Refactoring-based Executable UML Transformation for Embedded System Design

Nurul Azma Zakaria, Noriko Matsumoto, Norihiko Yoshida

Graduate School of Science and Engineering, Saitama University, Saitama 338-8570, Japan

Summary
Typical stepwise refinement is widely used in design process, but due to increased design complexity a new way of designing is necessary. In this paper, we present a system-level design methodology using Executable-UML (xUML) and Model Driven Architecture (MDA) concepts. This proposed research aims at developing a language-independent framework for stepwise refinement by refactoring of xUML models. We investigate a transformation mechanism from an abstract specification model to a concrete implementation model in xUML representation exploiting the MDA capabilities in a step-by-step manner. Our modeling framework uses selected subsets of UML diagram types with action semantics. We show the application of our work by several design examples including GSM Vocoder design. As a result, we obtained executable models and a set of well-defined refactoring rules to accelarate design processes and improve product qualities of not only SoC but embedded systems in general.

Key words:
Executable-UML, Stepwise Refinement, System-Level Design, Model Driven Architecture, Refactoring

1. Introduction

The increasing design complexity coupled with requests for shorter development timeframe have produced high pressures in the design of System-on-Chip (SoC) or embedded systems in general. This leads to exploration of new ways or methods of designing SoC from various aspects. In order to cope with this issue, one accepted approach by the design community is to raise the level of abstraction of the design process to a higher level called system-level. At the system-level, there is no difference between hardware and software. Great productivity gains can be achieved by starting design from this level. In system-level design, there are several programming languages called system-level description languages, proposed to facilitate the design methodology such as SpecC [20], SystemC [22] and SystemVerilog [23]. The choice of the language used depends on various factors. However, all of these methodologies are very dependent on the language and platform it used and this will restrict the design flexibility. A new way is needed to describe an entire system. Thus, what we need is a language-independent framework to increase the design portability.

The same issue has already been addressed in Software Engineering area and Model Driven Architecture (MDA) [12, 16] is proposed. The application of MDA concepts are being actively researched in this area and have received a lot of attention in recent years from industry and academic communities. With MDA, an abstract specification model is defined in the form of Executable-UML (xUML) [9, 12, 16], and transformed into a concrete implementation model with specific platform information and constraints, and then is translated into a program code. We found that model transformation employed in MDA is equivalent to transformation of stepwise refinement in system-level design. Therefore, we believe that the MDA concepts can also be applied to system-level design. Hence in our research, we investigate and organize a design methodology of stepwise refinement in system-level design with xUML, where xUML is the basis of MDA concept. There are already some related studies on application of UML or MDA to system design. Brown [3] and Sunye et al. [21] discussed general MDA principles and practice. There are also researches applying MDA to SoC design or embedded system design such as Boulet et al. [2], De Jong [4], Mellor et al. [13], Riccobene et al. [17] and Schattkowsky and Muller [19]. In addition to that, the roles of UML or xUML in various areas of system-level design have been explored and investigated by Katayama [8], Martin and Muller [11], Muller et al. [14], Nguyen et al. [15], Riccobene et al. [18] and Tan et al. [24]. However, there has been none yet on xUML transformation based on refactoring for system-level design.

Thus, our research addresses the transformation of xUML models in a step-by-step manner from an abstract model to a more refined and concrete model. Our goal is to accomplish a step-by-step transformation in xUML from an abstract specification model of the system under design and eventually creates a more refined and concrete implementation model ready for manufacturing. We perform the xUML transformations by using the

Manuscript received June 5, 2009
Manuscript revised June 20, 2009

refactoring technique [5, 26] and generate a well-defined set of refactoring rules to guide design process. In this paper, we show how this proposed approach is carried out using some design examples. The construction of the work presents in this paper is part of our ongoing effort to develop a full design of language-independent framework for xUML-based stepwise refinement. Fig. 1 illustrates the flow of proposed approach in comparison with typical system-level design and MDA flow.

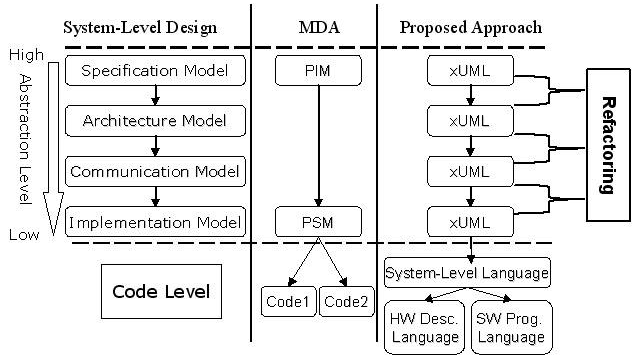

Fig. 1 Proposed design flow.

