Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 243 (2009) 69–87
www.elsevier.com/locate/entcs

BAS: A Case Study for Modeling and Verification in Trustable Model Driven Development

Dehui Du, Jing Liu[1,4]

Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai, P.R.China

Honghua Cao[2]

State Key Lab of Software Engineering, Wuhan University ,Wuhan, P.R. China

Miaomiao Zhang[3]

School of Software Engineering, Tongji University, Shanghai, P.R.China

Abstract

Multi-view modeling and separation of concerns are widely used to decrease the design complexity of the large-scale software system. To ensure the correctness and consistency of multi-view requirement models, the formal verification technology should be applied to the model-driven development process. However, there still lacks unified theory foundation and tool supports for the rigorous modeling approach. To solve these problems, we implemented an integrated modeling and verification environment tMDA (Trustable MDA) based on the theory of UTP. In tMDA, developers model system requirements with UML static and dynamic models and verify the correctness and consistency of different models. A multidimensional model is proposed, which supports the consistency verification, liveness and safety property verification, OCL constraints and LTL formula verification. A Bank ATM System (BAS) is introduced to demonstrate how to utilize tMDA for design and verification.

Keywords: MDA, Verification, Contract, UTP, LTL

1 Introduction

The complexity of software is increasing steadily and the correctness of software is more and more important. How to develop complex applications and guarantee the

[1] Email: dhdu@sei.ecnu.edu.cn, jliu@sei.ecnu.edu.cn
[2] Email: caohonghua@hotmail.com
[3] Email: miaomiao@mail.tongji.edu.cn
[4] Corresponding author

doi:10.1016/j.entcs.2009.07.006

correctness of software arouse researchers' interest increasingly. The component-based model-driven development is a promising approach for dealing with software complexities, which helps to ease the problems in identification, modeling and design of the different views. It is essential that the approach adopts multi-view UML modeling and allows separation of concerns [1]. Different concerns are described in different viewpoints of a system at different levels of abstraction, including interfaces, functional services, synchronization behavior, interaction protocols, resources and timing constraints. However, there are no rigorous theory foundations and integrated tools which support specification, verification of the different models produced in the development process [2].

Design by Contract (DbC) is a software correctness methodology [3]. It uses preconditions and postconditions to document (or programmatically assert) the change in state caused by a piece of a grogram. The idea of DbC has been accepted by the industry community, for example, the Eiffel language has made greate success [4]. Contract can be used to increase the reliability of software, but, how to model the contract with UML is still a challenge. As we all known, errors introduced early in the development process are known to have significantly higher correction costs [5]. In the development process constructing the high quality software models can avoid propagating design errors to implement stage. So, the model checker should be integrated into the model-driven development environment, which used to check the consistency of different models and the constraints in system requirements.

In the past half a century, semantic foundations, formal technologies and verification tools have been developed, including testing, static analysis, model checking, formal proof and theorem proving, etc. However, these formal verification technologies are not easy to be applied to the practical development process. There lacks an integrated modeling and verification environment for developers. Recently, we explored how to apply the model checking technology to the model-driven development process. An integrated modeling and verification environment is implemented to model systems with multi-view models and to verify the correctness and consistency of system models. The new features of our work are:

- An integrated modeling and verification environment is implemented, which is based on the approach of MDA-based trustable software development. We call it tMDA (Trustable MDA).
- tMDA supports the use case-driven requirement analysis and modeling approach effectively. Developers can model the static structure and dynamic behavior of a system. The consistency between different models can be verified on the model level, which ensures the correctness of PIM models. Moreover, tMDA supports model simulation, which helps developers analyze models and locate errors. Finally, the trustable models can be transformed to C++ code framework.
- Contract is modeled with OCL expressions on the model level so that the pre/post condition constraints of some methods can be expressed. Especially, the OCL expressions as constraints can be added into the transitions of a statechart. Thanks to the assertion verification of SPIN, OCL constraints can be verified. Besides,

tMDA can automatically transform OCL expressions to pre/post condition comments in the generated code framework, which facilitates the application of DbC in the mode-driven development process.

The rest of the paper is organized as follows. Section 2 gives an overview on the framework of tMDA, and provides the formulation of some key concepts for the consistency verification. In section 3, we use a BAS case study to demonstrate the procedures of model constructing, model verification, model simulation and code generation in the model-driven development process and how to apply the rigorous modeling approach to the practical software development. Section 4 summarizes our work and discusses some future work.

2 tMDA: Trustable Model-driven Development Environment

The main idea of our approach is to provide an integrated modeling and verification environment for developers so that they can construct the system requirement models and verify the correctness and consistency of these multi-view models. The approach helps to find early design errors and guarantee the quality of models, which makes the component-based model-driven development for trustable software possible. The rigorous unified theory foundation for tMDA is based on Hoare and Jifeng He's UTP [6], which provides semantics for unifying models of different views and supports the consistency verification. Based on the RUP development process [7], we propose the framework of tMDA, which facilitates the whole software development process from requirements elicitation, models constructing, component-based architecture design to code generation. The main functionalities of tMDA are presented in fig.1: Developers can model system requirements by "*Edit models*"and verify system models by "*Verify models*" which includes "*OCL check*", "*Consistency verification*", "*Activity verification*" and "*LTL check*". The consistency verification means verifying the dynamic consistency between UML statechart and sequence diagram. The use case "*Activity verification*" describes the verification of a activity diagram. "*Simulate models*" describes the process of model simulation. Besides, C++ code framework for the quality class diagram and statechart is generated automatically, which is described by the use case "*Code generation*".The implementation of each functionality is discussed in the following section.

