ReUML: a UML Profile for Modeling and Verification of Reactive Systems

S. Fatemeh Alavizaedh
University of Tehran, ECE Department
North Karegar av. Tehran, Iran
f.alavi@ece.ut.ac.ir

Alireza Hashemi Nekoo
University of Tehran, ECE Department
North Karegar,Tehran, Iran
a.hashemin@ece.ut.ac.ir

Marjan sirjani
School of Computer Science, IPM and University of Tehran
msirjani@ipm.ir

Abstract

The Unified Modeling Language, has become effectively the standard modeling language for analysis and design of software systems. However, despite achievements in defining semi-formal semantics, with a combination of OCL constraints and textual descriptions of the UML semantics, UML is still an informal language. This paper introduces a tool for developing correct models of distributed and reactive systems using UML and Rebeca. Rebeca is an actor-based modeling language supported by a formal verification tool. This approach can bridge the gap between software development and formal verification by allowing users to develop their systems using UML and yet getting advantage of formal verification support of Rebeca tools and theory. In this way, we combine two separate approaches to modeling by adding verification step to software development lifecycle. Furthermore, this can make a contribution to defining rigorous semantics for UML diagrams and to provide tool support for verification of these diagrams.

1. Introduction

The Unified Modeling Language, UML, while not officially announced, has become the standard notation for modeling and documenting software systems and is under constant evolution by OMG group [24]. The idea of one standard language for modeling provides many advantages to software development, such as simplified training and unified communication between development teams. Furthermore, the UML inspired a new approach to design: Model Driven Architecture (MDA) [24, 10]. One of the most important aspects of MDA is code generation which is to automatically generate as much as possible code from the input model, leaving only little for the error-prone manual coding [5].

However, there are some drawbacks in UML. First, UML is complex, thus tools can support only a part of it [16], and while it was supposed that training will become easier with a unified language, the actual results are not as desired. Second, despite achievements in defining semi-formal semantics with OCL constraints, and even with various studies toward formalizing semantics [9, 19, 6, 12] of UML, the language still lacks formal semantics. This can damage the former perspective of a unified modeling language for all software systems, since many systems are too critical to be left unverified. In many cases, developers need to model their systems with a formal language to make the verification in early stages of development possible. One of the advantages of UML, which can help to surmount its drawbacks, is the possibility of customization [7]. Because of large amount of concepts and complexity of UML, usually the usage of its concepts is restricted by defining UML profiles. A profile is a subset of the syntax of UML plus a number of well-formedness rules. These rules add standard elements to the subset and specify additional semantics in natural language. In other words, UML profile is a subset of UML concepts which is adequate to define our domain.

In our previous studies [1] a new profile for modeling concurrent reactive systems has been proposed. In this paper, we extended this profile to model reactive components of large systems, which supports interaction with other subsystems. Moreover, we have developed a tool based on our profile for designing concurrent and distributed systems consisting of asynchronously communicating reactive objects in UML. This tool fully supports our profile and confines UML concepts to the one we have used in this profile. Using this tool we can convert our models to Rebeca, an actor-based language for modeling and verification of reactive systems. The Rebeca to Java converter can help us to generate Java code from our models. Consequently, there would be a path from UML model to executable code that supports verification via Rebeca. With this approach developers don't need to be involved in all complexities of UML diagrams, and they also don't need to be concerned about intricacies of any formal language since the complete code can be generated from their models.

2. Related Work

There have been many UML profiles proposed to OMG [24], and some of these profiles have became standard[25]. CORBA allows applications to communicate with each other regardless of the location and design, the UML profile for CORBA which is a standard profile provides means for expressing the semantics of CORBA IDL (Interface Description Language) using UML artifacts, which enable expressing these artifacts with UML tools. UML Profile for Enterprise Application Integration (EAI) [25] provides a metadata interchange standard for information about accessing application interfaces to simplify application integration. There is a UML profile for relational database [20]. Key, secondary key, table, and file are examples of stereotypes proposed in this profile. Omega is a UML profile for embedded and real-time systems based on formal techniques [23]. TURTLE [4] is another UML profile dedicated to the modeling and formal validation of real-time systems. TURTLE defines a set of operators and diagrams addressing the need for analyzing, designing and deploying temporally constrained systems. Formal semantics of TURTLE relies on RT-LOTOS which is a formal description technique based on process algebra. TTool [4] is a free toolkit for editing and validating TURTLE / UML diagrams.