The outcomes of our proposed research will benefit the system-level design area in general where it will help to improve the existing system design methodology and increase design portability. It also produces a richer and more useful documentation of a well-defined stepwise refinement process. For system designers, this approach provides a simplified design process based on meticulous, clear, and structured models at each design phase which enables quick exploration and synthesis. This will reduce the amount of resources and the man power required to complete any design. Furthermore, with a clear and detailed set of refactoring rules which acts as a guidance in designing, the design process can easily be understood, thus low designer expertise is required.

The remainder of this paper is organized as follows. Section 2 and Section 3 provide a brief introduction to stepwise refinement and refactoring technique. Section 4 explains on MDA and xUML concepts in general. Section 5 is the key part of this paper, gives an overview of modeling in xUML, application of refactoring rules and transformation by examples. Then, Section 6 describes the real system design example. Finally, the last section draws main conclusion and introduces future work.

2. Stepwise Refinement

The stepwise refinement process is a design methodology for system-level design. We adopted the SpecC design methodology [6, 7] in the system-level design from several proposals, because it has been widely accepted and has been a fundamental part of SystemC and SystemVerilog. The SpecC system-level design methodology starts with the generation of the initial specification model that describes the functionality as well as other constraints of the intended design. This is the highest level of abstraction model which ignores any implementation details.

The design flow of the SpecC methodology consists of two main tasks namely the architecture exploration and the communication synthesis tasks. The architecture exploration tasks include design steps of allocation, partitioning of behaviours, channels and variables, and scheduling. This is an iterative process generating an architecture model which represents a refinement of the specification model. Then, it is followed by the communication synthesis task which refines the abstract communication between behaviours in the architecture model into an implementation. The tasks of communication synthesis includes the insertion of communication protocols, synthesis of interfaces and transducers, and inlining of protocols into synthesizable components. In the resulting communication model, communication is described in terms of actual wires and timing relationships are described by bus protocols. Next, the result of the synthesis flow is handed off to the backend tool for generation of final implementation model. At this level, the model represents a clock-cycle accurate description of the whole system. This description with a concrete level of abstraction serves as the basis for manufacturing of the system.

3. Refactoring

Refactoring [5, 26] is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour. It is a series of small behaviour-preserving transformations. Each transformation called refactoring does little, but a sequence of transformations can produce significant restructuring. Since each refactoring is small, it is less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. Refactoring is also structured as a practical method that improves maintainability, readability, reusability and modularity in later design phase by restructuring a system.

For the above-mentioned reasons, we applied refactoring technique to our transformation process. A set of refactoring rules is developed for each transformation to guide and support the preservation of the original behaviour of the system in the design process.

4. Model Driven Architecture and Executable-UML

Model Driven Architecture (MDA) introduces a transformation process from an abstract specification of "platform independent model" (PIM) to a corresponding "platform specific model" (PSM). PIM captures all information or requirements required for the system. It only defines the behaviors and functions of a system, which are independent from its implementation platform such as programming language, middleware and library. PSM comprises all the functionality expressed in the PIM with the added design concerns of the realizing platform such as translation into a specific program code implementation. A different platform yields a different PSM from a single PIM, for example, Java-based or C++-based.

Models in MDA are formal representation of the function, behavior and structure of the system. Each model is represented in the form of executable version of UML (Unified Modeling Language) [1, 25], i.e. Executable-UML (xUML) [12, 16]. It is a selected subset of the UML notation with the addition of fully defined execution semantics of UML which enable executable modeling. Some notations use in xUML acts as an informal notation such as use cases and activity diagrams. These diagrams are included in order to provide informal description and assist understanding and development of the system. The core and formal part of xUML comprises the class diagram and the state machine diagram. The former describes the static structure of the system, while the latter defines the dynamic behavior. Within the state machine diagram, the action semantics of the model is defined using action language. In this research, Action Specification Language (ASL) is adopted [9] so as to allow model execution.