2.1 Framework of tMDA

The framework of tMDA is shown in fig.2, which mainly contains four modules: ***Model Editor, Model Verifier, Model Simulator*** and ***Code Generator***. These four modules respectively accomplish the above functionalities. The control flow of the framework is shown as follows:

- Developers model system requirements in "***Model Editor***", which supports UML static models and dynamic behavior models.
- UML models generated in"***Model Editor***" can be input into "***Model Simu-***

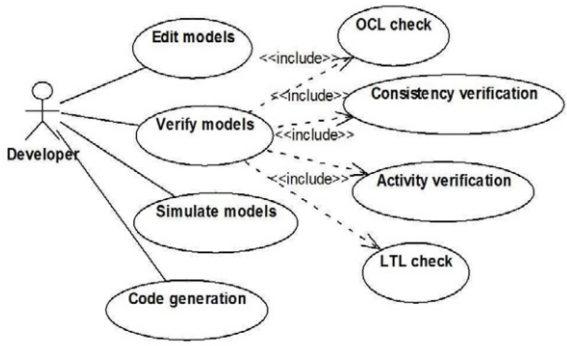

Fig. 1. Main Functionality of tMDA

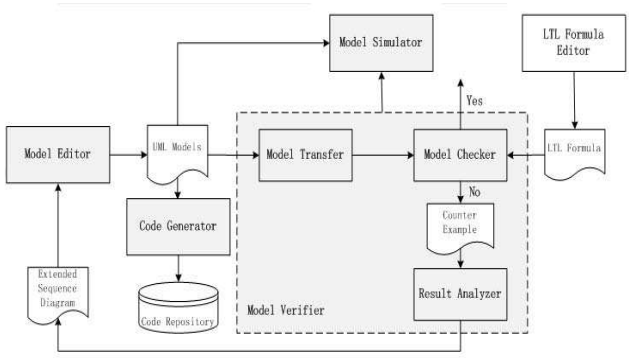

Fig. 2. Framework of tMDA

lator" to reappear the system execution path, which supports the analysis of system dynamic behaviors and helps to locate the error of models when an error is detected by "***Model Verifier***".

- On the other hand , UML models can be transformed to the input language of the model checker, which is accomplished by "***Model Transfer***". And then, the model checker is activated to verify UML models. If an error is detected, the counterexample will be generated by "***Model Checker***". With the help of the counterexample, developers can revise the models. The counterexample-guided refinement approach was proposed by E.M.Clarke [8]. However, how to guide developers to revise models is still a difficult problem. In this paper, we attempt to propose a feasible solution.
- The verified models can be transformed to high quality codes by "***Code Generator***", which facilitates the model-driven development for trustable software. The generated codes can be saved in "***Code Repository***" for code reuse.

The characteristics of the framework are generic and extendable. As have been known, different model checkers, such as SPIN [9], FDR [10], SMV [11] can be integrated in the framework for different verification purposes. In tMDA, we chose SPIN as the model checking engine due to its maturity and popularity.

2.2 Functionality of Each Module in tMDA

In tMDA, developers model system requirements with use case diagrams. According to the use case-driven requirement analysis approach, developers model the event flow of each use case with sequence diagrams to show the interaction between objects. Besides, class diagrams, statecharts and activity diagrams are used to model the static structure and dynamic behaviors of the system. The separation of concerns modeling approach provides different views for the whole system. However, the consistency between multi-view models should be guaranteed. The "***Model Editor***" is implemented based on a subset of the UML2.0 metamodel,which provides a user-friendly modeling environment. The core framework of "***Model Editor***" follows the MVC pattern. "***Model Verifier***"is the key part of tMDA, which contains "***Model Transfer***", "***Model Checker***" and "***Result Analyzer***". The functionality of"***Model Transfer*** " is to transform UML models to PROMELA-the input language of SPIN. If an error is detected in UML models, a counterexample described by formal specification language is generated by SPIN. However, the counterexample path is too complicated and too long to be understood by developers. Therefore, the counterexample should be interpreted and analyzed so that developers without formal mathematics background can also utilize the counterexample information to locate the error. Developers use "***Result Analyzer***" to accomplish the counterexample analysis [12]. According to the extended sequence diagram, developers can locate the error and then revise models in "***Model Editor***". The characteristic of "***Model Verifier***" is that it encapsulates the formal model checking technology, which helps developers reuse the existing model checker without requiring more detailed formal model checking technology. Besides, "***Model Verifier***" can also activate "***Model Simulator***" to simulate the execution path of system models. The algorithm of the simulator is implemented by execution the system path step by step according to the semantics of UML models. And the detailed simulation algorithm can be referred to [13]. Up to now, "***Model Simulator***" provides guidance for simulating the statechart and activity diagram.

Another notable characteristic of tMDA is that the LTL formula [14] can be generated by the property specification templates based on the common property specification patterns in embedded system. The expected properties of the system should be formulated with LTL formula. But, how to define the correct LTL formula is a difficult problem. A stepwise approach based on property specification patterns is proposed, which provides several LTL formula definition templates. Developers can choose appropriate template according to the property constraints of the system. The details of "***LTL Formula Editor***" can be consulted to [12] [13]. The integration of UML modeling and verification technology forms a rigorous modeling approach, which facilitates to generate high quality codes. According to the idea of model-driven development, we implement "***Code generator***" to generate C++ code framework for TUML (Trustable UML) models automatically. Based on the generated code framework, developers can customize the generated codes for specific functionalities. This approach will improve the efficiency of the code generation and the quality of codes.

2.3 Theory Background

Based on the theory of rCOS proposed by Jifeng He, et al [24], the rigorous component-based development approach provides the theory foundation for tMDA [15]. In tMDA, class diagram and component diagram are used to describe the static properties, or structural properties, whereas sequence diagram and activity diagram, as well as statecharts are used to describe the dynamic properties. Thus we describe the related models and show how to apply them to the modeling and verification process. For giving a formal definition of a class diagram, assume CN, AN and $AttrN$ are three disjoint sets, denoting class names, associations and attributes respectively. Each attribute of an object takes a value in a type of pure data called a data type. Examples of data types of natural numbers $\mathbf{N}$, Boolean values **Bool**, etc. Let $\mathcal{T}$ denote the set of the data types.

Definition 2.1 *A* **class diagram** *is a tuple* $< CN, Ass, Att, Inh, Meth >$*, where*

- *Ass is a partial function,* $\mathbf{C}$ *is a subset of* $\mathbf{CN}$*.* $Ass : \mathbf{C} \nrightarrow (\mathbf{AN} \nrightarrow \mathbb{P}\mathbf{N} \times \mathbb{P}\mathbf{N} \times \mathbf{C})$ *such that*

$$Ass(C_2)(A^{-1}) = \langle M_2, M_1, C_1 \rangle \text{ iff } Ass(C_1)(A) = \langle M_1, M_2, C_2 \rangle$$

  where $\mathbb{P}\mathbf{N}$ *is the powerset of* $\mathbf{N}$*. If* $Ass(C_1)(A) = \langle M_1, M_2, C_2 \rangle$*, then* A *is called an* association *between* C_1 *and* C_2*,* M_1 *and* M_2 *are called the cardinalities of* C_1 *and* C_2 *in* A*. An association* A *is in general denoted by* $A : (C_1, M_1, M_2, C_2)$*. We use* $AssN(C_1, C_2)$ *to denote the set of all the associations between* C_1 *and* C_2*.*

- *Att is a partial function* $Att : \mathbf{C} \nrightarrow (\mathbf{AttrN} \nrightarrow \mathcal{T})$*. We use* $C.a : \mathbf{T}$ *to denote* $Att(C)(a) = \mathbf{T}$*, and call* a *an* attribute *of* C *and* $\mathbf{T}$ *the* type *of* a*. We use* $attV(C)$ *to denote the set* $\{a : \mathbf{T} \mid Att(C)(a) = \mathbf{T}\}$ *of all the attributes of* C*.*

- $\mathbf{Inh} \subseteq \mathbf{C} \times \mathbf{C}$ *is the* direct generalization relation *between classes. We use* C_1 $\mathbf{Inh}$ C_2 *to denote* $(C_1, C_2) \in \mathbf{Inh}$ *and say that* C_1 *is a* direct superclass *of* C_2*, and* C_2 *is a* direct subclass *of* C_1*.*

- $Meth$ *is a mapping from* C *to a set of methods.*

We can use Java-like format to specify a class model as follows.

