

Formal Specification of UML Requirement Models

Xiaoshan Li¹ and Zhiming Liu²

¹ Faculty of Science and Technology, The University of Macau, Macau.
xsl@umac.mo

² Department of mathematics and Computer Science,
The University of Leicester, Leicester, UK
Z.Liu@mcs.le.ac.uk

Abstract. The Unified Modeling Language (UML) is the de-facto standard modeling language for the development of software with broad ranges of applications. It supports for modeling a software at different stages during its development: requirement analysis, design and implementation. The use of UML encourages software developers to devote more effort on requirement analysis and modeling to produce better software products. The most important models to produce in an object-oriented requirement analysis are a *conceptual class models* and a *use-case models*. This paper proposes a method to combine these two models by using a classic *transition system*. Then we can reason about and refine such systems with well established methods and tools.

Keywords: *Object-orientation, UML, Conceptual Model, Use-Case Model, Object-Orientation, Transition Systems*

1 Introduction

Object-orientation is now a popular approach in software industries. The Unified Modeling Language (UML) [BRJ99,RJB99,JBR99] is the de-facto standard modeling language for the development of software with broad application ranges, covering the early development stages of requirement analysis and with strong support for design and implementation. [BRJ99,DW98]. One of the main advantages of UML is that different modeling diagrams are used at different stages to represent the system from different views at different levels of abstraction.

The main models for the requirement analysis of a system are a *conceptual model* and a *use-case model*. The conceptual model represents the domain concepts as *classes* and their relationships as *associations*. It determines the possible *objects* and relationships between these objects. Requirement analysis is not usually concerned very much about what an object does or how it behaves [Lar98,DW98,Liu00]. Therefore, a conceptual model is mainly used as a *static model* of the *structure* of the application domain.

The use-case model is used to specify the required functional services that the system is expected to provide for different kinds of users. A use-case model contains a number of *use cases*. Each use case describes a pattern of interactions between some users and the system.

One of the main problems when using UML is to ensure consistency between different diagrams used in a system development. When there is not yet a well established semantics for the whole language it is impossible to check consistency or to reason about relationship among the different models. In [EKHG01], problems concerning consistency between models for different views are classified as *horizontal consistency* and those about models at different levels of abstraction as *vertical consistency*. And consistency of each kind is divided into *syntactical consistency* and *semantic consistency*. Obviously semantic consistency requires syntactical consistency. Formal treatment of these kinds of consistency in fact requires the establishment of a formal framework for the specification of object-oriented software systems and the manipulation of such specifications through well disciplined transformations.

Syntactical consistency conditions are expressed in UML in terms of the well-formedness rules of OCL (Object Constraint Language). The article [EKHG01] defines and checks a particular *behavioral consistency* between different statecharts by translating them into Hoare's CSP. The work in [Egy01] deals with automated checking of horizontal syntactical consistency among models, such as design class diagrams and object sequence diagrams.

There is currently a lot of active research on formalization of UML. However, most of it focuses on translating a individual UML notation into an existing formal notation. For example, a class diagram is in Z or VDM [Ken97,pG99], and an interaction diagram or a statechart is translated into a CSP specification [EKHG01]. For UML to be more effectively and precisely used in a software development process, more research is needed on the *relationships* among the different models used in UML. This work is an attempt in this direction.

The long term aim of this research is to support formal use of UML in OO system development processes and development of tools for consistency checking. The method is expected to be usable within an incremental and iterative Rational Rose Development Process (RDP) [JBR99]. We believe this will on the one hand to change today's situation that OO software development in practice is usually done in a non-scientific manner [Hoa96] based on pragmatic and heuristics. On the other hand, with incorporation of our method into RDP, we hope to improve the use of formal methods in the development of large scale system.

This paper proposes a method for specifying and reasoning about the UML conceptual model and the use cases of a system. It is based on the well known notation of *transition systems* [MP81] of the form $\mathcal{S} \stackrel{def}{=} (\Gamma, Inv, Init, P)$, where

- Γ is a set of declared state variables with known value domains. These variables and their data domains are constructed from the conceptual model.
- Inv is a state predicate called the *invariant* of the system. It has to be true during the operation of the system. This is determined by the conceptual model too.
- $Init$ is a state predicate determining the initial states of the system. It is establish by the installation of the system.
- P is a set state transitions that models the execution of the use use cases.