5. Modeling in Executable-UML

Our modeling approach is based on xUML which mainly consists of class and state machine diagrams. Class diagrams are predominantly used to describe the static structure of a system while state machine diagrams depict the dynamic behaviour of a class. A set of refactoring rules is created to guide the modeling process and a complete model is validated and tested by simulation using iUML [9], a MDA compliance tool.

5.1 Refactoring Rules in Executable-UML

A set of refactoring rules is generated and as we continue working on this research, this list will be expanded in the future. Currently, the work completed is up to 3(a) task. The rest of the tasks will be left for future works.

1. Specification model
   a) Separation of concerns
      - Define channel class to represent communication
      - Define class to represent behaviour
      - Define data and relationship
   b) Identify level of hierarchy for related behaviours, data and communication
   c) Create state machine to explicitly model the semantic of the design
2. Architecture exploration
   a) Allocation and behaviour partitioning
      - Introduce additional level of hierarchy to existing class diagram
      - Identify class for processing elements (PEs)
      - Add relationship to behaviour classes
      - Group behaviour classes to related PEs classes
      - Add behaviour and channel classes to introduce synchronization
      - Move channel classes to represent global channels
   b) Variable partitioning
      - Move variables into related classes
      - Add communication channel classes
      - Update variable accesses
   c) Scheduling
      - Serialize behaviour classes' hierarchy
3. Communication synthesis
   a) Bus allocation and channel partitioning
      - Add another level of hierarchy to existing class diagram to model bus structure
      - Bind channel classes to buss classes and group communication
      - Renewal of access towards channel classes
   b) Protocol insertion
      - Insert protocol code
      - Generate application layer
      - Replace bus channels
   c) Transducer synthesis
      - Insert transducers
      - Encapsulate with wrappers
      - Synthesis transducer code
   d) Inlining
      - Inline communication methods
      - Optimize
4. Backend
   a) Hardware synthesis
   b) Software development
   c) Interface synthesis

5.2 Transformation Examples

In this section, we demonstrate some examples to show the representation of the models in SpecC which are translated and transformed into xUML models by applying the refactoring rules listed above.

Example 1: Representation of Specification Model

Fig. 2 depicts a simple yet typical example design of the specification model which consists of behaviour, channel and variable in xUML diagram. Each behaviour and channel is represented by a class and the relationship is depicted by association between the classes. Fig. 3 depicts the state machine diagram which relates to class organization shown in Fig. 2. This type of state machine diagram is attached to every class presented in the class diagram.

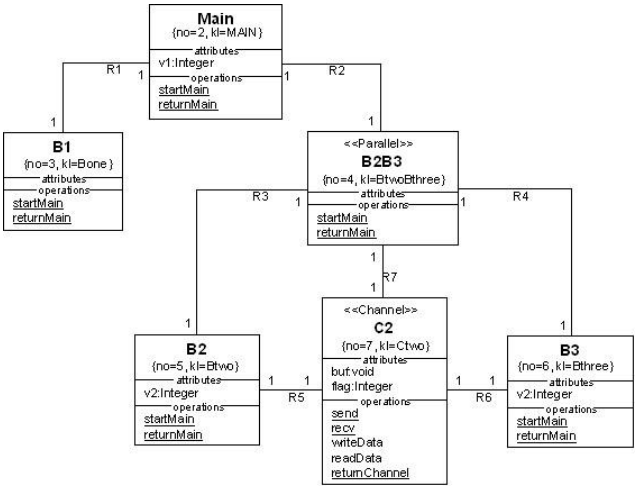

Fig. 2 xUML specification model.

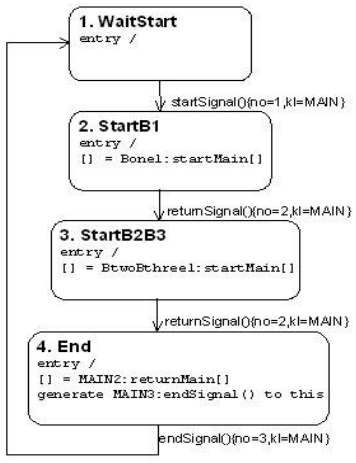

Fig. 3 State diagram of Main class.