Class Model CM

Class $\mathbf{C}_{11}$ **Extends** $\mathbf{C}_{12}$ $\{\mathbf{T}_{11}\ x_1; \ldots; \mathbf{T}_{1m}\ x_m\}$

$\ldots\ldots$

Class $\mathbf{C}_{n1}$ **Extends** $\mathbf{C}_{n2}$ $\{\mathbf{T}_{n1}\ y_1; \ldots; \mathbf{T}_{nk}\ y_k\}$

Association $(M_1^1, \mathbf{C}_1^1, \mathbf{C}_1^2, M_1^2)\ A_1;\ \ldots;\ (M_j^1, \mathbf{C}_j^1, \mathbf{C}_j^2, M_j^2)\ A_j$

Invariant Φ

End CM

where $\mathbf{C}_1$ **Extends** $\mathbf{C}_2$ denotes that $\mathbf{super}(\mathbf{C}_1) = \mathbf{C}_2$. M_i^1 and M_i^2 are sets of natural numbers and represent the *multiplicities* of the roles $\mathbf{C}_i^1$ and $\mathbf{C}_i^2$ of association

A_i, $i = 1, \ldots, j$. A sequence diagram consists of objects and messages that describe how the objects communicate. An interaction occurs when one object invoke a method of another.

Definition 2.2 *A* **message** *is a tuple:* $Msg =< N\ s, Action, M\ t, Ord >$, *where*

- $N\ s$ *is an object* s *of class* N *called source of the message and can be denoted by a function* $source(Msg)$.
- $M\ t$ *represents the target object* t *of the message and its type* M, *can be denoted by* $target(Msg)$.
- *Action, denoted by* $action(Msg)$, *is a guarded method call of the form* $g \longrightarrow act$, *where* g *is a Boolean expression of attributes of* $source(Msg)$ *that are not associations, and act is either a command without method calls (an internal action) or a method name* m.
- Ord *is a natural number representing the order of the message in the sequence diagram, denoted by* $order(Msg)$.

A guarded command $g \longrightarrow act$ is defined as $g \wedge act$. We require that if action (Msg) is a guarded command without a method call then $source(Msg) = target(Msg)$, i.e. interactions between objects can only be carried out via method invocation.

Definition 2.3 *A* **sequence diagram** *is a tuple* $SD =< Actor\ a, Start, MSG >$

- *Actor is the initiating (actor) object that calls the message Start;*
- *Start is the starting such that* $source(Start) = Actor\ a$.
- *A set of messages* MSG *contains* $Start$, $order(Start) < order(Msg)$ *for any other message in* MSG, *and if* $order(Msg_1) = order(Msg_2)$ *only if* $source(Msg_1) = source(Msg_2)$.

$MS(Msg)$ denotes the sequence of message $Sm = \{Msg_1, ...Msg_k\}$ in which message Msg_2 *directly follows* message in a sequence diagram Msg_1 if $order(Msg_1) < order(Msg_2)$, but there is not a message Msg in the diagram such that $order(Msg_1) < order(Msg) < order(Msg_2)$. Usually, the target object in the starting message is a use case handler and the operation is a method (i.e. a use case) call of the handler.

The specification of a sequence diagram is given by the definition of methods in

the classes. Assume N is a class of target object $Tobj$, m_i is a method of N:

$i := 1;$

while $i \leq k_1$ **do**{

Class$N :: m_i$**{** $// *$ for each start = < o:M, t.m, t:N, ord >

$j := 1;$

while $j \leq k_2$ **do**{

if$\{(g_{ij} \longrightarrow Action(Msg_{ij}))|1 \leq j \leq k_2\}$**fi**};

$j := j + 1\}$;

$i := i + 1\}$;

where k_1 is the number of the messages for which N is the class of target object, k_2 is the number of massages handled between m_i and m_{i+1}. Msg_{ij} are all those messages from source object to target object. g_{ij} is the guard of executing message Msg_{ij}.

Definition 2.4 *Given the specification of a design class diagram DC and a family of sequence diagram SD. DC and SD are* consistent *if*

- *For each message* $Msg =< N\ s, Action, M\ t, Ord >$ *in the SD, the target object* $M\ t$ *is declared as an attribute of the class* N *of source object* $N\ s$.
- *If* $action(Msg)$ *is* $g \longrightarrow m$ *and* m *is a method name then* m *is a defined method in the class of* $target(Msg)$.
- *The corresponding class declaration section* $cdecls_d$ *obtained from DC is well-defined.*

In component-based design, component acts as essential module. A component has of a set of interfaces, or port, to communicate with other components. In fact, each component offers some business functionality through ports, which is implemented by method call.

Definition 2.5 *A* **port** p *is a tuple* (M, t, c)*, where* M *is a finite set of methods in* p*,* t *is the port type that can be provided or required and* c *is the communication type that can be synchronous or asynchronous.*

We use $p.M$ *to denote the operation set of port* p*,* $p.t$ *to denote the port type and* $p.c$ *to denote the communication type, where* $p.t = \{providedport, requiredport\}$*,* $p.c = \{synchronous, asynchronous\}$

Definition 2.6 *A* **component** Com *is a tuple* (P_p, P_r, G, W)*, in which* P_p *is a finite set of provided ports,* P_r *is a finite set of required ports,* G *is a finite sub component set,* $W \subseteq TP \times \bigcup_{C \in G}(C.P_p \cup C.P_r)$ *, is the port relation that is non-reflexive, where* $TP = P_p \cup P_r \cup \bigcup_{C \in G} C.P_r$*,* $C.P_p$ *and* $C.P_r$ *denote the provided and required port sets of the subcomponent respectively.*

A component $Com = (P_p, P_r, G, W)$ *is an atomic component if* $G = \Phi \wedge W = \Phi$

, otherwise Com is a composite.