Both syntactic and semantic consistency between a conceptual model and a use case model are taken into account in the formal definitions of a conceptual model, object diagrams and system operations.

After this introduction, a syntax and a semantics for a conceptual model are defined in Section 2. The syntax follows the traditional graph definitions. The semantics of a conceptual model is defined in terms of the variables, their value domains and the object diagrams as the state space of the model. Section 3 defines a syntax and semantics of the a use-case model. The semantics of a use case is defined based on the semantics of the conceptual model and how it carries out state transitions. This will lead to a combination of a conceptual model and a use case model into a transition system. Finally conclusion and discussion are given in Section 4.

2 Conceptual Model

One of the main artifacts to produce in an OO analysis is a conceptual class diagram. Such a diagram captures the physical *concepts* and their *relations* of the system's application domain. In UML, a concept is represented by a *class* with a given name. An instance of a concept is called an *object* of the corresponding class. A relation between two concepts are denoted by an *association*. In addition to associations between concepts, a concept may have some *properties* represented by *attributes*. For example, **Account** has a *balance* as an attribute and **Customer** has a *name* as an attribute. There are two approaches to deal with attributes. The first is to introduce *types of pure data values* [BRJ99] and then to represent attributes as *component fields* of classes. Alternatively, these types of pure data values can be treated as classes and attributes as associations [LHL01,LLH01]. In this paper, we follow the former approach.

We must understand that at the requirement level, a class simply represents a set of objects. A system requirement specification is concerned with what the system does as a whole rather than what an individual object does, how an object behaves, or how an attribute of an object is represented. The decision on the later issues will be made during design. A use case is designed by decomposing its *responsibilities* and assigning them to appropriate objects [Lar98,Liu00]. Use case decomposition and responsibility assignment are carried out according to the *knowledge* that the objects maintain³. *What an object can do depends on what it knows, though an object does not have to do all what it can do.* What an object knows is determined by its attributes and associations with other objects. Only when the responsibilities of the objects are decided in design, can the directions of the associations (i.e. *navigation* and *visibility*) and the methods of the classes be determined. This indicates that an association has no direction or equivalently two directions and a class has no methods.

2.1 Conceptual class diagram

To define a syntax for class diagrams, we introduce three disjoint sets of names **CName**, **AName**, and **attrName** to denote classes, associations and attributes. For each name, $A \in \mathbf{AName}$, we assume there is a unique name $A^{-1} \in \mathbf{AName}$ called the *inverse* of A , and $(A^{-1})^{-1} = A$.

³ This is the main idea of the design pattern called *Expert Pattern*.

Each attribute of an object takes a value in a *type of pure data* called a *data type*. Examples of data types include types of natural numbers \mathbf{N} , integers \mathbf{Int} , Boolean values \mathbf{Bool} , characters \mathbf{char} , etc. Let \mathcal{T} denote the set of the data types.

Definition 1. (Conceptual Class Diagram) A conceptual class diagram is a tuple: $\Delta = \langle \mathcal{C}, Ass, Att, \triangleleft\text{---} \rangle$, where

- \mathcal{C} is a nonempty finite subset of \mathbf{CName} , called the *classes* or *concepts* of Δ .
- Ass is a partial function $Ass : \mathcal{C} \rightarrow (\mathbf{AName} \rightarrow \mathbb{PN} \times \mathbb{PN} \times \mathcal{C})$ 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{PN} 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 $Att : \mathcal{C} \rightarrow (\mathbf{attrName} \rightarrow \mathcal{T})$. 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 .
- $\triangleleft\text{---} \subseteq \mathcal{C} \times \mathcal{C}$ is the *direct generalization relation* between classes. We use $C_1 \triangleleft\text{---} C_2$ to denote $(C_1, C_2) \in \triangleleft\text{---}$ and say that C_1 is a *direct superclass* of C_2 , and C_2 is a *direct subclass* of C_1 .

Definition 1 allows more than one association between two classes, a same name for associations between two different pairs of classes, and a same attribute name for attributes of different classes. In Figure 1, we give two class diagrams *Bank1* and *Bank2* for two possible banking systems. We only show either an association or its inverse, but not both in a diagram.

