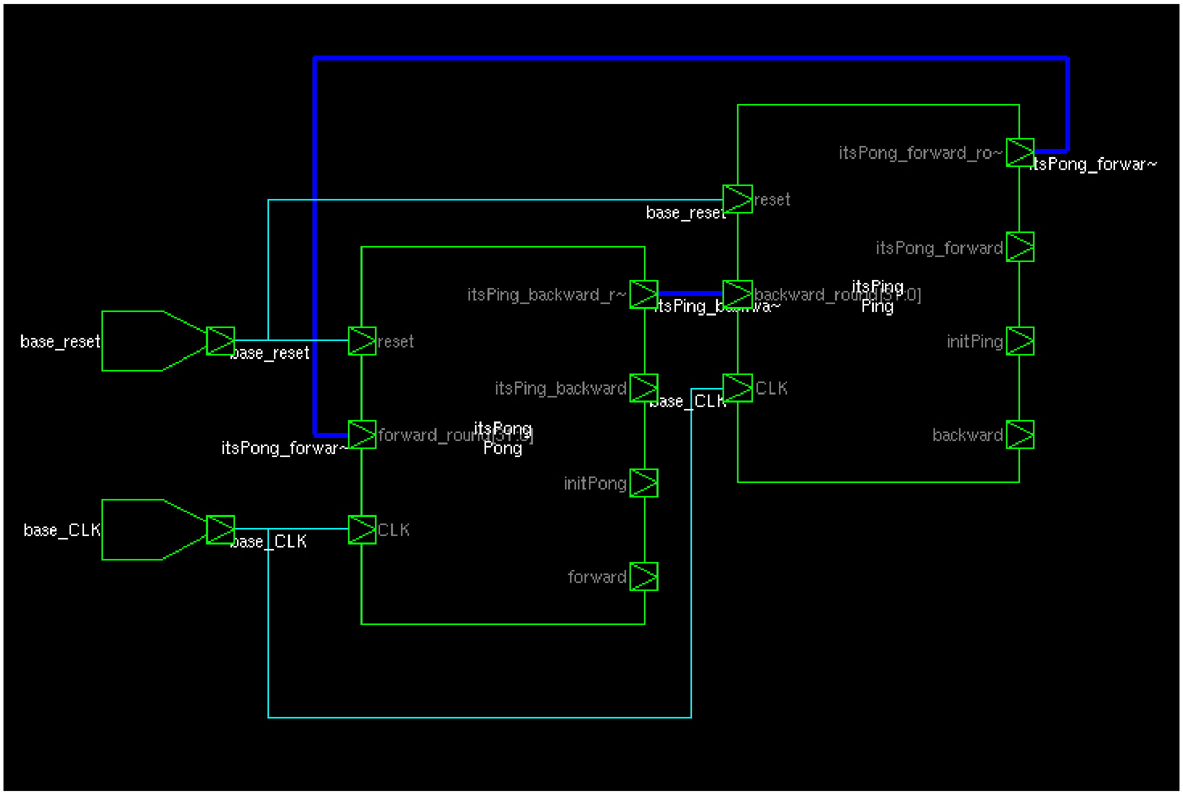


Figure 17 Synthesized Hardware

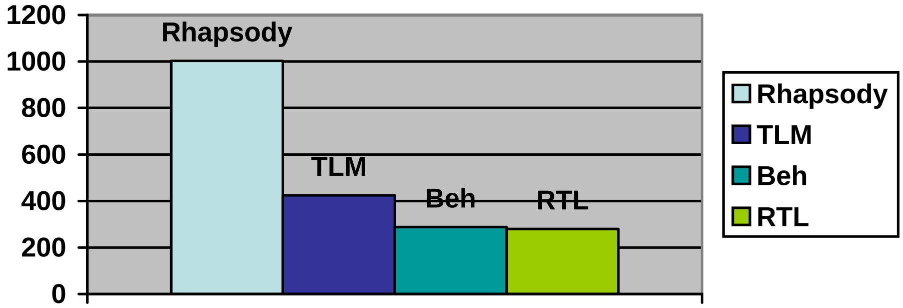
To compare the generated code execution speed, we conduct test on the generated TLM level and Behavioral level code. The test was conducted under linux environments, using Ping Pong example by running 5000 rounds. Table 4shows the simulation result.