The contract is the specification of a port to describe the behavior semantics of a component, which is modeled with OCL expressions in tMDA.

Definition 2.7 *A* **contract** *Ctr is a quadruple* $(P, Init, Spec, Prot)$ *where*

- P *is a port.*
- *Spec maps each operation* m *of* P *to its specification* (a_m, g_m, p_m) *where*

(i) a_m *contains the resource names of the port* P *and the input and output parameters of* m.

(ii) g_m *is the firing condition of operation* m*, specifying the environments under which* m *can be activated.*

(iii) p_m *is a reactive design, describing the behavior of* m.

- $Init$ *identifies the initial states.*
- $Prot$ *is a set of operations or service calling events* $<?m_1(x_1)...?m_j(x_j)>$ *i.e. behavior protocol of components. Here, protocols are introduced to coordinate the interactions between components with the environment.*

In tMDA, the protocol $Prot$ is modeled by UML sequence diagram. While there is some work on the protocol verification by model checking sequence diagram, we implement the protocol verification by verifying the consistency between statechart and sequence diagram. That is whenever the actors follow the interaction protocol described by the sequence diagram and the set of traces can be accepted by the statecharts. A statechart describes the process of state transitions in the life cycle of an object. The semantics model of statechart is represented by a Label Transition System (LTS). According to LTS, the formalization of statechart is as following:

Definition 2.8 *A* **statechart** *is quadruple* $SM =< S, s_0, L, \Delta >$*, where* S *denotes the set of states;* s_0 *denotes the initial state;* L *labels a set of atomic propositions for every state* $L : S \longrightarrow 2^{AP}$ *, which is described by* $e \rightarrow action$*,* e *is the triggered event and action is the generated action;* Δ *denotes the transition relation, where* $\Delta \subseteq S \times (L \cup \varepsilon) \times S$

Definition 2.9 *The* **trace** *of a staechart* $SM =< S, s_0, L, \Delta >$ *is an ordered set of transitions* $tr = \{\forall i, (0 \leq i \leq k) | (s_i, l_i, s_{i+1})\}$*. We use* $Ev(tr)$ *to represent the set of events in the trace of a statechart, where* $Ev(tr) = \{ev(l_0), ev(l_1), ev(l_2), ...ev(l_{k-1})\}$ *and* $sord(ev(l_0)) < sord(ev(l_1))$*,i.e the event* $ev(l_1)$ *follows* $ev(l_0)$ *directly,* $ev(l_i)$ *denotes the triggered event of* l_i.

Definition 2.10 *Given the specification of a sequence diagram SD and a family of statecharts Schart. SD and Schart are* consistent *if*

- Sm *is a sequence of messages and* $Sm = \{Msg_1, ...Msg_k\}$ *is in SD, then each message* $Msg =< N\ s, Action, M\ t, Ord >$ *should be defined in the SD.*
- *the sequence of sending, or receiving, message in a sequence diagram corresponds to the trace of the interaction behaviors described by statecharts. That is,if* Tr *is a set of events,* $Tr = \{ev_0, ev_1, ev_2, ..., ev_k\}$*, or* $\forall\ i, (k \geq i \geqslant 0) | (ev_i) \in Tr$*,* then

Sm can be mapped to Tr.

3 Case Study: A Bank ATM System

In this section, we will demonstrate how tMDA facilitates the modeling and analysis of system requirements. The main purpose is to illustrate the feasibility of integration the formal verification technology into the model-driven development process to improve the quality of system models. The development process is use case-driven and component-based,which includes the following steps. Firstly, we model the system requirements with use case diagrams. Secondly, the component diagram is given to model the system services. And then, according to the component diagram, the class diagram, sequence diagram, statechart diagram are constructed to describe the static structure and dynamic behaviors of the components. During the modeling process, the formal model checking technology is applied to verify the correctness and consistency of the different UML models. The whole modeling and verification process is progressed in an integrated environment tMDA developed by ourselves.

3.1 Use case-driven Requirement Analysis and Design

We will illustrate our approach with a simple BAS case study, which describes the several scenarios of the interaction between an automatic teller machine (ATM), a bank computer and a single user. A user can deposit money, withdraw money and transfer money on some ATM, which is described by the use case diagram for BAS (shown in fig.3). These use cases can be seen as the contract of some components. For example, these services described by use cases can be provided by the *Transition* component (shown in fig.4(left)). In the component diagram, we just show the provided service withdrawMoney of *Transition* component which interacts with the component ATM and Bank through the ports. Unlike the usual approach, the requirement component diagram is derived from the use case diagram. That is, in the requirement analysis process, we start from analyzing the service needed by the customers, then, assign the service to the port, which in turn, delegates the service to the components. Afterward, the components can be designed and implemented based on tMDA. The collaboration classes composing the components are presented in the class diagram (shown in fig.4(right)), which lays out the static structure of the system. The dynamic behaviors are specified by the statecharts for the class ATM and Bank (shown in fig.5,6). Besides, a certain scenario of the interaction protocol is modeled by the sequence diagram (shown in fig.7).

3.2 Verification of Requirements Consistency

3.2.1 Consistency between Class Diagram and Sequence Diagram

The consistency of static structure and dynamic behavior is very important, however, this kind of consistency is often ignored by developers. The messages appearing in the sequence diagram should already be declared in the related class diagram. The model verifier of tMDA can verify this kind of consistency by the syntax check.