Fig. 1. An example of class diagram

2.2 Semantics of class conceptual diagrams

A class diagram specifies a family of types to represent the *data domain* of an application. Each class name C in a conceptual class diagram Δ is associated with a *class* of *objects* in the application domain. Let us assume a set \mathcal{O} of objects in the universe. Therefore, Δ assigns each $C \in \mathcal{C}$ a non-empty subset \mathbf{C} of \mathcal{O} . We call \mathbf{C} the *object type* of C [AC96]. The generalization relation $\triangleleft\text{---}$ in a class diagram defines a sub-superclass $<$: relation between the classes contained in Δ :

SUBT-1: $\mathbf{C} <: \mathcal{O}$ for each $C \in \mathcal{C}$.

SUBT-2: $\mathbf{C} <: \mathbf{C}$ for each $C \in \mathbf{CName}$

SUBT-3: $C_1 <: C_2$ if $C_2 \leftarrow C_1$ is contained in Δ

SUBT-4: $C_1 <: C_3$ if $C_1 <: C_2$ and $C_2 <: C_3$.

The meaning of $C_1 <: C_2$ is defined as the set inclusion $C_1 \subseteq C_2$.

We require that a class diagram satisfies the following conditions.

1. The generalization is acyclic:

$$W_1(\Delta) \stackrel{def}{=} C_1 \leftarrow C_2 \Rightarrow C_1 \not\equiv C_2$$

2. The attribute names of a class are all distinct:

$$W_2(\Delta) \stackrel{def}{=} \forall C \in \mathbf{CName} \bullet dist(\pi_1(attV(C)))$$

where $\pi_1(attV(C))$ is the list of attribute names of C , and $dist$ is true if all these names are distinct.

3. An attribute name assigned to a class C_2 should not be assigned to its subclass C_1 :

$$W_3(\Delta) \stackrel{def}{=} \left(\begin{array}{l} C_1 <: C_2 \\ \wedge C_1 \not\equiv C_2 \end{array} \right) \Rightarrow \pi_1(attV(C_1)) \cap \pi_1(attV(C_2)) = \emptyset$$

4. Similarly, any association name assigned to a class C_2 should not be assigned to its subclasses:

$$W_4(\Delta) \stackrel{def}{=} \left(\begin{array}{l} C_1 <: C_2 \\ \wedge C_1 \not\equiv C_2 \end{array} \right) \Rightarrow \left(\forall C \in \mathbf{CName} \bullet (AssN(C_1, C) \cap AssN(C_2, C) = \emptyset) \right)$$

5. Different associations between the same pair of classes should have different names:
for any $C_1, C_2 \in \mathcal{C}$ and $A_1, A_2 \in \mathbf{ANmae}$:

$$W_5(\Delta) \stackrel{def}{=} \left(\begin{array}{l} (Ass(C_1)(A_1) = \langle M_{11}, M_{12}, C_2 \rangle) \\ \wedge (Ass(C_1)(A_2) = \langle M_{21}, M_{22}, C_2 \rangle) \end{array} \right) \Rightarrow A_1 \equiv A_2$$

A class diagram Δ is well-formed if it satisfies

$$W(\Delta) \stackrel{def}{=} W_1(\Delta) \wedge W_2(\Delta) \wedge W_3(\Delta) \wedge W_4(\Delta) \wedge W_5(\Delta)$$

A class diagram Δ also identifies the following sets of variables that use cases operate on.

1. $\mathbf{CVar} \stackrel{def}{=} \{C : \mathbb{P}\mathbf{C} \mid C \in \mathcal{C}\}$ in which each C records the current set of objects of class \mathbf{C} existing in the system.
2. $\mathbf{AVar} \stackrel{def}{=} \{A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2) \mid A : (C_1, M_1, M_2, C_2)\}$ in which each A records the links between objects existing in the system.

We call variables in $\mathbf{CVar} \cup \mathbf{AVar}$ variables in Δ too.

2.3 Object diagrams as system states

By introducing \mathbf{C} , we have made an important distinction between classes and types. We denote by \mathbf{C} the type of class C . With this distinction, we can avoid the confusion of using C as both a type and a variable.

In UML, an object diagram of a class diagram Δ consists of some objects and links between these objects. The objects have to be instances of classes in the class diagram, and the links have to be instances of associations in the class diagrams. In our formalization, we define an object diagram as a *state* of the variables in Δ .