In addition, many works have been done on formal semantics of UML. There are two common approaches in this field: UML formalization, and translation to formal languages. In the first approach, in order to formalize the UML, mathematical theories are used [16]. In [9] authors worked on algebraic specifications of OMT object model diagrams. Lano and Bicarregui in [13] have specified a possible semantic for a large part of the UML using structured theories in a simple temporal logic. The semantic model of UML used in their work is based on the set-theoretic Z-based model of syntropy. In the second approach, formal semantics can be given to UML by introducing a mapping from UML diagrams to an established formal method or language. Examples of this approach are the mapping of UML into Object-Z which is an object oriented formal specification language for modeling computing systems [16], and B, a formal method that is based around abstract machine notation [8].

Furthermore, in the area of modeling reactive systems a lot of researches have been done. In [17], in order to bridge the gap between system model and program code, state charts and SWITCH-technology are used to develop reactive object oriented programs. In [18], a way for graphical design of reactive systems using state charts, is introduced which translates state charts into the Abstract Machine Notation (AMN) of the B method. In [10], a UML profile for development of distributed reactive systems is introduced which is also based on state charts.

3. Rebeca Modeling Language

Rebeca [2], Reactive Objects Language, is an actor-based language for modeling and verifying concurrent and distributed systems. Rebeca is designed in an effort to bridge the gap between formal verification approaches and real applications. It is also a platform for developing object-based concurrent systems in practice. The key features of Rebeca are: using actor-based concepts for the specification of reactive systems and their communications, providing a formal semantics for the model, providing a tool for model checking Rebeca code and using abstraction techniques to reduce the state space in model checking.

Rebeca is supported by Rebeca Verifier tool, as a front-end, to translate codes into existing model-checker languages to verify their properties [3]. In addition, Rebeca direct model checker is now under development [15]. For a formal verification method, more than a model, there should be a specification language to embody correctness requirements. Here, temporal logic is used to specify safety and progress properties which are based on state variables of each rebec (reactive object) in the model [2].

Rebeca model is similar to the actor model in that it has independent active objects, and asynchronous message passing. These objects are reactive and self-contained. We call each of them a rebec, for reactive object. Computation takes place by message passing and execution of the corresponding methods of messages. Each message specifies a unique method to be executed when the message is serviced. Each rebec has a buffer, called a queue (or inbox), for arriving messages. This queue has a queue length that has been defined in the reactive class definition, and it is the number of messages that the rebec can buffer. When a message at the head of a queue of a rebec is serviced, its method is invoked and the message is deleted from the queue. Each rebec is instantiated from a reactive class, *'reactiveclass'*, and has a single thread of execution. We define a model, representing a set of rebecs, as a closed system. A Rebeca code is consisted of definition of reactive classes and a *'main'* part. In the main part rebecs are instantiated from reactive classes. Each reactive class consists of known objects, state variables and a set of message servers. Known objects of a rebec, *'knownrebecs'*, are those rebecs that this rebec can send them messages. All message servers, *'msgsrv'*, of a reactive class can use its state variables, but they are not public to be used by other reactive classes. The set of state variables, identified by keyword *'statevars'*, are variables which represent the state of the rebec. It is required that every reactive class definition has at least one message server named *'initial'* which is put in the queue when a rebec is instantiated. In declaring a rebec, the bindings to its known rebecs is specified in its parameter list.