In our research, generated models are simulated for design validation and we used iUML simulation suite [9, 16] for simulation process. The model execution is started by executing the initialisation sequence segment which creates all classes and associations described in class diagram. Then, the *startMain* operation of the *Main* class as shown in Fig. 2 is called in order to generate the first start signal to the first state of *Main* class in state machine diagram which is *WaitStart* as depicted in Fig. 3. The *startSignal* activates the *startMain* operation of desired class. The *startMain* operation consists of execution statement which then generates the following signal to *WaitStart* state of other relevant classes for further execution. Once the desired process is completed, *returnMain* operation is instantiated to generate *returnSignal* to the respective classes which at the end activates the *returnMain* operation of *Main* class and eventually *endSignal* is generated in order to stop the execution. The complete execution involves the combination of the different classes outlined in the class and state machine diagrams for each class. At the end of the simulation process, results such as output messages are displayed on the application window. Data tables and signal traces window can also be viewed for design conformation. The simulation process explained above is generally the same for every model including the execution of the following examples describe in later sections.

Example 2: Allocation and Behaviour Partitioning

The second example describes the xUML transformation involved in allocation and behaviour partitioning task which corresponds to transformation decribed in SpecC. This task determines the allocation of a set of system components in the specification model, partitions the behaviours onto the processing elements (PEs) according to functionality and also arranges variables and channels of the system architecture. Through this process, the system architecture model is developed from the specification model. Fig. 4 shows the xUML representation of the transformed model of this example.

In brief, this process determines the groups of computation in the specification model which defines the functionality to be implemented by each processing element (PE). Based on the analysis, we allocated two processing elements, *PE1* and *PE2*, and we mapped the main leaf behaviours *B1* and *B2* onto *PE1* while leaf behaviour *B3* is placed on *PE2*. After several intermediate refinements, other leaf behaviours such as *B3Stub*, *B13snd*, *B34rcv*, *B13rcv* and *B34snd* are mapped to *PE1* and *PE2* respectively. Variable *v1* and message-passing channels named *cb13*, *c2* and *cb34* become system-global variable and channels which both represent the communication between PEs.

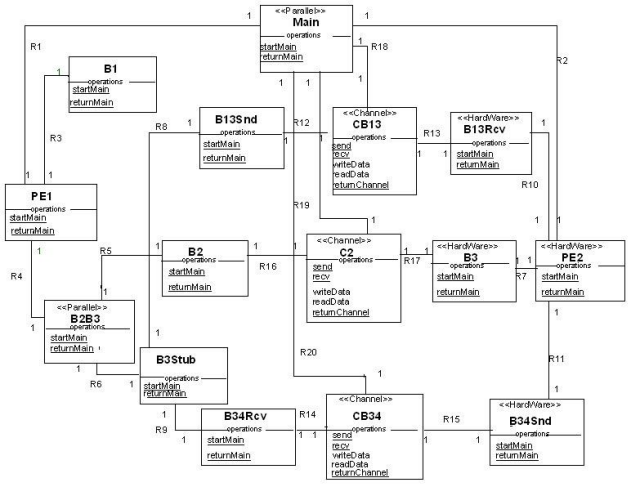

Fig. 4 xUML model after allocation and behavior partitioning.

In Fig. 4 which illustrates the xUML model, the components which represent by classes are divided into two main processing elements, *PE1* and *PE2*. *PE1* is the software component whilst *PE2* is the hardware component. In Fig. 4, the *Hardware* tag is added in some of the classes named *B13Rcv*, *B3*, *B34Snd* to differentiate them from the software classes which residing in the software component. The *Channel* tag is attached to class *CB13*, *C2* and *CB34* to depict the message-passing communication in the design. The *Main* class which represents the top-level hierarchy is attached with *Parallel* tag in order to show that *PE1* and *PE2* classes are running in parallel to preserve the original specification of the design.

Example 3: Channel Partitioning

Our third example is the main example which describes the step-by-step refinement involved in channel partitioning process. This work is a continuation from our previous examples. Channel partitioning refines abstract communication between components in architecture model towards an actual implementation of system busses. In this process, an abstract architecture model is gradually refined into more concrete model.