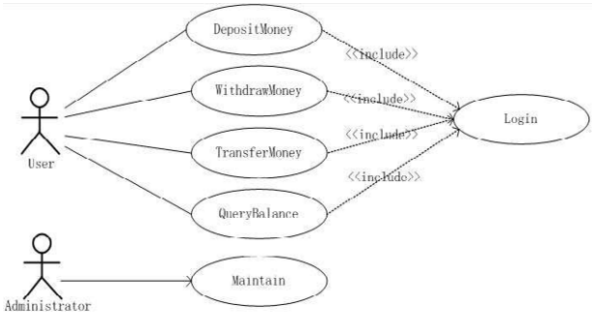

Fig. 3. Use case diagram for BAS

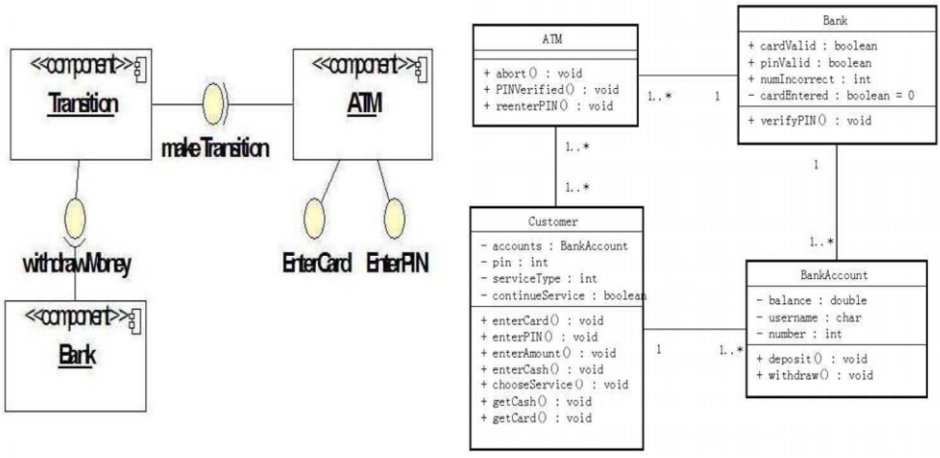

Fig. 4. Component diagram for BAS (left) and class diagram for BAS (right)

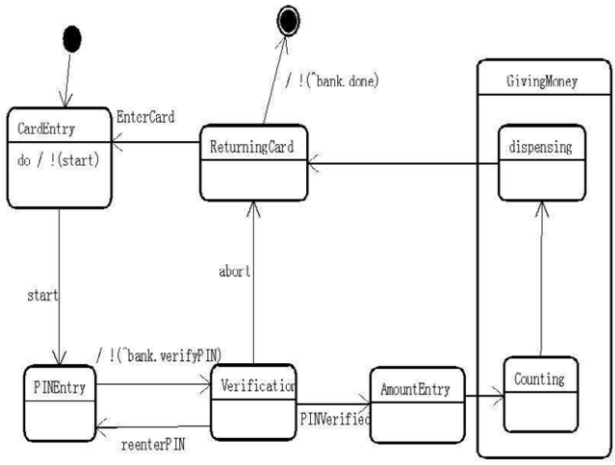

Fig. 5. Statechart for ATM

For example, the message $verifyPIN$ in the sequence diagram shown in Fig.7. We can get $Msg = (ATM, Action, Mt, Ord)$, where $Action(Msg) = verifyPIN$,

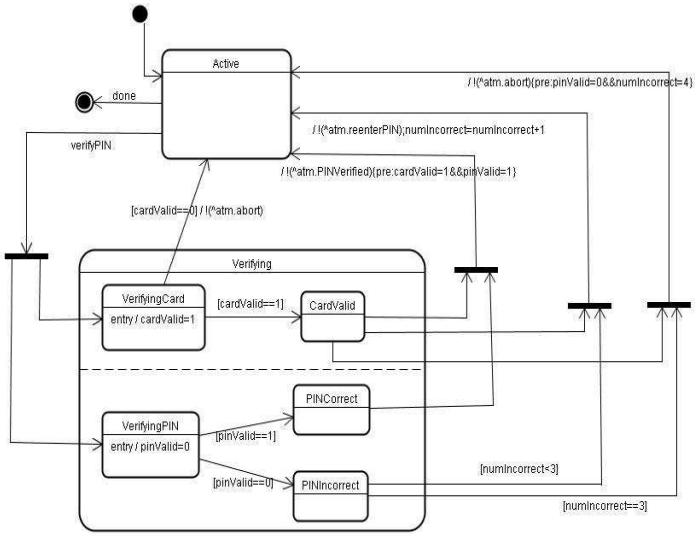

Fig. 6. Statechart for Bank

$target(Msg) = Bank$. According to the definition 2.4, if the class diagram is consistent with the sequence diagram, the event $verifyPIN$ should be declared in the class of Bank. Otherwise, the inconsistency warning message will be generated.

3.2.2 Consistency between Statechart and Sequence Diagram

The main functionality of "***Model Verifier***" is to verify the dynamic consistency between statechart and sequence diagram. Informally, the consistency must ensure that whenever the actors follow the interaction protocol described by the sequence diagram and the set of traces can be accepted by the statechart. But, how can we verify the consistency between statechart and sequence diagram with SPIN? LTL formula is suitable to describe the linear temporal sequence of events occurrence, so, we can use LTL formula to represent the traces of state transition triggered by the events in the sequence diagram. In nature, the verification of the consistency between statecharts and sequence diagram means to verify whether the statechart satisfy the LTL formula which describes the interaction trace of the sequence diagram. So, the algorithm of transforming the sequence diagram to LTL formula is the key point, which utilizes the characteristics of the sequence diagram to generate the LTL formula automatically. In BAS case study, the interaction between statecharts should comply with the protocol specified by the sequence diagram(shown in fig.7). Note, we extend the UML sequence diagram by adding the state information to the message passing, which records the next state entered by the object. For example, after the object ATM sends the event *verifyPIN* to the object Bank , the object ATM will enter the state *Verifying*.

The extended sequence diagram (shown in fig.7) specifies the behavior protocol in which if some customer wants to withdraw money, he must enter the correct PIN code. If the PIN code is wrong, he can reenter the PIN code. For simplicity, we only discuss the correct behaviors of the protocol.

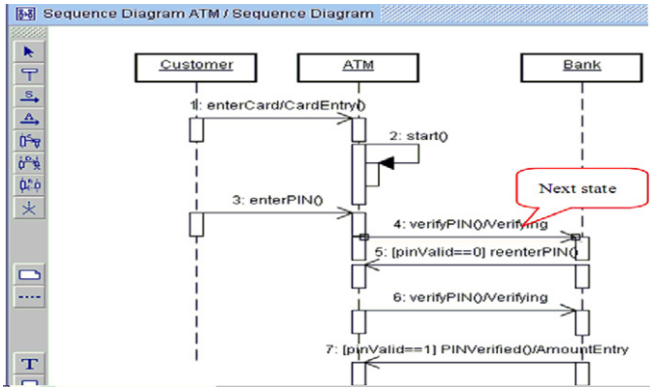

Fig. 7. Extended sequence diagram for BAS

According to the definition 2.3, we can get the event sequence of this scenario $Sm = \{enterCard, verifyPIN, reenterPIN, verifyPIN, PINverified\}$. The interaction behaviors of the statechart for ATM and Bank can be described by the trace $tr = \{(CardEntry, enterCard, PINEntry), (PINEntry, verifyPIN, Verifying), (Verifying, reenterPIN, PINEntry), (PINEntry, verifyPIN, Verifying), (Verifying, PINverified, AmountEntry)...\}$. So, $Ev(tr) = \{enterCard, verifyPIN, reenterPIN, verifyPIN, PINVerified\}$. The statecharts is consistent with the sequence diagram if $Sm \subseteq Ev(tr)$, according to the definition 2.7. tMDA can automatically generate the LTL formula for the sequence diagram supported by SPIN. The LTL formula is as follows:

```
#define p0 CardEntry
#define p1 Verifying
#define p2 PINEntry
#define p3 AmountEntry
 <>(p0&&<>p1)&&<>(p1&&<>p2)||<>(p2&&<>p3)     (LTL formula )
```