4. ReUML: The UML Profile for Modeling Reactive Systems

Since UML is customizable, we can introduce ReUML (Reactive UML) as a new profile for modeling concurrent and distributed systems. In addition, as Rebeca is an object-based language the mapping from UML diagrams to Rebeca models is natural. In the following we will discuss this mapping briefly; details of the profile and mappings are discussed in [1].

Like all oriented approach, to model the static design view of a system, it is necessary to sketch a class diagram. A class diagram shows a set of classes and collaborations and their relationships [7]. At first we should define the building blocks which are classes, and subsequently, we should recognize and model their behavior and their collaborations.

In Rebeca, rebecs are the only entities constructing our systems, and reactive classes are used to make a template for the behavior of these rebecs. Naturally, objects are mapped to rebecs, and classes are mapped to reactive classes. So a class with *'reactiveclass'* stereotype, is a template for all reactive objects in our model. Each reactive object has some variables defining its state, in Rebeca vocabulary these are known as state variables. In other words, attributes of a reactive object are state variables in Rebeca. Recognizing methods of reactive object, called msgsrv in Rebeca, is the next step toward systems modeling. Therefore, each attribute with *'statevar'* stereotype models a state variable, and each method with *'msgsrv'* stereotype reflects a message server in our models. Additionally, each reactive class works in collaboration with others. In our reactive and concurrent environment objects are communicating with message passing. We can represent message sending with an association between reactive classes. The mapping between Rebeca and UML to show the static structure of a reactive system by a class diagram is shown in Table 1.

Given that all software systems consist of objects rather than classes, in executable models, we need a real snapshot of the system that shows a set of objects and their relationship. This is the responsibility of an Object diagram in a model. So, we need to show the object diagram of our reactive system as well.

After describing the static structure, we should define dynamic behavior of our system. State diagrams are common tools in modeling reactive systems. However, state machines are used to show the behavior of objects which have a lifecycle that is of interest to be modeled. Nevertheless, in an abstract view, each rebec starts in *'idle'* state and after receiving a message goes to *'waiting'* state waiting for its turn, and then goes to *'running'* state. We can break running state to sub-states to show the execution of each method, but Larman, in [11], encourages using state chart diagrams only for the purpose of illustrating external and temporal events and the reaction of an object to them, in Rebeca, dequeueing a message and executing an atomic message server cannot be considered as an external event. Moreover, we want to perform formal verification in early stages of modeling, so we prefer to use sequence diagrams as they can be extracted in a straight forward way from the domain. Therefore we chose sequence diagrams. This is discussed in more details in [1].

Table 1. Mapping between Rebeca and UML Constructs

UML Element	*Rebeca Element*	*Stereotype*
Class	Reactive Class	reactiveclass
Attribute	State Variable	statevar
Method	Message Server	msgsrv
Association End	Known Object	-

In our model, similar to what we usually do in software engineering, we draw a sequence diagram for each message server, and we show the sequence of actions invoked by receiving a message. In each sequence diagram we have one main rebec and its known objects (because they are the only possible message receivers). The sequence of actions initiates by receiving a message from the sender, the rest of the diagram shows what would happen if the receiver rebec gets such a message. In the following case study this explanations are illustrated.

5. ReUML designer: the Modeling Tool

One of the most difficult phases in software development is testing. One reason is that typically testing happens after implementation; If we could get confident that the system works correct prior to implementation the process would be considerably easier. To do so, we need to verify our models. However, complexities of formal modeling languages and verification process usually limit their usage to critical systems. As the analysis and design stage of the development process mostly uses UML, using it to perform verification as well, can reduce costs and time of software development lifecycle significantly. ReUML helps modelers to use the defined profile which confines UML concepts to needed ones. It lets UML modelers to verify their models without being concerned about details of Rebeca, and it al-