	Execution Time	Simulation Time
TLM	66439µs	50002ns
Beh	935100µs	500004ns
RTL	658564µs	200001ns

Table 4 Test results of PingPong example

Execution time is defined as the time in the real world that it takes to finish the execution. While simulation time is the time reported by the SystemC simulator, and it gives the information about the cycles needed for the simulation. Based on the results, we can see that Behavioral level code takes 10times more time than TLM code to complete the same number of rounds. In general, sc_cthread is much slower than sc_thread. Besides, extra wait makes the execution of Behavioral code runs even more slowly. RTL code is slower than TLM level code, but faster than Behavioral level code. We found that TLM is suitable for simulation and verification, because it is the fastest among the 3. On the other hand, RTL can be the last level before the design being converted into hardware. RTL is more cycle-accurate and relatively fast in the speed.

Second statistics is collected based on the generated code length.



Rhapsody	1003
TLM	424
Beh	288
RTL	280

Figure 18 Comparison of generated code length

Rhapsody	**10**
TLM	**7**
Beh	**5**
RTL	**5**

Table 5 Number of generated file

Figure 18 and Table 5 shows the code length and the file number generated from Ping Pong example by Rhapsody and our tool. We can see that the length of Rhapsody generated code is longest as some Rhapsody dependent code is generated. RTL level code seems to be shortest and most compact among three.

5.2 Practical application: Software Radio

5.2.1 Introduction to Software Radio

A software radio is a radio whose channel modulation waveforms are defined in software. [20] That is, waveforms are generated as sampled digital signals, converted from digital to analog via a wideband DAC and then possibly upconverted from IF to RF. The receiver, similarly, employs a wideband Analog to Digital Converter (ADC) that captures all of the channels of the software radio node. The receiver then extracts, downconverts and demodulates the channel waveform using software on a general purpose processor. Software radios employ a combination of techniques that include multi-band antennas and RF conversion; wideband ADC and Digital to Analog conversion (DAC); and the implementation of IF, baseband and bitstream processing functions in general purpose programmable processors. The resulting software-defined radio in part extends the evolution of programmable hardware, increasing flexibility via increased programmability. And in part it represents an ideal that may never be fully implemented but that nevertheless simplifies and illuminates tradeoffs in radio architectures that seek to balance standards compatibility, technology insertion and the compelling economics of today's highly competitive marketplaces.

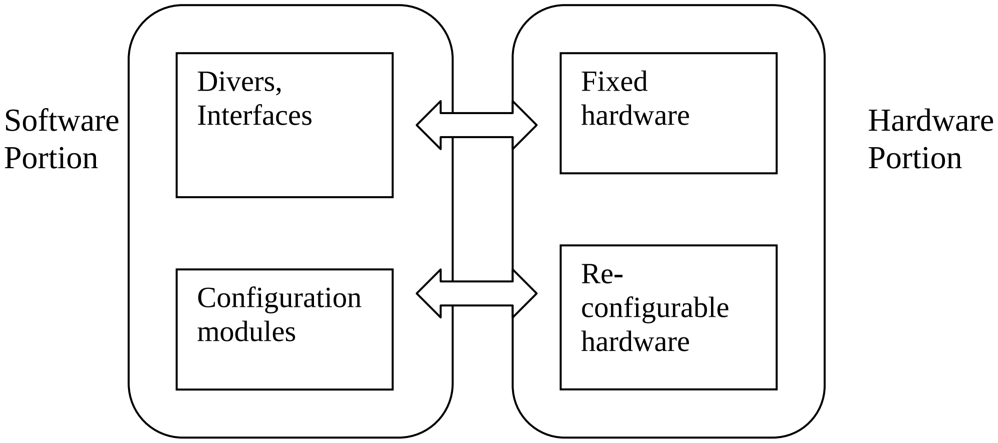


Figure 19 A general structure of software radio

Figure 19 shows a general structure of software radio. It is system where the hardware functionality and behaviors can be changed dynamically by software portion. Therefore, the hardware and software are tightly related to each other. Besides, the software/hardware partitioning as well as the optimization has to be considered during the design stage.

In fact, the most important feature of software radio is the flexibility. The most of components can be either implemented in software or hardware. In the case that designer wants to modify a system with different software/hardware resource configuration, it is not worth to build a completely new design model. However, our approach is perfectly suitable for software defined radio design. At early stage, the resource allocation will not be considered for model simulation and verification. With different settings, a single model can produced different implements. This will save a lot of effort in hard coding while developing software radio system with the same structure and different resource allocations.