Definition 2. (Object Diagram) Let $\Delta = \langle \mathcal{C}, Ass, Att, \leftarrow, \rangle$ be a conceptual class diagram. An *object diagram* σ is a *state* over the variables $\mathbf{V} \stackrel{def}{=} \mathbf{CVar} \cup \mathbf{AVar}$, that is a mapping from variables in $\mathbf{CVar} \cup \mathbf{AVar}$ to values of their types:

- For each $C \in \mathbf{CVar}$, the value $\sigma[C]$ of C in state σ is a subset of \mathbf{C} .
- For each $A : (C_1, M_1, M_2, C_2) \in \mathbf{AVar}$, the value $\sigma[A]$ of A in state σ is a subset of $\mathbf{C}_1 \times \mathbf{C}_2$.
- For each $C \in \mathbf{CVar}$, each $a : \mathbf{T} \in att(C)$, and each $o \in \sigma[C]$, $o.a$ is a variable too and its value $\sigma[o.a]$ in state σ is taken from \mathbf{T} .

Let \mathbf{Att} is a state variable that take values of a set of typed variables the form

$$\{o.a_1 : \mathbf{T}_1, \dots, a.x_n : \mathbf{T}_n\}$$

Its value $\sigma[\mathbf{Att}]$ in a state σ is

$$\{o.a : \mathbf{T} \mid \exists C \in \mathcal{C}. (o \in \sigma[C] \wedge (a : \mathbf{T}) \in Att(C))\}$$

Unlike \mathbf{CVar} and \mathbf{AVar} that are fixed for a class diagram, \mathbf{Att} changes during the operation of the system that the class diagram models.

An example of an object diagram of *Bank1* in Figure 1 is given in Figure 2.

Fig. 2. An example of an object model

2.4 State assertions

An application may require some property always hold during the execution of the system. For example, Figure 1 shows that a customer is allowed to have up to 3 accounts in a “big bank” modeled by diagram *Bank1*, while a customer has one and only one account in a “small” bank modeled by *Bank2*. In general, we can use predicate over $\mathbf{V} \cup \mathbf{Att}$ to specify a state constraint.

For an association $A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2)$ and an objects $o_1 \in C_1$, let

$$A(o) \stackrel{def}{=} \{o_2 \mid o_2 \in C_2 \wedge (o_1, o_2) \in A\}$$

Apart from the syntactical constraint expressed in Definition 1 and the well-formed condition $W(\Delta)$ for a class diagram Δ , the following *state invariants* must be met by any valid state of Δ : for any classes C, C_1, C_2 , and any association $A : (C_1, M_1, M_2, C_2)$ in Δ ,

$$\begin{aligned} \theta_1 &\stackrel{def}{=} \forall A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2) \in \mathbf{AVar}, o : \mathbf{C}_1, o_2 : \mathbf{C}_2 \bullet \\ &\quad ((o_1, o_2) \in A \Rightarrow (o_1 \in C_1 \wedge o_2 \in C_2)) \\ \theta_2 &\stackrel{def}{=} \forall A \in \mathbf{AVar}, o_1 \in C_1, o_2 \in C_2 \bullet (|A(o_1)| \in M_2 \wedge |A^{-1}(o_2)| \in M_1) \\ \theta_3 &\stackrel{def}{=} \forall A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2) \in \mathbf{AVar}, o_1 : \mathbf{C}_1, o_2 \in \mathbf{C}_2 \bullet \\ &\quad ((c_1, c_2) \in A \Leftrightarrow (c_2, c_1) \in A^{-1}) \\ \theta_4 &\stackrel{def}{=} \mathbf{C}_1 <: \mathbf{C}_2 \Rightarrow C_1 \subseteq C_2 \end{aligned}$$

where M_1 and M_2 are the cardinalities of C_1 and C_2 in A .

Property θ_1 ensures that associations only link currently existing objects in a state, and all links of a object must be removed as well if this object is removed from the system; θ_2 characterizes the cardinalities of the roles in an association; θ_3 asserts that an association is of no direction; and θ_4 describes the inheritance. A *valid object diagram* of a conceptual class diagram Δ is state σ of Δ that satisfies

$$\theta \stackrel{def}{=} \theta_1 \wedge \theta_2 \wedge \theta_3 \wedge \theta_4$$

Definition 3. (Semantics of a Conceptual Diagram) The semantics of a conceptual class diagram Δ is the set of all its valid object diagrams, denoted by $[[\Delta]]$

The object diagram in Figure 2, is a state of *Bank1* but not a state of *Bank2* in Figure 1.