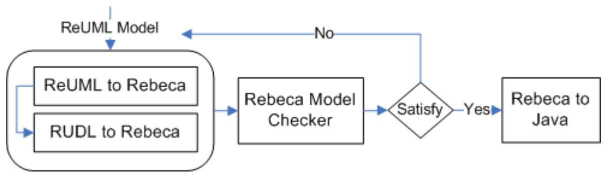


Figure 1. Development Process

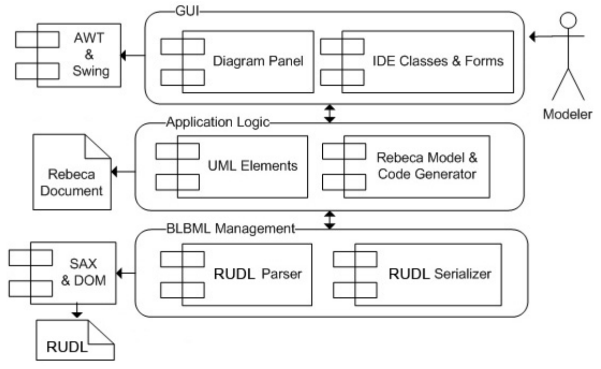


Figure 2. Tool Architecture

lows Rebeca developers to model their systems with UML and use automatic code generation. As we can see in Figure 1, which shows the development process under ReUML, the software will store a ReUML model in a RUDL document (ReUml Description Language). RUDL is an XML based language to store ReUML models; then RUDL to Rebeca component of ReUML designer parses RUDL and generates Rebeca model. We could have used XMI instead of RUDL, however not all aspects of our models, such as queue length, can be expressed in XMI, in addition XMI supports large amount of concepts which are not needed; Thus using XMI may lead to inefficiency. After model checking, Rebeca to Java translator can make executable program.

The component diagram of the ReUML tool is shown in Figure 2. The three-tier architecture is developed using Java 1.5 to be portable and self dependent. In top level there is a graphical user interface developed with Swing and AWT packages. The Logic layer contains objects of the model and the code generator; this layer has interaction with GUI as well as RUDL manager. In RUDL manager, SAX and DOM packages are used to restore and retrieve models from RUDL documents.

ReUML users have four separate views to construct their models; each view is associated with one diagram: class diagram, object diagram, sequence diagrams, and use case diagram. As discussed earlier in section 4, class diagram, object diagram, and sequence diagram are needed to generate Rebeca code. Use case diagram may help us with the generation of executable program. Each of these views will be discussed in brief. Beside diagram construction parts, there is a controller for RUDL document management. Each RUDL document has at most four types of diagrams two of which are obligatory in defining a model: class diagram and object diagram. The software can generate Rebeca code any time during modeling process. In case of errors user will be informed with appropriate message.

Use case diagram: Usually software systems have interaction with other systems, layers, or modules, and we should define these interaction points in our model. Among UML diagrams, component diagram and use case diagram may be used for interaction management. Component diagrams are used in modeling the physical aspects of organization and dependencies among a set of components[7]. Use case diagrams are central to modeling the behavior of a system, a subsystem, or a class. They make systems and subsystems understandable by presenting an outside view of how those elements may be used in context[7]. We have used use case diagram to show functionality, since component diagrams are not suitable for design stage. It is not necessary to draw a use case diagram if we just want to verify our models. However, some of our rebecs may receive messages from sources other than rebecs in our model. For instance, a rebec may receive messages from sensors. correspondingly, it may want to send data to external systems such as controllers. The tool supports this by allowing to declare a message server as *'Out callable'*. A use case shows the message server and an actor shows the corresponding caller. Although this has no effect in generated Rebeca code (we just show it with a comment) in related executable code these methods are public while others are protected.

Class diagram: A class diagram shows a set of classes, interfaces, and collaborations. It shows the static design view of a system [7]. Similarly in our profile, class diagrams show reactive classes, their state variables, message servers, queue length, and relationships with other reactive classes.