(1) Refactoring Rules of Channel Partitioning

The elaborate step-by-step refactoring rules involved in this process are:

1. Add another level of hierarchy to the existing class diagram to model its bus structure
   - Introduce an additional level of hierarchy by creating a new class to represent the bus structure
   - Add relationship to connect bus class to *Main* class
2. Bind channel classes to the bus class and group communication
   - Remove relationship of channel classes from *Main* class
   - Add relationship to connect channel classes to the bus class
3. Renew access towards channel classes
   - Remove the relationship of related behaviour classes from channel classes
   - Add relationship to connect related behaviour classes to bus class

The bus structure is represented by channel and employs all channel properties and it acts as top-level channel. Then, the abstract communication channels instantiated between the components are grouped under the bus channel according to the selected mapping. At this point, the bus channel *Bus1* does not have any connection with the other components such as *PE1* and *PE2*. Lastly, the communication inside the components is updated to merge all bus communication over the bus channel. Bus interfaces are created as the union of all sub channel interfaces grouped under the bus. A virtual bus addressing scheme is introduced for the sub channels at the bus interfaces. Then, the components are connected to the busses via bus ports and bus interfaces, and all channel accesses inside the PEs are replaced with bus access.

(2) Transformation of Architecture

The class organizations are specified in xUML as shown in Fig. 5, Fig. 6 and Fig. 7. Each function or behaviour is assigned to a class.

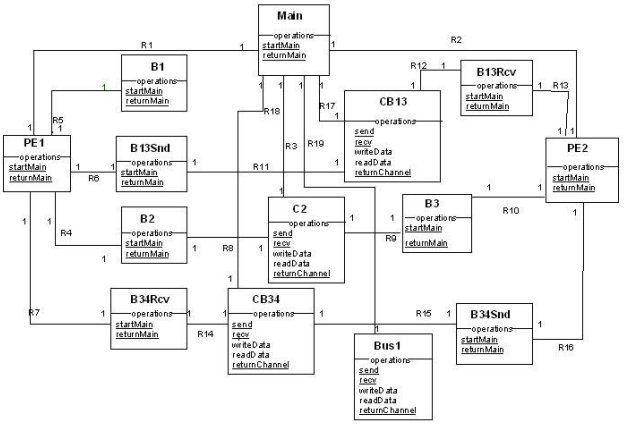

Fig. 5 xUML model before channel partitioning.

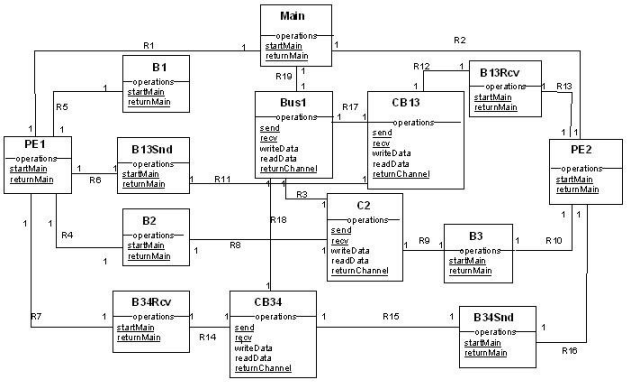

Fig. 6 Intermediate xUML model of channel partitioning.

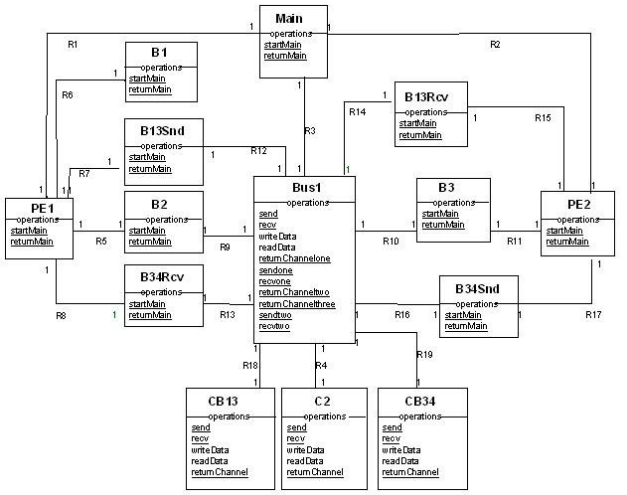

Fig. 7 Final xUML model after channel partitioning.

In the initial model, *Main* class represents the top-level hierarchy of the design which connects to subclasses such as *PE1*, *PE2* and *C2*. A new level is added with creation of a new class and its properties namely *Bus1*. This *Bus1* is linked to *Main* class to model the bus structure. No other relationship exists for this new class.