Note, p0,p1,p2,p3 represent the state information of interaction behaviors. So, the LTL formula tracks the record of system state transition process. Once the LTL formula is generated, the verification of consistency between the statecharts and sequence diagram can be implemented. If we delete the transition *PINVerified* from the statechart of ATM (fig.5) and perform the consistency verification, the results show that there is an invalid end state in the statechart. The experiment demonstrates the interaction between the statecharts don't comply with the protocol described by the sequence diagram.

3.3 Verification of Liveness Property of Statechart

The most general approach of verifying UML statechart is that translating the statechart into the input language of the model checker. Some tools have been developed for the analysis of system models specified in terms of UML, such as vUML [17], Hugo [18]. The work by Lilius and Paltor discussed a formalization of

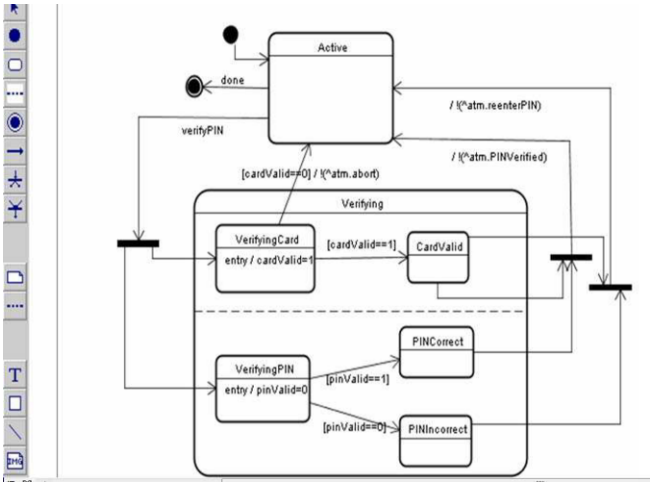

Fig. 8. Modified statechart of Bank

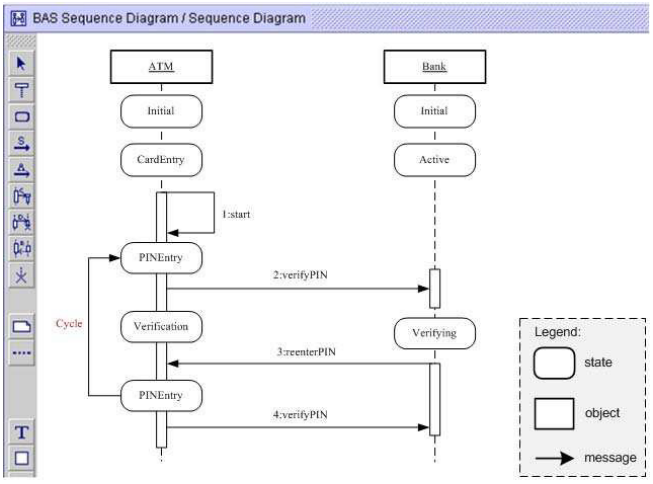

Fig. 9. Extended sequence diagram for the counterexample

UML state machines for translation to PROMELA as part of their verification tool vUML. The main features of vUML are (1)The verification performed by vUML is limited to deadlock checking and some robustness checking.(2)Only a subset of UML statecharts is considered. Different from their work, we pay attention to the properties verification of statecharts, such as liveness or safety. This subsection illustrates how to verify the liveness property of the statecharts in BAS case study.

System specifications satisfying liveness property means that the good things will happen at last. SPIN verifies the liveness property by exploring whether there is a cycle path in the execution process of a system. For example, we verify whether the ATM models satisfy the liveness property. The modified statechart is shown in fig.8, which represents if the PIN code is wrong, the customer will reenter PIN code continuously. This don't accord with the fact, as we all known, it is impossible for the customer to enter the PIN code so many times. An error is found and

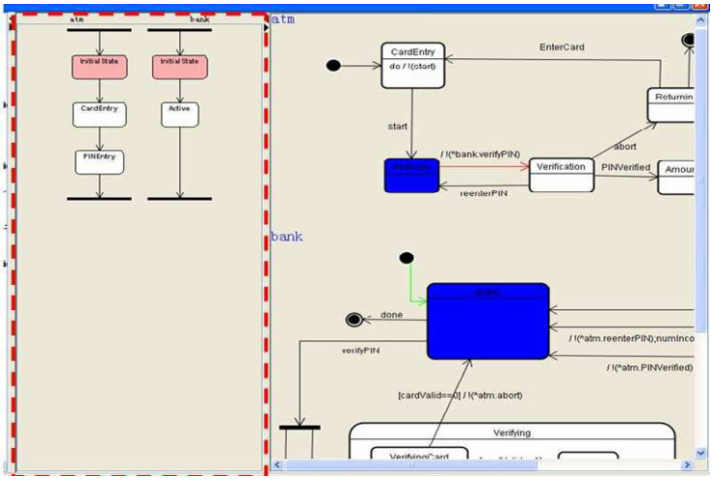

Fig. 10. Simulation of the statechart

the counterexample is generated when we perform the liveness verification of the statecharts. According to the counterexample, the extended sequence diagram is produced shown in fig.9. There is a cycle path in which the user continues to reenter the PIN code. To locate the error, developers can also simulate the execution path of the system, shown in fig.10. The left dotted rectangle is the message interaction process when we simulate the statecharts. During each step is performed, the corresponding event and state information will be generated in the left dotted rectangle. The right part is the simulation process of the statecharts, where the blue state represents the active state, the red transition represents the next triggered transition and the green transition represents the performed transition. The simulation of statechart helps to analyze the detailed execution path of a system so that the error can be located easily.