Object diagram: An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the elements found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases [7]. In our profile this diagram shows rebecs and their bindings to other rebecs. ReUML designer generates Rebeca code of the "main" part from this diagram; A queue length of a class defined in class diagram can be overwritten in each rebec of object diagram. This can make significant optimization in verification process. Although this feature is not supported in Rebeca verifier at the moment, it can be added to the tool in future.

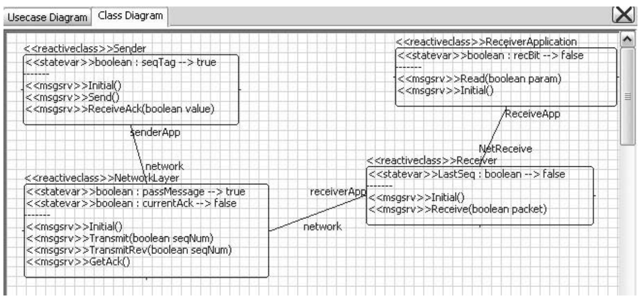
Sequence diagram: Sequence diagrams address the dynamic view of a system. They show an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them [7].

Each sequence diagram shows the dynamic behavior of a message server, consequently we should draw one for each message server in order to generate complete code of our systems. If-then-else structures, assignments, and asynchronous messages with their parameters can be shown in these diagrams. There is a feature to show synchronous messages between other systems and our model called *'x-call'*. Any out-callable massage server in our model can be called from outside. Also, x-calls can be made from our rebecs to other systems, for example calling a web service within a message server and getting back an answer. Currently Rebeca cannot support this feature so the program replaces these calls in Rebeca code with nondeterministic assignment from a range of values specified in sequence diagram for the return value.

6. Case study

Case study We modeled and verified Alternating Bit Protocol, ABP, with our tool.

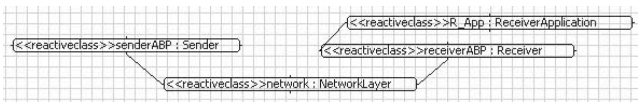
Problem definition: This Protocol is a simple yet effective protocol for managing the retransmission of lost messages. The system consists of a sender and a receiver. They are connected through a channel. Each data message sent by the sender contains one bit tag, 0 or 1, as well as message body and checksum value. The sender sends the message repeatedly until it receives acknowledge from receiver that contains the same tag value as the sent message. With the appropriate acknowledgment, the sender stops sending current message and will send the next message with alternative tag. Whenever the receiver gets a new message, it tests the checksum value. In the case of a correct message, it will deliver the message to the application and will send back an acknowledgment with the same tag value. The receiver will ignore incorrect messages, and subsequent messages with same tag value will be simply acknowledged. We must consider that the channel is not reliable and both messages and acknowledgments are imposed to loss and corruption [14].



Usecase Diagram
Class Diagram
<<reactiveclass>>Sender
<<statevar>>boolean : seqTag --> true
<<msgsrv>>Initial()
<<msgsrv>>Send()
<<msgsrv>>ReceiveAck(boolean value)
senderApp
network
<<reactiveclass>>NetworkLayer
<<statevar>>boolean : passMessage --> true
<<statevar>>boolean : currentAck --> false
<<msgsrv>>Initial()
<<msgsrv>>Transmit(boolean seqNum)
<<msgsrv>>TransmitRev(boolean seqNum)
<<msgsrv>>GetAck()
<<reactiveclass>>ReceiverApplication
<<statevar>>boolean : recBit --> false
<<msgsrv>>Read(boolean param)
<<msgsrv>>Initial()
ReceiveApp
NetReceive
<<reactiveclass>>Receiver
<<statevar>>LastSeq : boolean --> false
<<msgsrv>>Initial()
<<msgsrv>>Receive(boolean packet)
receiverApp
network

Figure 3. Class diagram

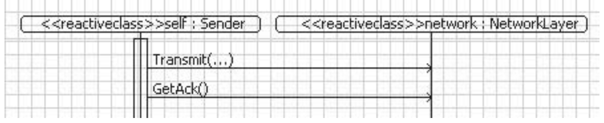
Figure 3 and 4 show the class diagram and object diagram of our model respectively. These diagrams show the