The constraint θ of Δ is enforced by the diagram itself. However, only classes, associations, and their cardinalities are not enough to express all constraints that the application requires. For example, the diagram in Figure 3 does not describe the property that a copy being held for a reservation must be a copy of the publication reserved for the reservation. This property cannot be represented by drawing elements. In UML, it can only be represented by a *comment* in text. In our model, this constraint can be described as the state assertion:

$$\begin{aligned} &\forall c \in Copy, r \in Reservation, p \in Publication \bullet \\ &IsHeldFor(c, r) \wedge IsOn(r, p) \Rightarrow Has(p, r) \end{aligned}$$

where we used the convention $R(a, b)$ for $\langle a, b \rangle \in R$ for a relation R . This constraint can be written in terms of the algebra of relations

$$IsHeldFor \circ IsOn \subseteq Has^{-1}$$

where \circ is the *composition* operation of relations.

Fig. 3. A class diagram for a library system

2.5 Conceptual models

Definition 4. (Conceptual Model) A conceptual model $CM = \langle \Delta, Inv \rangle$ where Δ is a conceptual class diagram and Inv is a state constraint over Δ .

A state property ψ of a conceptual model can be reasoned about by showing that $\theta \wedge Inv \Rightarrow \psi$, meaning that ψ can be proven from θ and Inv in the relational calculus. We denote by $CM \models \psi$ that CM satisfies ψ . This also allows us to define transformations between conceptual diagrams that preserves a state constraint.

2.6 Associative classes

UML allows *associative classes*. An example of this kind of classes is shown in Diagram (a) of Figure 4. The Class *JobContract* is about the association *Employs*. It can be modeled by a decomposition of the association into two associations as shown in Diagram (b) of Figure 4. Notice the cardinalities of *Company* in the association *Has* and *People* in the association *IsFor* are both $\{1\}$. However, we further need to relate the association *Employs* with the two newly introduced associations by the constraint

$$\text{Has} \circ \text{Is-for} = \text{Employs}$$

Fig. 4. Representation of Associative Class

In general, an association $A : (C_1, M_1, M_2, C_2)$ in UML can be decomposed by adding a class *AClass* and two associations, $A_1 : (C_1, \{1\}, M_2, AClass)$ and $A_2 : (AClass, M_1, \{1\}, C_2)$ such that $A_1 \circ A_2 = A$. Such a decomposition also changes the many-to-many association into one-to-many associations that are much easier to realize in a design. This treatment of associative classes can be also used in applications where some classes are needed to relate any number of classes.

3 Use-Case Model

Given a conceptual model CM , an object diagram represents a snapshot of the system at a moment of time. The execution of an *atomic use case* will change the system from one state into another by creating new objects, deleting old objects; forming or breaking links between objects; or modifying attributes of objects. Such an atomic use case is called a *system operation* [Lar98,Liu00]. However, the system only executes an atomic use when it is called by an actor. Therefore, we should also model actions performed by the actors and the interaction between the actors and the system.

3.1 System operations

To define an operation, we use the notion of *design* in [HH98]. A design is a predicate that relates the initial values of state variables to their final values. It takes the form of $p(x) \vdash R(x, x')$

$$(p(x) \vdash R(x, x')) \stackrel{def}{=} ok \wedge p(x) \Rightarrow R(x, x') \wedge ok'$$

where x represents the value of x in the initial state and x' represents the value of x in the final state. Such as design asserts that the precondition p must be true before the operation starts, and the post-condition R holds when the operation terminates.

However, a particular operation only changes part of the system variables and keep the rest unchanged. Thus, its design is always *framed* with the set of the variables it changes:

$$V : (p \vdash R) \stackrel{def}{=} p \Rightarrow R \wedge (\underline{w}' = \underline{w})$$

where \underline{w} contains all variables but those in V .

A system operation operates on the following variables.