3.4 Verification of LTL Formula

It is difficult for developers to define and understand the formal LTL formula, because LTL has complicated syntax and semantics. This subsection discusses how to help developers define LTL formula. In tMDA, the property definition templates for LTL formula can be generated based on Property Specification Patterns (PSP) proposed by Dwyer [19]. Dwyer's specification patterns system contains several patterns applicable to software properties specified in different formalisms, such as LTL, CTL [20]etc. The PSP system is categorized into two major groups: *occurrence patterns* and *order patterns*, which is called *Qualitative* shown in fig.11(left). *Occurrence patterns* are concerned with the occurrence of single state/events during the execution of a program, such as existence and absence of certain states/events. *Order patterns*, on the other hand, are concerned with the relative order of multiple occurrences of states/events during the execution of a program, such as *precedence* or *response* relations between states/events. Besides, S.Konrad [21] proposed the Real-time Specification Patterns (RSP), which focused on the time-related property in embedded systems. RSP has three type patterns: duration, periodic and real-

time order. The presentation of RSP is similar to Dwyer's pattern system. The PSP and RSP represent the two categories property patterns: *Qualitative* and *Real-Time*, which compose the *Specification Patterns System*. These specification patterns have been found sufficient to specify most commonly occurring properties [22].

In tMDA, developers can choose the proper property definition template according to the system constraints. The stepwise generation process can be referred to [13]. For example, the definition template for **Response Pattern** is shown in fig.11(right).

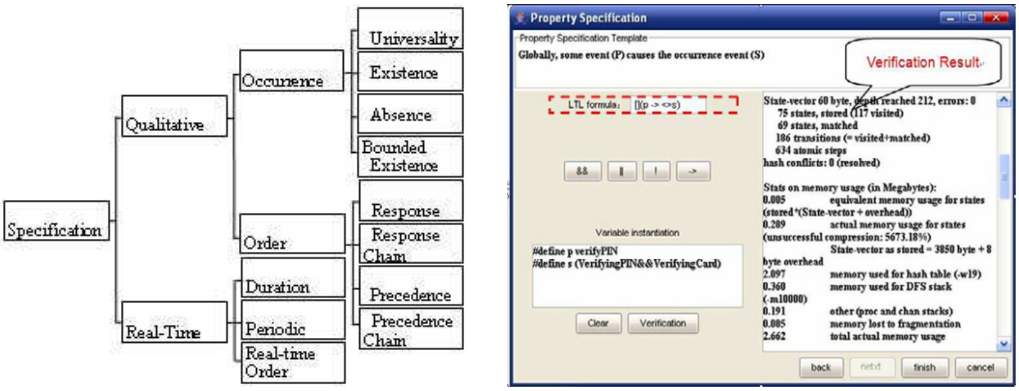

Fig. 11. Category of SPS (left) and property definition template for LTL formula (right)

The generated LTL formula is as such:
$[](verifyPIN \rightarrow <> (VerifyingCard \&\& VerifyingPIN))$, which means the system enters the state *VerifyingCard* and the state *VerifyingPIN* due to the occurrence of the event *verifyPIN*. Afterwards, tMDA can verify whether the statecharts satisfy the constraints specified by the LTL formula. The verification result is shown in the right part of fig.11(right).

3.5 Verification of OCL Expression for the Contract

The core of OCL is given by an expression language. Expressions can be used in various contexts, for example, to define constraints such as class invariants and pre/post conditions of a method. To model the method contract in UML models, we utilize the constraints denotation mechanism of UML and take the OCL expression as a kind of constraint on the event of the transition. This kind of modeling approach can express the pre/post condition of some method. In this subsection, we illustrate how to verify OCL expressions with SPIN. In the context of statechart, we model method contract with OCL pre/post conditions of the triggered events. In tMDA, every OCL constraint expression represents the condition should be satisfied when the method is activated, which can be taken as the assertion of the PROMELA program. So, the key point of our approach is to transform the OCL expression to an assertion. Thanks to the assertion verification of SPIN, the OCL expression can be verified. In the case study, we define the OCL expressions in the statechart shown in fig.12.

The transition "$!(atm.PINVerified)\{pre : cardvalid = 1 \&\& pinValid = 1\}$" denotes that the object Bank can send *PINVerified* event to the object ATM, if

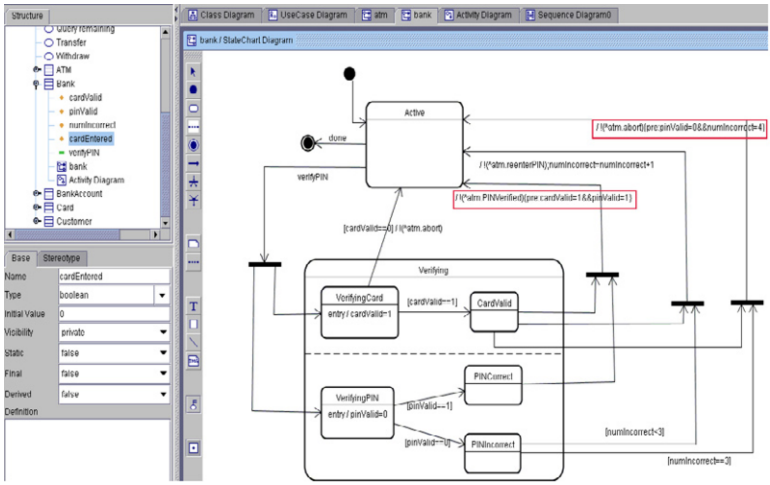

Fig. 12. Verification of OCL expressions in statechart

the precondition "$cardvalid = 1 \&\& pinValid = 1$" is satisfied. The assertions generated from the OCL expression: $\{pre : cardvalid = 1 \&\& pinValid = 1\}$ are "$assert(cardValid == 1)$;" and "$assert(pinValid == 1)$".

4 Conclusion and Related Work

B.Hnatkowska,et al discussed the problem of consistency among components of UML system model [23]. They proposed OCL to formalize the consistency conditions that must hold between model components, where a state diagram of a class vs. a class diagram and a state diagram of a class vs. state diagrams of other classes. However, their work laid the emphasis on the well-formed rules described by OCL constraints, which just verified the syntax correctness of the model. A research team in UNU-IIST has made great progress in applying formal specification technology to the model-driven component-based software development process [24][25]. In CoCoME project, they use the model checker FDR to prove trace equivalence of the sequence and the state diagram and identify where formal methods support can be "plugged" into the tool MasterCraft [26] to make software development more efficient [27]. A verification approach of rCOS using SPIN was proposed by X. Yu, et al [28]. They transformed rCOS specification to PROMELA codes and then call SPIN engine to verify the generated PROMELA codes.