We performed an intermediate refactoring to the initial model which derived a more concrete model as shown in Fig. 6. It represents the layout towards channel bus and communication grouping. In this step, *Bus1* class is connected to other channel classes, named *CB13*, *C2* and *CB34*. Lastly, we carried out another refactoring to the previous model and finally generated a more refined model as defined in Fig. 7. This xUML model describes the renewal of access towards channels and communication update by linking the *Bus1* class to other related classes which are *B13Rcv*, *B13Snd*, *B2*, *B3*, *B34Rcv* and *B34Snd*. In this transformation, relationships which connect *Bus1* with other channel classes, *CB13*, *C2* and *CB34* remain. However, the relationships which connect channel classes *CB13*, *C2* and *CB34* with classes *B13Rcv*, *B13Snd*, *B2*, *B3*, *B34Rcv* and *B34Snd* are removed as the communication from these classes to channel classes is managed by *Bus1* class. In this step, the xUML model becomes more detailed due to change in connectivity among classes. However, what is important is that the functions and behaviours of the original specification model are strictly preserved, while its structure is modified so as to make the model more concrete.

(3) Transformation of State Transition

The transformation of state transition for channel partitioning is specified in xUML as shown in Fig. 8 which relates to the transformation described in previous section. This is represented in ASL. The state machine diagram of *B13Rcv* class is shown as an example. Each rounded rectangle represents a state, and codes in the rectangle represent action semantics at each state. Arrows represent state transitions. This sort of state transition definition is attached to every class presented in the class diagram. Fig. 8 (a) diagram denotes the initial state machine diagram which corresponds to the static structure of the design illustrated in Fig. 5. Initially, before undergone the transformation process the *startSignal* instantiates the second state which starts the execution of *startMain* operation of *CB13* class. However, after the transformation happened the *startSignal* activates *startMain* of *Bus1* class. This is due to change in connectivity of related classes as presented in Fig. 7. Finally, after all statements are executed the *returnSignal* starts *returnMain* operation of *B13Rcv* class to end the process in both cases.

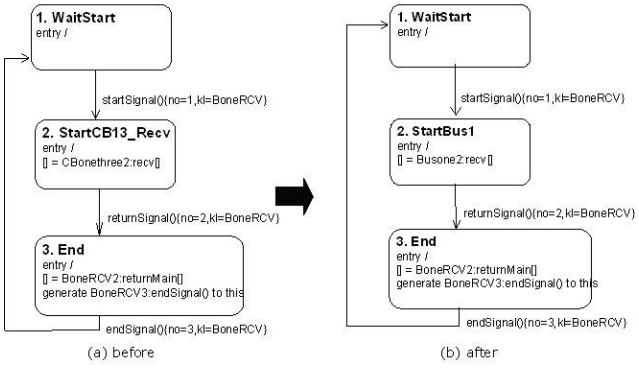

Fig. 8 State transition of B13Rcv class.

6. Application to GSM Vocoder Design

We illustrate a design example of a voice encoder/decoder (vocoder) for cellular applications known as GSM Vocoder [6, 7, 10] in order to show the applicability of our approach to real system application. GSM Vocoder is based on the standard for speech coding and compression in the European cellular telephone network system GSM (Global System for Mobile Communications). The Enhanced Full Rate (EFR) speech transcoding is standardized by the European Telecommunication Standards Institute (ETSI) as GSM 06.60 which is widely used in voice compression and encoding for speech transmission [6].

For the purpose of this research we focused on the encoder part of the vocoder specification. The model shown in Fig. 9 only depicts the top level of the hierarchy that we dealt with. For a complete hierarchy of the encoding part of the vocoder down to the leaf behaviours are discussed in [6, 7]. We applied our previous modeling design of allocation and behaviour partitioning task to this real system design example. For simplicity, we only explain on transformation of architecture involved in this design. It shows the vocoder specification model before undergoing a series of transformation of refinement process.

All of the classes in Fig. 9 represent components in the Vocoder system. Based on analysis, we selected the *Codebook* behavior to move into hardware component. This is shown by the *HW* tag with a circle in Fig. 9. In order to realize this, several intermediate refinements are performed by following the refactoring rules. The specification model is refined to a more concrete model as described in Fig. 10.

xUML model shown in Fig. 10 consists of additional classes and relationships. This is as a result of more detailed design of the system. Although the model undergone a number of transformation, the functions and behaviors of the original specification model are strictly preserved. In the final model, the initial *Codebook* class depicted in Fig. 9 is removed from its original relationship to connect with a new *Hw* class. Subclasses namely *Codebook_Start_Recv*, *Codebook* and *Codebook_Done_ Send* comprise of *HardWare* tags to represent subbehaviours residing in a hardware component which denotes by *Hw* class. Other classes without any tag are classes related to DSP component. In addition to that, *Codebook_CN* class is extended to have three subclasses namely *Codebook_Stub*, *Codebook_Start_Send*, and *Codebook_Done_Recv* to preserve the semantic of the original design.