1. It may change an class variable $C : \mathbb{PC}$ by creating a new object or deleting an existing object. Therefore, an operation may access variables in **CVar**.
2. It may change an association variable $A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2)$ by forming a new link or breaking an existing link between two objects. So variables in **AVar** may be accessed by an operation.
3. The value of **Att** will be changed when a new object is created or an old one is deleted.
4. An attribute $o.x$ in the current value of **Att** may be modified or read, e.g. changing or outputting the name of an existing person.
5. A system operation will be executed when some input value parameters are provided and will output some results to some variables. Therefore, an operation is specified as a procedure with a list $\underline{x} : \underline{\mathbf{T}}_1$ of *formal value parameters* and a list $\underline{y} : \underline{\mathbf{T}}_2$, of *formal result parameters*. Such a procedure can only be executed when it is called by an *actor object* that provides the *actual* value and result parameters. We use **val** to denotes the set of all actual value parameters and **res** the set of all actual result parameters for the calls to the system operations in a system. These two set of parameters are also variables that are accesses by the execution of the system operations. We require that these variables does not introduce new classes types.
6. The actors decide the protocol in which they interact with the system. Local control variables are need for sequencing, choices and iterations. A control variable is of a simple data type.

We have **V** to be the set that contains the variables identified in items 1-4. Let **U** be the set of variables that contain the actual parameters and control variables identified in items 5 and 6, and **P** to be the set of formal parameters required for the specification

of the system operations. We define $\Gamma \stackrel{def}{=} \mathbf{V} \cup \mathbf{U}$, and $\Omega \stackrel{def}{=} \mathbf{V} \cup \mathbf{P}$. Now an system operation is defined in the form

$$op[\underline{x} : \mathbf{T}_1; \underline{y} : \mathbf{T}_2] :: \mathbf{Pre} : Pre; \mathbf{Post} : Post$$

where op is the name of the operation, $\underline{x} : \mathbf{T}_1$ and $\underline{y} : \mathbf{T}_2$ the formal value and result parameters, Pre is a predicate over Ω that defines the precondition, $Post$ is a predicate over variables in Ω and their primed versions Ω' that defines the post condition.

The semantics of operation op is defined as a *guarded design*:

$$op \stackrel{def}{=} V : (Pre \vdash Post)$$

where V is the set of variables that can be modified by op .

3.2 Actors operations

A *primitive* operation by an actor in the environment is also a guarded design of the form:

$$op(\underline{e}, \underline{y}) \stackrel{def}{=} V : (\neg en(op) \wedge (true \vdash \underline{x}' = \underline{e} \wedge en(op)))$$

where \underline{x} is the list of value parameters of op , \underline{e} is a list expression well-typed according to the types $\underline{\mathbf{T}}_1$ of \underline{x} , and V contains all variables in Γ , but $en(op)$ and those in \underline{x} .

To write a protocol in which actors call system operations to carry out a full use case, operators on actors' operations can be used.

Consider two operations of actors $a_1 \stackrel{def}{=} p_1 \vdash R_1$ and $a_2 \stackrel{def}{=} p_2 \vdash R_2$, and Boolean expression b_1 and b_2 .

- Guarded action $b_1 \longrightarrow a_1 \stackrel{def}{=} b_1 \wedge (p_1 \vdash R_1)$. This action only takes place when b is true.
- Choice $b_1 \longrightarrow a_1 \square b_2 \longrightarrow a_2 \stackrel{def}{=} (b_1 \longrightarrow a_1) \vee (b_2 \longrightarrow a_2)$. This can be extended to choices among any number of actions.
- $(a_1; a_2) \stackrel{def}{=} (p_1 \vdash R_1 \wedge c') \vee c \longrightarrow (p_2 \vdash R_2 \wedge \neg c')$, where c is only used for the control and it is initially *false*. So $a_1; a_2$ ensures a_2 can only be executed after a_1 though in between other actions can be executed as well.

For example, we can specify system operations $FindUser(Id : \mathbf{N}, u : \mathbf{User})$, $FindCopy(Id_1 : \mathbf{N}, c : \mathbf{Copy})$, $Loan((u, c) : (\mathbf{User}, \mathbf{Copy}))$ for a library system. Actor librarian can carry out the use case $LendCopy(i_1, i_2)$ by the protocol

$$FindUser(i_1, u); (u \neq null) \longrightarrow FindCopy(i_2, c); (c \neq null) \longrightarrow Loan(u, c)$$

3.3 Constructing a system specification

Given a conceptual model $CM = (\Delta, Inv)$, a set of systems operations OP , a protocol Act in which the actors uses the system operations that is a set of actors actions. We define

- The set I of variables that the system operations and actors operations operate on.
- We assume an initial condition $Init$ which define the set of states from which the system can start to work.
- The set of all operation $P \stackrel{def}{=} OP \cup Act$.