Our work involves the whole MDA-based software development process including requirement modeling, verification, simulation and code generation. More important, a rigorous modeling and development approach is implemented. We presented our recent experience on the application of the formal verification technology to the MDA-based software development process. Through BAS case study, the integrated modeling and verification environment tMDA is illustrated to be useful for modeling the system requirements and verifying the consistency between multi-view UML models.

The main contributions of our work include:

(i) An integrated modeling and verification environment is implemented, which applies the idea of DbC to modeling system requirements for ensuring the correctness of requirement models. Use case diagram is used to model the contract of components. Based on the contract, the service needed by the system is derived and assigned to the ports of components. On the other hand, the method contract is modeled with OCL expressions in statecharts, which presents the pre/post condition of some method.

(ii) A multi-dimension verification model is proposed, which contains serval verification functionalities:
- Verification of the consistency between static models and dynamic models, such as the consistency between class diagram and sequence diagram.
- Verification of the liveness or safety property of statecharts.
- Verification of the consistency between dynamic models, such as statechart and sequence diagram, which can verify whether the interaction between statecharts comply with the protocol modeled by the sequence diagram.
- Verification of OCL expressions is accomplished by transforming OCL expressions to the assertions of PROMELA.

Our future work includes extending tMDA to support SOA-based software development, especially, to implement the modeling and development of trustable service components. We will also explore pattern based refinement rules to facilitate the model transition. This helps to accomplish the transition from PIM models to PSM models

Acknowledgement

We are grateful to the reviewers for their careful reading and detailed comments that helped us to improve the paper. This work is partially Supported by the National Key Research program of Dependable Software Theory under Grant No. 90718014; National High Tech Research 863 Program of China under Grant No. 2006AA01Z165; the National Natural Science Foundation of China under Grant No. 60673114 and No. 60603037.

References

[1] X. Li, Z. Liu and J. He, *Consistency Checking of UML Requirement*, Proc. of the 9th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS2005, pages 411-420, IEEE Computer Society Press, 2005.

[2] Z. Liu, J. He and J. Liu, *Unifying Views of UML*, Electronic Notes in Theoretical Computer Science, 101, pages 95-127, Elsevier, 2004.

[3] B. Meyer, *Object-Oriented Software Construction*, Upper Saddle River, Prentice Hall, 1997.

[4] B. Meyer, *Eiffel: The language*, Prentice Hall, 1992.

[5] R. R. Lutz, *Targeting Safety-related Errors During Software Requirements Analysis*, Journal of Systems and Software, pages 222-230, 1993.

[6] C. A. R. Hoare and J. He, *Unifying Theories of Programming*, Prentice-Hall, 1998.

[7] P. Kruchten, *The Rational Unified Process-An introduction*, Addison-Wesly, 2000.

[8] E. M. Clarke, O. Grumberg and S. Jha, *Counterexample-Guided Abstraction Refinement*, Lecture Notes in Computer Science 1855, Springer-Verlag, 2006.

[9] G. J. Holzmann, *The Model Checker SPIN*, IEEE Transaction on Software Engineering, pages 279-295, 1997.

[10] A. W. Roscoe, *Model Checking CSP*, In A.W. Roscoe, editor, A Classical Mind: Essays in Honour of C.A.R. Hoare, Prentice Hall, pages 353-378, 1994.

[11] K. L. McMillan, *Symbolic Model Checking: an Approach to the State Explosion Problem*, Report CMC-CS-92-131, 1992.

[12] D. Du, H. Cao and K. He, *MCF4U: A Flexible Model Checking Framework for UML Models*, DCDIS Series B, Vol.14(S6), special issue on software engineering and complex networks, pages 281-286, 2007.

[13] D. Du, *Research on UML Model Checking Framework and Key Technologies*, Ph.D. thesis, Wuhan University, Wuhan, 2007.

[14] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.

[15] Z. Ding, Z. Chen and J. Liu, *A Rigorous Model of Service Component Architecture*, Electronic Notes in Theoretical Computer Science 207, pages 33-48, Elsevier, 2008.

[16] J. Liu, J. He, *A Strategy for Model Construction and Intergation in MDA*, Journal of Software, Vol17, No.6, pages 1411-1422, 2006.

[17] J. Lilius, *vUML: A Tool for Verifying UML Models*, Proceedings of the 14th IEEE International Conference on Automated Software Engineering, ASE1999, IEEE Computer Society Press, 1999.

[18] T. Schafer, A. Knapp, and S. Merz, *Model Checking UML State Machines and Collaborations*, Electronic Notes in Theoretical Computer Science 55(3), Elsevier, 2001.

[19] M. B. Dwyer, G. S Avrunin and C. Corbett, *Patterns in Property Specifications for Finite-state Verification*, Proceedings of the 21st International Conference on Software Engineering, ICSE1999, pages 411-420, IEEE Computer Society Press, 1999.

[20] E. M. Clarke and E. A. Emerson, *Design and Synthesis of Syschronization Skeletons Using Branching Time Temporal Logic*, Logic of Programs, Workshop, Yorktown Heights, NY, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.

[21] S. Konrad and B. H. C. Cheng, *Real-time Specification Patterns*, Proceeding of the 27st International Conference on Software Engineering, ICSE2005, ACM Press, 2005.

[22] M. B. Dwyer and G. S Avrunin, C. Corbett, *Patterns in Property Specifications for Finite-state Verification*, Proceeding of the 21st International Conference on Software Engineering, ICSE1999, pages 411-420, IEEE Computer Society Press, 1999.

[23] B. Hnatkowska, *Consistency Checking in UML*, Proceedings of 4th Internation Conference on Information System Modeling , 2001.

[24] Z. Liu and J. He, *Towards a Rigorous Approach to UML-Based Development*, Electronic Notes in Theoretical Computer Science 130, pages 57-77, Elsevier, 2005.

[25] J. He, X. Liu and Z. Liu, *rCOS: A Refinement Calculus of Object Systems*, Electronic Notes in Theoretical Computer Science 365, pages 109-142, Elsevier, 2006.

[26] Tata Consultancy Service. MasterCraft. URL:http://www.tata-mastercraft.com/.

[27] Z. Chen, X. Li and Z. Liu and V.Stolz, *Harnessing rCOS for Tool Support: The CoCoME Experience*, UNU-IIST, Report No.388, 2007.

[28] X. Yu and Z. Wang, *The Verification of rCOS Uing SPIN*, Electronic Notes in Theoretical Computer Science 207, pages 49-67, Elsevier, 2008.