Besides, software radio system requires software/hardware co-design. As we mentioned above, the software defined radio system use software to control dynamically change the hardware behaviors. The relation between software and hardware are very tight. As a main feature, our approach can ease user with graphical notations to handle the complex system. User will not be troubled on coding and debugging on the software/hardware connections and interface design, which will be automatically generated by our tool. software/hardware partition can be performed in later stage. This enables optimization to be done. User can try with different partition and test them with the generated code to achieve optimization. Moreover, test can be done directly. As we mentioned, the test code can be generated automatically. This is make system verification and validation much easier. Beside, the cost will also be reduced. Finally, IP reuse can be applied during the development. It is very costly to build a big system from nothing. There are only a few basic components in the system. By using UML notation, we can model the basic building blocks first, and by changing, extending, connecting and combining the basic blocks, the complex system can be built with much less effort.

5.2.2 DDC 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 extract the signal of interest from the incoming data stream. Our implementation closely follows the MATLAB example in Xilinx's system generator (see Figure 20).

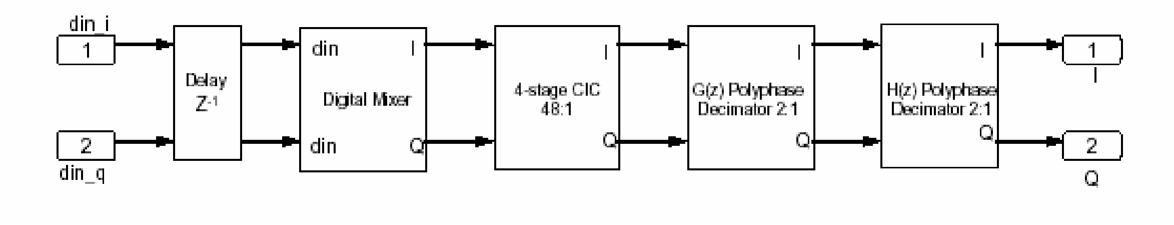


Figure 20 Block Diagram of DDC for GSM

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.

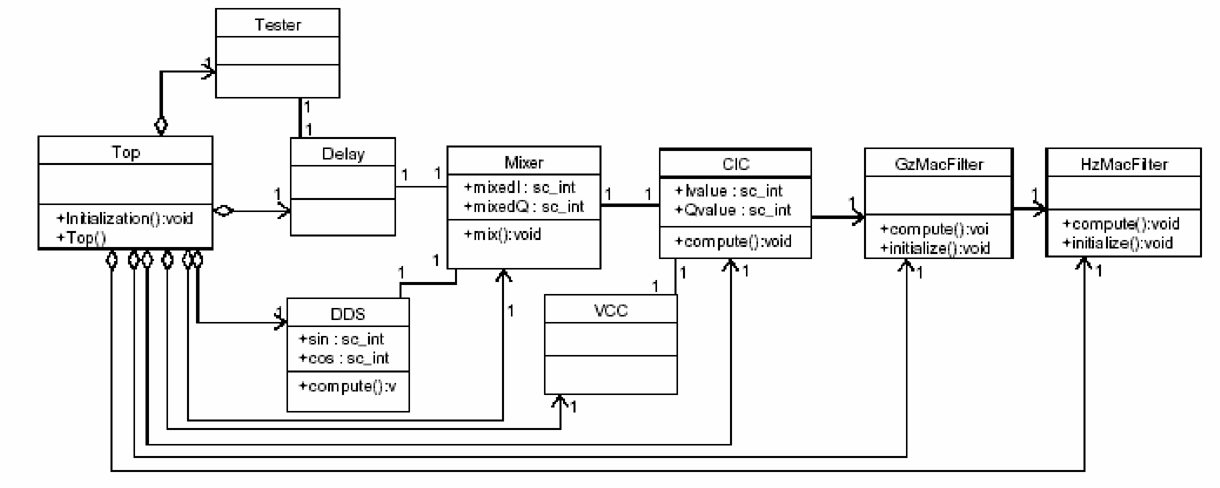


Figure 21 Class Diagram of DDC

Each of the components is mapped into a module, and the data is sent through the chain by events (see Figure 21). 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 net-list using Synopsys tc6a_cbacore library, which targets cell-based array architectures [21]. 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. Table 7 shows the Area statistics of RTL generated code.