The system is then specified by the transition system $\mathcal{S} = (\Gamma, Inv, Init, P)$.

The semantics of \mathcal{S} is defined to be all the possible execution sequences of the operations in P . Formally, an execution of \mathcal{S} is an infinite sequence of states, $\sigma_0, \sigma_1 \dots$, such that

- σ_0 satisfies $Init$.
- Each step (σ_i, σ_{i+1}) is a carried out by an operation in P , i.e., there is an operation op_i such that σ_i satisfies precondition of op_i and (σ_i, σ_{i+1}) satisfies the post condition of op_i .

The invariant properties Inv can be prove by showing that for each $op \in P$,

$$Pre \wedge Inv \wedge Post \Rightarrow Inv'$$

where pre and $Post$ are the precondition and post condition of op , and Inv' is the predicate obtained from Inv by replacing its variables with their primed versions.

3.4 Examples of use cases

This subsection gives some small examples of system operations (use cases). We use the two models $Bank_1$ and $Bank_2$ in Figure 2. Under $Bank_2$, we can specify an operation that allows a customer to withdraw a certain amount of money from his/her.

$$\begin{aligned} Withdraw_1[(c, b) : (\mathbf{Customer}, \mathbf{Real})] &\stackrel{def}{=} \\ Pre : c \in Customer & \\ Post : Holds(c).balance' = Holds(c).balance - b & \end{aligned}$$

Under model $Bank_1$ which allows customer to have no account or a number of accounts, the withdraw use case should then be defined as

$$\begin{aligned} Withdraw_2[(c, a, b) : \mathbf{Customer}, \mathbf{Account}, \mathbf{Real}] &\stackrel{def}{=} \\ Pre : c \in Customer \wedge a \in Account \wedge Holds(c, a) & \\ Post : a.balance' = a.balance - b & \end{aligned}$$

Model $Bank_1$ can support a “withdraw” operation that behaves different from the above one:

$$\begin{aligned} Withdraw_3[c : \mathbf{Customer}] &\stackrel{def}{=} \\ Pre : c \in Customer \wedge \exists a \in Account \bullet Holds(c, a) & \\ Post : Let a = choice(Holds(c)) in a.balance' = a.balance - b & \end{aligned}$$

In fact this withdraw use operation behaves the same under $Bank_1$ and $Bank_2$, and it behaves the same as $Withdraw_1$ under $Bank_2$. In fact,

$$Bank_2 \models c \in Customer \Leftrightarrow \exists a \in Account \bullet Holds(c, a)$$

We can see that $Bank_1$ supports the an operation for a customer to transfer money from one account to another owned by him or her, but $Bank_2$ cannot. $Bank_2$ also requires that when a customer is created, an account must be created for him or her too; and an existing customer cannot open another account under $Bank_2$. Therefore, $Bank_1$ supports more operations than $Bank_2$. Under $Bank_1$, the transfer use case can be written as:

$$\begin{aligned} & Transfer[(c, from, to, b) : \mathbf{Customer}, \mathbf{Account}, \mathbf{Account}, \mathbf{Real}] \stackrel{def}{=} \\ & Pre : c \in Customer \wedge Holds(c, from) \wedge Holds(c, to) \\ & Post : (from.balance' = from.balance - b) \wedge (to.balance' = to.balance + b) \end{aligned}$$

Figure 5 shows the effect of $Withdraw_1$ on a state of $Bank_2$, and Figure 6 illustrates the effect of $Transfer$ use case on a state of $Bank_1$.

Fig. 5. Effect of Withdraw Use Case

Fig. 6. Effect of Transfer Use Case

4 Conclusion & Discussion

We have provide a model to formally combine a conceptual models and a use-cases model of UML to form a system specification. The model is the well-know notation of transition systems [MP81,Bac88] for general reactive systems. This is well justified as an object-orient system is in nature a concurrent and reactive system.

The advantage of using a well-establish model is that we do not have to develop or study new semantics and tools for verification. Methods and tool for specification and verification of transition system are well-established, e.g. [Lam91,MP91]. Furthermore, this model is already extended to deal with real-time and fault-tolerance [HMP91,AL92,LJ99]. The same methods can be used to deal with real-times requirements on use cases. Also, the model of transition systems is isomorphic to that of the statecharts which is a part of UML.