<<reactiveclass>>R_App : ReceiverApplication
<<reactiveclass>>senderABP : Sender
<<reactiveclass>>receiverABP : Receiver
<<reactiveclass>>network : NetworkLayer

Figure 4. Object diagram

static structure of our design. The model has a *Sender*, a *Receiver*, a *NetworkLayer*, and a *receiver application* reactive classes, with one instance of each. In the model we have two abstraction. First, sender application is abstracted; we assumed that sender application has enough data to send. Second, we omitted the message body and checksum field. Because they have no effect on modeling. We can assume corrupted messages as lost messages that won't be acknowledged. For modeling the dynamic behavior we should use sequence diagrams which are represented in figures 5 to 8.



<<reactiveclass>>self : Sender
<<reactiveclass>>network : NetworkLayer
Transmit(...)
GetAck()

Figure 5. Sender.Send

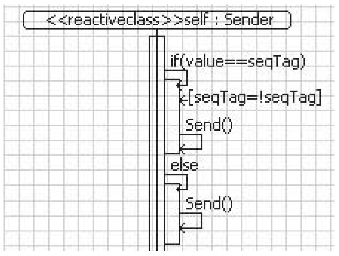
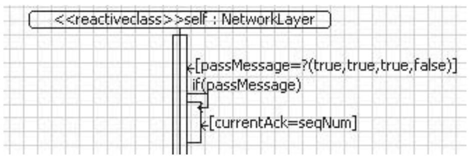


Figure 6. Sender.ReceiveAck

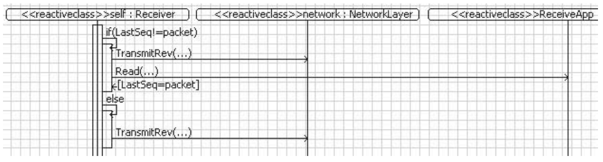
Figure 5 shows the *send* message server. Each time this message server is called the sender transfers the current bit to network layer and asks for the last acknowledgment. Sender's initial message server is put in the queue when a rebec is instantiated, thus in order to start protocol we should put a send call in initial message server. Whenever the sender receives an acknowledgment it checks its value with current bit. Same value determines successful transmission, thus the bit will be alternated. This is shown in figure 6.

Figure 7 shows *TransmitRev* message server which can be invoked by receiver to transmit an acknowledgment. Network randomly drops some of the acknowledgments, and instead of sending the rest of acknowledgments to sender it will keep the last one in *currentAck* state variable and will pass it to sender whenever sender asks for it. Thus Network layer's *GetAck* message server would be a message from network layer to sender. The reason for this behavior is to prevent sender queue from overflowing. Similarly, in



<<reactiveclass>>self : NetworkLayer
[passMessage=?(true,true,true,false)]
if(passMessage)
[currentAck=seqNum]

Figure 7. NetworkLayer.TransmitReverse



<<reactiveclass>>self : Receiver
<<reactiveclass>>network : NetworkLayer
<<reactiveclass>>ReceiveApp
if(LastSeq!=packet)
TransmitRev(...)
Read(...)
[LastSeq=packet]
else
TransmitRev(...)

Figure 8. Receiver.Receive

Transmit message server, network layer drops some of the messages from sender to receiver and transmits the rest.

Whenever the receiver gets a new bit it checks the value with last received bit, if the value is different it will pass it to the application and sends an acknowledgment back to sender, otherwise it will simply acknowledge the bit. This is shown in figure 8. Receiver Application gets values from receiver in Read message server.

The complete ReUML model and Generated Rebeca code of this example is located at [22]. The model has been translated to Promela by Rebeca to Promela translator.Model checking results show that the model is deadlock free. Also, the Alternating Bit Protocol implies that after each 1 the receiver will receive a 0 and vice versa. The following LTL properties has been checked with spin model checker, and both are valid.