	FIR (Synopsis) (S)	FIR (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	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%

Table 6 Area statistics for FIR component implemented on cell-based array architecture

	FIR(DDC) in RTL
Number of ports:	261
Number of nets:	10013
Number of cells:	4213
Number of references:	114
Combinational area:	389431.6563
Noncombinational area:	26958.01563
Net Interconnect area:	1469243.625
Total cell area:	416176.5938
Total area:	1885633.25

Table 7 Area statistics of RTL generated code implemented on cell-based array architecture

VI: Conclusions and Further works

6.1 Summary

In this project, we explored a new methodology to develop real-time system with use of UML-notations. We use Class diagram and statechart to model system structure and behaviors. Rhapsody provides powerful support of states diagrams and generates XMI document from the model. Using XMI as input, our auto-generate generate executable SystemC code. The generated code can be used for testing and simulation. With certain restriction in the coding style, the generator can even produce synthesizable code, which can be further complied into gate level model.

Through the experiment from the UML models to simulated and synthesized SystemC the following points are quite interesting for us:

- UML are very good at capturing and formalizing the initial design requirements
- Target architecture model can be directly associate to the design specification model
- Class and state diagrams are all dedicated to simulation or synthesis without having to re-formalize the semantics.

In fact, the early system modeling and simulation is heavily software oriented, and all the functional behaviors can be well described using UML. A directly mapping from description to system level implementation has already been established. However, this is far away from our exception, and more effort is need to achieve a UML wrapper of entire system design. We are aiming to build a UML profile of system models which allows

- Leveraging of the informal expressiveness of the UML to capture the requirements at the early stages

- Early modeling of requirements using formal executable UML Models, without committing to system partitioning to Hardware/Software.
- Follow the core OO Method of step-wise iterative refinement to take the behavioral UML simulation models into executable RTL UML Models code generation into appropriate languages for each levels for Implementation model.
- Rapid prototyping based on the models and automatically generating the testing and validation models and codes
- IP reusing and portioning optimization

To achieve the goals, some future work has to be done, and we will discuss them in the rest of this chapter.

6.2 Future works

Current generator is only concerning with class diagrams and state diagrams. As we known, besides class diagram and statechart diagram, other UML diagrams also plays a important role in system design. Therefore, exploring the use of other UML notations will be our next step towards the UML profile.

Besides, component diagrams, sequence (collaboration) diagrams can be used to perform test case design. In software design, these diagrams are often used to generate executables. By specifying calling sequence and parameters, the test case are easily constructed. The same idea can be applied to construct testing on the design.

Moreover, deployment diagrams are found to be suitable to perform software/hardware partitioning. In conclusion, we believe that the more UML notation we can handle, the better that system requirement can be captured.

Up to now, we have only concerned with hardware design. However, further extension can be made such that UML2code generator can generates the co-design codes. In another word, the tool can generate software and hardware components as well as the communication interfaces. UML is originally used for software design. There will be no difficulty to model software part. Furthermore, SystemC can model both software and hardware components. Software/Hardware co-design will be affordable with UML itself.

Traditional co-design approach requires user to partition the model in advance, and expert does the partitioning. Therefore, the partition may not be optimized. In our approach, we can partition the model by adding partition information. By changing the partition information, we can easily change the generated hardware and software. This will reduce the refinement stage of partitioning. Designer can compare simulation and synthesis results the generated from different partitioning to find the optimized solution. Furthermore, as a trend, software and hardware boundary are getting blur now. [21] Software/hardware co-design has increasing importance, especially ease the user to handle the complex system. We believe that with this extension can make our project more valuable and practical.

Testing is very costly. Traditional test requires carefully design on the tests manner, and it may cause difficult to handle the software-hardware boundary. However, in our approach, test code can be directly generated with user defined Top level class. The executable code will be auto-generated regarding hardware/software issues. Furthermore, our target language, SystemC, have its own simulation kernel. Testing can be easily performed in different levels. We believe that our approach can reduce the testing cost dramatically.

Bibliography

[1]. S. Klaus, S. Huss, T. Trautmann. Automatic Generation of Scheduled SystemC Models of Embedded Systems From Extended Task Graphs. Proc. Int. Forum on Design Languages, Marseille, France, September 2002.