Variables inside classes and message passing channels class which are *Ch_Codebook_Start* and *Ch_Codebook_Done* are marked with *Channel* tag become system global variables and channels to represent the communication between PEs. *Coder* class consists of *Parallel* tag to define the parallelism of DSP and *Hw* classes. The correctness of these models has been validated by simulation process using iUML [9]. The results of the simulation process showed that the designs behaved in desired manner following the rules specified for each transformation.

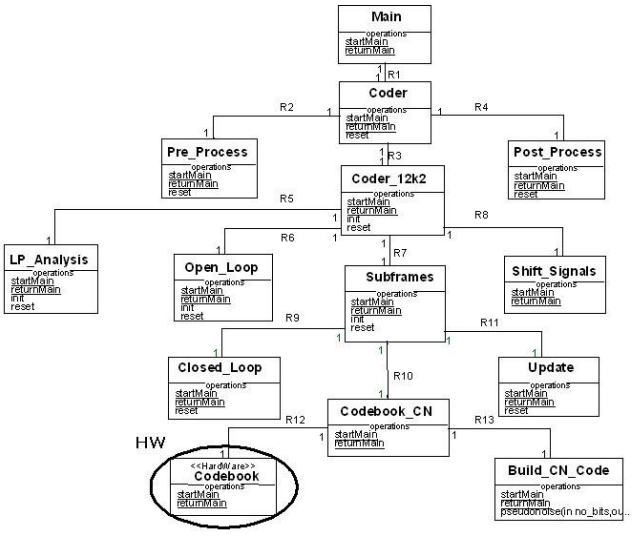

Fig. 9 Vocoder specification model.

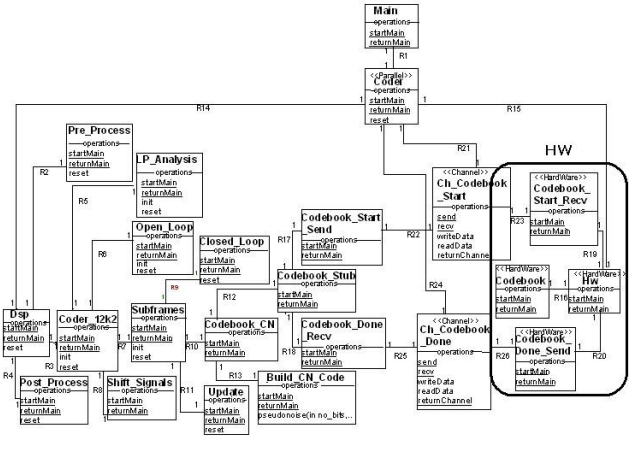

Fig. 10 Vocoder refined model.

7. Conclusion and future work

In this paper, we have introduced the modeling framework in which designs can be specified using xUML notations to support the specification and modeling of complex SoCs or embedded system in general. We outlined a transformation mechanism from an abstract specification model to a more concrete model by applying the refactoring technique through well-defined sequence steps. We also presented a set of refactoring rules which is used as a guideline in the design process. We have verified our approach by simulating the models using iUML suite, a MDA tool. In addition to that, we also verified the applicability of using xUML concepts in realizing the design process by several design examples including real system example of GSM Vocoder design.

As future works, we will extend our work by modeling the remaining tasks of stepwise refinement to enable generation of complete system-level design methodology using our framework, so that we can formalize and catalog a full set of guidelines and apply them to other real applications. We are still at the starting point in this research, and strategically developing a full language-independent modeling framework for xUML-based stepwise refinement and also a tool for design automation. Consequently, we suppose this empirical work will benefit to accelerate design processes and improve product qualities.

Acknowledgments

The authors are grateful to Mr. Masahiro Kimura (Toshiba Solutions Corp., Japan) for his valuable contribution.