The main difference between our work and that in [pG99,EKHG01,Egy01] is that we study formal semantic relationships between different models of UML, rather than only formalization of individual diagrams. The paper [HR00] also treats a class as a set of objects and an association as a relation between objects. However, it does not consider use cases. Our work also shares some common ideas with [BPP99] in the

treatment of use cases. However, we believe our model is simpler and addresses the relationships among different models clearer.

In our related work [LLH01,LLG01], we used case studies to demonstrate that the formalization supports building up a model step by step. In [LLHC02], a specification language is developed with which we can write a specification as a Java-like program.

We have developed a model for requirement analysis in this paper, a specification language in [LLHC02] and a model for object-oriented programming in [HLL01]. Further work is needed to close the gap between requirement analysis and programming by providing a method to transform a use-case model to a design model. Progress in this direction is made in [HLL02].

References

- [AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
- [AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. In J.W. de Bakker, C. Huizing, W.P. de Rover, and G. Rozenberg, editors, *Real-Time: Theory in Practice, Lecture Notes in Computer Science 600*. Springer-Verlag, The Netherlands, 1992.
- [Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
- [BPP99] R.J.R. Back, L. Petre, and I.P. Paltor. Formalizing UML use cases in the refinement calculus. Technical Report 279, Turku Centre for Computer Science, Turku, Finland, May 1999.
- [BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.
- [DW98] D. D’Souza and A.C. Wills. *Objects, Components and Framework with UML: The Catalysis Approach*. Addison-Wesley, 1998.
- [Egy01] A. Egyed. Scalable consistency checking between diagrams: The Viewintegra approach. In *Proc. of the 16th IEEE International Conference on Automated Software Engineering*, San Diego, USA, 2001.
- [EKHG01] G. Engels, J.M. Kuster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *The Proc. of International Conference on Foundation of Software Engineering, FSE-10*, Austria, 2001.
- [HH98] C.A.R. Hoare and J. He. *Unifying theories of programming*. Prentice-Hall International, 1998.
- [HLL01] J. He, Z. Liu, and X. Li. A relational model for object-oriented programming. Technical Report UNU/IIST Report No 231, UNU/IIST, P.O. Box 3058, Macau, March 2001.
- [HLL02] J. He, Z. Liu, and X. Li. Towards a refinement calculus for object-oriented systems. To appear at ICCI02 as a Kenote Talk, August 19-20, 2002, Alberta, Canada, 2002.
- [HMP91] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 8th ACM Annual Symposium on Principles of Programming Languages*, pages 269–276, U.S.A., 1991. ACM Press.
- [Hoa96] C.A.R. Hoare. The role of formal techniques: past, current and future or how did software get so reliable without proof? In *Proc. of the 18th International Conference on Software Engineering*, pages 233–235. Berlin - Heidelberg - New York, Springer, 1996.

- [HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, September 2000.
- [JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [Ken97] S. Kent. Constraint diagrams: Visualising invariants in object-oriented models. In *OOPSLA97*. ACM Press, 1997.
- [Lam91] L. Lamport. A temporal logic of actions. Technical Report 79, Digital SRC, California, 1991.
- [Lar98] C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 1998.
- [LHL01] Z. Liu, J. He, and X. Li. Toward a formal use of UML for software requirement analysis. In Hamid Arabnia, editor, *The Proceedings of PDPTA'2001 International Conference*, pages 27–33, Las Vegas, USA, June 2001. CSREA.
- [Liu00] Z. Liu. Lecture notes for mc206 software engineering and system development. 2000.
- [LJ99] Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing and scheduling. *ACM Transactions on Languages and Systems*, 21(1):46–89, 1999.
- [LLG01] X. Li, Z. Liu, and Z. Guo. Formal object-oriented analysis and design of an online ticketing system. In *The proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC2001)*, pages 259–266, Macau, P.R. China, December 2001. IEEE Computer Society.
- [LLH01] X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *COMPSAC01*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.
- [LLHC02] Z. Liu, X. Li, J. He, and Y. Chen. A relational model for object-oriented analysis. Submitted for publication, 2002.
- [MP81] Z. Manna and A. Pnueli. The temporal framework for concurrent programs. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–274. Academic Press, 1981.
- [MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
- [pG99] The pUML Group. The precise UML web site: /<http://www.cs.york.ac.uk/puml>. 1999.
- [RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.