1. [] (recBit == 1) $->$ <> (recBit == 0)
2. [] (recBit == 1) $->$ <> (recBit == 0)

7. Conclusions and future works

In this paper we have introduced a new tool for modeling and verification of distributed systems consisting of reactive objects which are communicating via asynchronous messages. The tool supports all features of our previous studies and introduces new features for development process. As it is suggested in figure 1, by development of ReUML the path from UML model to Java code with verification support is now complete. As a part of future work, we can consider development of an integrated environment for ReUML, Rebeca verifier, and Rebeca to Java to enhance modeling, verification, and executable code generation. Supporting synchronous messages using standard Rebeca, development of reengineering tool to generate UML models from current Rebeca codes, and support for using any compatible XML documents generated by other tools as an input model are other practical works to be done in the future.

References

[1] F. Alavizadeh and M. Sirjani. Using uml to develop verifiable reactive systems. *Software Engineering Research and Practice*, pages 554–561, 2006.

[2] M. Sirjani, A. Movaghar, A. Shali and F. De Boer. Modeling and verification of reactive systems using rebeca. *Fundam. Inform*, 63(4):385–410, 2004.

[3] M. Sirjani, A. Shali, M. Jaghouri, H. Iravanchi and A. Movaghar. A front-end tool for automated abstraction and modular verification of actor-based models. *ACSD*, pages 145–150, 2004.

[4] L. Apvrille, J. Courtiat, C. Lohr and P. de Saqui-Sannes. Urtle: A real-time uml profile supported by a formal validation toolkit. *IEEE Trans*, 30(7):473–487, 2004.

[5] M. Marinschek. Towards executable uml - code generation from interaction and state chart diagrams. Master's thesis, Technischen Universitat Wien, 2003.

[6] I. Ober. An asm semantics of uml derived from the metamodel and incorporating actions. *Abstract State Machines*, pages 356–371, 2003.

[7] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[8] H. Ledang and J. Souquires. Hung ledang, jeanine souquires. *IFM*, pages 109–127, 2002.

[9] J. Smith, M. Kokar and K. Baclawski. Formal verification of uml diagrams: A first step towards code generation. *pUML*, pages 224–240, 2001.

[10] M. Kersten, J. Matthes, C. Manga, S. Zipser, H. Keller. Customizing UML for the development of distributed reactive systems and code generation to ada 95. *Ada User Journal*, 23(6), 1999.

[11] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design Prentice Hall* , 1998.

[12] A. Evans, R. France, K. Lano and B. Rumpe. The uml as a formal modeling notation. *UML*, pages 336–348, 1998.

[13] K. Lano and J. Bicarregui. Formalising the uml in structured temporal theories. *OOPSLA*, 1998.

[14] G. Kamsteeg. A formal verification of the alternating bit protocol in ucrl.

[15] M. Jaghoori, A. Movaghar and M. Sirjani. The model checking engine of rebeca. *ACM SAC*, 2006.

[16] I. Porres. *Modeling and Analyzing Software Behavior in UML*. PhD thesis, bo Akademi University, 2001.

[17] V. Gurov, M. Mazin, A. Narvsky and A. Shalyto. Unimod: Method and tool for development of reactive object-oriented programs with explicit states emphasis. *St. Petersburg IEEE Chapters*, 2:106–110, 2005.

[18] E. Sekerinski. Graphical Design of Reactive Systems. *2nd International B Conference*, Springer-Verlag, 1998.

[19] W. McUmber and B. Cheng. A general framework for formalizing uml with formal languages. *ICSE*, 433–442, 2001.

[20] D. Gornik. Uml data modeling profile.

[21] http://khorshid.ut.ac.ir/∼rebeca.

[22] http://khorshid.ece.ut.ac.ir/∼rebeca/ReUML.htm.

[23] http://www-omega.imag.fr/index.php.

[24] http://www.omg.org/.

[25] http://www.omg.org/technology/documents/profile_catalog.htm.