[2]. Pierre Boulet, Jean-Luc Dekeyser, Cedric Dumoulin, Philippe Marquet. MDA for SoC Embedded Systems Design, Intensive Signal Processing Experiment. SIVOES-MDA workshop at UML 2003, 20-24 October 2003 San Francisco

[3]. Martin, G. UML for Embedded systems Specification and Design: Motivation and Overview. Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02), p. 773 -775

[4]. Stephan Bourduas, Ferhat Khendek and Daniel Vincent. From MSC and UML to SDL. COMPSAC. P.153-158 2002

[5]. W.H. Tan, P.S. Thiagarajan, W.F. Wong, Y. Zhu and S.K. Pilakkat Synthesizable SystemC Code from UML Models. National University of Singapore, School of Computing. 2004

[6]. UML homepage. http://www-306.ibm.com/software/rational/uml/

[7]. SystemC homepage. http://www.systemc.org/

[8]. Grant Martin , Luciano Lavagno , Jean Louis-Guerin. Embedded UML: a merger of real-time UML and co-design, Proceedings of the ninth international symposium on Hardware/software codesign, p.23-28, April 2001

[9]. Karsten Lüth, Thomas Peikenkamp, and Jürgen Niehaus. HW/SW Cosynthesis using Statecharts and Symbolic Timing Diagrams. In Proceedings of the 9th IEEE International Workshop on Rapid System Prototyping, Juni 1998.

[10]. W.Fornaciari, F.Salice, P.Micheli and L.Zampella. A First Step Towards Hw/Sw Partitioning of UML Specifications. IEEE/ACM Design Automation and Test in Europe (DATE'03), Munich, Germany, p.668-673 March, 2003.

[11]. V. Sinha, F. Doucet, C. Siska, R. K. Gupta, S. Liao, A. Ghosh. YAML: A Tool for Hardware Design Visualization and Capture. In Proc. International Symposium on System Synthesis, 2000

[12]. J.R. Beauvais, T. Gautier, P. Le Guernic, R. Houdebine, E. Rutten. A translation of Statecharts into Signal. Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98), IEEE Publ., pages 52-62, Aizu-Wakamatsu, Japan, March 1998

[13]. William E. McUmber and Betty H.C. Cheng. UML-Based Analysis of Embedded Systems Using a Mapping to VHDL, in Proc. of IEEE High Assurance Software Engineering, Washington, DC, November 1999

[14]. P. N. Green and M. D. Edwards. The Modelling of Embedded Systems Using HASoC. in Proceedings of DATE 2002 Conference, Paris, France, March 2002.

[15]. Bichler, L.,Radermacher, A. and Schurr, A. Evaluating UML extensions for modeling real-time systems.Published in Proceedings of the International Workshop on Object-Oriented Real-Time Dependable Systems. P.271-278, January, 2002

[16]. Razvan Jigorea, Sorin Manolache, Petru Eles and Zebo Peng. Modelling of Real-Time Embedded Systems in an Object-Oriented Design Environment with UML.

Proc. IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC 2000), Newport Beach, California, 2000

[17]. Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji. An Object-Oriented Design Process for System-on-Chip using UML. In Proc. of the 15th International Symposium on System Synthesis (ISSS 2002), Kyoto, Japan. p.249-254, 2002

[18]. Grant Martin. SystemC and the Future of Design Languages: Opportunities for Users and Research. 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03), September, 2003

[19]. CoCentric SystemC™ Compiler RTL User and Modeling Guide, Synophsis Inc. 2003

[20]. Jeffery H. Reed. Software Radio, a modern approach to radio engineering. Prentice Hall PTR. 2002

[21]. Born M., Holz M. and Kath M.. A Method for the Designand Development of Distributed Applications using UML. International Conference on Technology of of Object-Oriented Languages and Systems (TOOLS Pacific), SydneyAustralia, November, 2000

[22]. W. CESARIO, Z. SUGAR, R. SUESCUN, A.A. JERRAYA. Overlap and frontiers between behavioral and RTL Synthesis, User's Forum, DATE'99, Munich, Germany, March 1999.

[23]. Kathy Dang Nguyen, Zhenxin Sun, P.S. Thiagarajan, Weng-Fai Wong. Model-driven SoC Design Via Executable UML to SystemC. The 25th IEEE International Real-Time Systems Symposium(RTSS2004), Portugal, Dec, 2004