References

[1] Booch, G., Rumbaugh, J. and Jacobson, I.: "The Unified Modeling Language User Guide", Addison-Wesley, 2005

[2] Boulet, P., Dekeyser, J., Dumoulin, C. and Marquet, P.: "MDA for SoC Design, Intensive Signal Processing Experiment", Forum on Specification and Design Languages (FDL'03), 2003

[3] Brown, A.W.: "Model driven architecture: Principles and Practice", Software and Systems Modeling, Springer, 2004

[4] De Jong, G.: "A UML-Based Design Methodology for Real-Time and Embedded Systems", Proc. of the Design, Automation and Test in Europe (DATE'02), 2002

[5] Fowler, M.: "Refactoring: Improving the Design of existing Code", Addison-Wesley, 1999

[6] Gajski, D.D., Zhu, J., Doemer, R., Gerstlauer, A., Zhao, S.: "SpecC: Specification Language and Methodology", Kluwer Academic Publishers, 2000

[7] Gerstlauer, A., Domer, R., Peng, J. and Gajski, D.D.: ``System Design: A Practical Guide with SpecC", Kluwer Academic Publishers, 2001

[8] Katayama, T.: "Extraction of Transformation Rules from UML Diagrams to SpecC", IEICE Trans. Information and System, Vol.E88-D, No.6, 2005

[9] iUML, Kennedy Carter Ltd: http://www.kc.com/

[10] Kimura, M., Kobayashi, K., Yamasaki, R. and Yoshida, N.: "Application of Refactoring-based Stepwise Refinement Design to GSM Vocoder", Embedded Systems Symposium, 2006 (in Japanese)

[11] Martin, G. and Muller, W.: "When Worlds Collide: Can UML Help SoC Design?", UML for SOC Design, Springer, 2005

[12] Mellor, S.J. and Balcer, M.J.: "Executable UML: A Foundation for Model-Driven Architecture", Addison-Wesley, 2002

[13] Mellor, S.J., Wolfe, J.R. and McCausland, C.: "Why Systems-on-Chip needs More UML like a Hole in the Head", UML for SOC Design, Springer, 2005

[14] Muller, W., Rosti, A., Bocchio, S., Riccobene, E., Scandurra, P., Dehaene, W. and Vanderperren, Y.: "UML for ESL Design: Basic Principles, Tools, and Applications", Proc. 2006 IEEE/ACM Int. Conf. on Computer-Aided Design, pp. 73-80, 2006

[15] Nguyen, K.D., Sun, Z., Thiagarajan, P.S. and Wong, W.F.: "Model-Driven SoC Design via Executable UML to SystemC", Proc. 25th IEEE International Real-Time Systems Symposium (RTSS'04), pp. 459-468, 2004

[16] Raistrick, C., Francis,P., Wright, J., Carter, C., Wilkie, I.: "Model Driven Architecture with Executable UML", Cambridge University Press, 2004

[17] Riccobene, E., Rosti, A. and Scandurra, P.: "Improving SoC Design Flow by means of MDA and UML Profiles", 3rd Workshop on Software Model Engineering (WiSME), 2004

[18] Riccobene, E., Scandurra, P., Rosti, A. and Bocchio, S.: "A SoC Design Methodology Involving a UML 2.0 Profile for SystemC", Proc. of the Design, Automation and Test in Europe (DATE'05), Vol. 2, pp. 704-709, 2005

[19] Schattkowsky, T. and Muller, W.: "Model-Based Design of Embedded Systems", 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), 2004

[20] SpecC Web Site: http://www.cecs.uci.edu/specc/

[21] Sunye, G., Pennaneach, F., Ho, W.M., LeGuennec, A. and Jezequel, J.: "Using UML Action Semantics

for Executable Modeling and Beyond", Lecture Notes in Computer Science (LNCS), Springer, 2005
[22] SystemC Web Site: http://www.systemc.org/
[23] SystemVerilog Web Site: http://www.systemverilog.org/
[24] Tan, W.H., Thiaggarajan, P.S., Wong, W.F., Zhu, Y. and Pialakkat, S.K.: "Synthesizable SystemC Code from UML Models", 41th Design Automation Conference, (workshop UML for SoC Design), 2004
[25] Unified Modeling Language (UML) Web Site: http://www.omg.org/
[26] Yamasaki, R., Kobayashi, K., Zakaria, N.A., Narazaki, S. and Yoshida, N.: "Refactoring-Based Stepwise Refinement in Abstract System-Level Design", Proc. IFIP 2006 Int. Conf. on Embedded and Ubiquitous Computing, (LNCS, No. 4096, Springer), pp. 712-721, 2006