Formalizing the Use of UML in Requirement Analysis*

Zhiming Liu†
Department of Computer Science
The University of Leicester
Leicester, LE1 7RH, England
email: Z.Liu@mcs.le.ac.uk

Jifeng He
International Institute for Software Technology
The United Nations University
P.O. Box 3058, Macau, SAR, P.R. China
email: jifeng@iist.unu.edu

Xiaoshan Li
Faculty of Science and Technology
The University of Macau
P.O. Box 300, Macau, SAR, P.R. China
email: fstxsl@umac.mo

Abstract

The Unified Modelling Language (UML) is now widely used for modelling a software at different stages: requirement analysis, design and implementation, during the system development. This work attempts to develop a method to support the *formal use* of UML in object-oriented software development. The method will include formal definitions of the modelling units in UML which can be used to relate the different UML models used at different stages during software development process. It intends to support step-wised refinement and component-based development in building models. As a starting point, this paper deals with the formalization of the UML models used in the requirement analysis, i.e. *use-case model* and *conceptual models*.

Keywords: *Conceptual Model, Use-Case Models, Object-Orientation, Refinement, UML.*

1 Introduction

Object-orientation has now become a popular approach in software industries [Mey97, Lar98, Fow97]. The Unified Modelling Language (UML) [BRJ99, RJB99, JBR99] is now widely accepted as the modelling language in OO software development [Lar98, Fow97, DW98]. Driven by this trend, universities have now started to teach OO programming and OO software engineering [Liu00, PS99]. Computer scientists is now also intensifying the research about the understanding and use of OO methods and UML [AC96, Ken97, BKS98, pG99, CN00]. One of the main advantages of UML is that different modelling notations are used at different stages to represent the system at a proper level of abstraction and to describe both of static structure and dynamic behaviour of the system.

*This works is partly supported by the British EPSRC Grant GR/M89447.

†This work is carried out when this author is on a study leave and working as a Visiting Scientist at the International Institute for Software Technology, the United Nation University (UNU/IIST), Macau, SAR, P.R. China.

The main models to be produced in the requirement analysis stage are the *conceptual model* and the *use-case model* [Lar98, JBR99, DW98]. A main part of a conceptual model is a class diagram that represents the application domain concepts as *classes*, and their relationships as *associations*. This diagram determines the possible *objects* and the possible relationships between these objects. Another part of a conceptual model is a set of assertions, called *comments* in UML, about the objects and their relations that cannot described by the diagram. We will describe these comments as general *state constraints*. The requirement analysis is not usually concerned very much about what an object will do or how it will behave. Therefore, a conceptual model is mainly used as a *static model* without showing dynamic semantic aspects.

A 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 of them describe a pattern of interactions between some users, called *actors* of the use case, and the system treated as a *black box*. The *consistency* between the use-case model and the conceptual model is concerned only with whether the conceptual model contains enough objects and associations to realize the use cases contained in the use-case model. However, this consistency can not yet be discussed precisely enough[Lar98, Liu00].

The *system design* is to produce, based on the conceptual model and the use-case model, the interaction diagrams that represent how the classes of objects to interact and collaborate with each other to realize the uses cases. The *correctness* of the interaction diagrams are reasoned against the use case-model to prove whether the interactions between objects do realize the use cases. Again there is no enough research to allow a systematic and precise reasoning or verification of such a correctness.

The interaction diagrams will derive a *design class diagram* which includes information about both classes of objects, visibility of classes and their methods. The design class diagram should be a *refinement* of the conceptual model, but there is not yet a formal definition for this relation.

Program code can be then easily produced from the interaction diagrams and the design class diagram. For describing the behaviour of objects, *state machines* (or *statecharts*) are used as well. Research needs to be done to relate the design model with the semantics of a programming language [CN00].

OO techniques and UML claim to support *component-based* development, and intuitively it looks so. There is a lack of research how the ideas in OO and component-based approaches be used more precisely and systematically so that they can be used in safety-critical applications. We need to provide formal definitions for components and how they can be composed.

The major problem of using UML is also caused by its main advantage that allows the use of different models. When there is not yet a well established semantics for the whole language it is impossible to check or reason about the consistency and their relationship among the different models. Currently, most of the research on formalizing UML focus on parts of UML, such as formal semantics of the class diagrams and formal definitions for statecharts [Ken97, pG99]. For UML is to be precisely used in a software development process, more research is needed on the formal semantics of the whole language and relationships among the different models used in UML.

This work is towards a formalization of UML. The overall aim is to support formal use of UML in OO system development processes and development of tools for consistency checking. We will develop a method that allows system design by component-based and stepwise refinement. The method is expected to be usable within an incremental and iterative development process [Lar98, JBR99]. We hope this will on the one hand to change the situation today's situation that OO software development in practice is usually done in a non-scientific manner [Hoa96] based on pragmatic and heuristics. One the other hand, with incorporation of the incremental an iterative development process, we hope to improve the use of formal methods in the development of large scale system. The semantic definitions will be given for the five main kinds of models which are use-case models, conceptual class models, object-diagrams, interaction diagrams, and design class diagrams. Generally, these five kinds of diagrams are enough for specifying the requirements and design of a system. The other kinds of diagrams: activity diagrams, deployment diagrams and component diagrams, are not difficult to deal with. The activity diagrams describe the system components co-operate concurrently to achieve the system large activities. Component diagrams and deployment diagrams give the system construction management, and thus their semantic is not concerned relevant to system design.

Most work on formal methods for OO development uses a bottom-up approach [Jon94, LW95, Jon96, AC96, CN00]. In such an approach, a formal semantics is defined for a low level programming language like specification language. This language is so expressive that even implementation details can be described, though it can also be used at relatively hight level of abstractions. One of the advantages of this approach is that most of the semantic issues are solved once for all. However, the main drawback of this approach is that one has to study the very complicated semantics for such a low level language to be confident to use the formal method. Also it is not trivial to extract the right subset of the notation that is proper for higher level specifications as the semantics of these languages are very complicated [Jon94, Jon96, LW95, CN00]. The main idea used in UML modelling is the opposite that simpler notation and semantics are used at higher level specifications. Further details and structures are gradually introduced when the development process progress. This work attempts to follow this evolutionary approach to develop a semantics for UML. Simplicity and ease of understanding and use are the main aspects that we want to achieve. The framework will be based on the set theory and the notion of *pre* and *post* conditions. As a starting point of our ongoing research, this paper only deals with the formalization of the UML models that are used in the requirement analysis, i.e. the use-case model, the conceptual class diagrams, and their relationship.

In the rest of this paper, we shall define a syntax and a semantics for a conceptual model in Section 2, and then define in Section 2.5 an *object model* as an instance of a conceptual class model. Section 3 defines the 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. A use case is defined in the context of a conceptual model. 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 model which captures the classes and their associations in terms of the physical *concepts*, the *instances* (or *objects*) defined by the concepts, and the relationships between these concepts or their instances. In UML, a concept is denoted by a *class* with a given name, and each instance of a concept is called an *object* of the corresponding class. This model intends to describe the structure of the application domain rather than that of the software. No all of the physical concepts in this model will be realized as software classes later on, though many of them will.

We must understand that at the requirement level, a class has a very simple semantics which denote a set of objects only without the need of knowing how the objects will behave, and thus has no methods. In general, a class may also have *attributes* to represent some information about the objects of the class themselves, rather than to be used to related classes. There are two approaches to deal with attributes at this level. The first is to introduce *types of pure data values* 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. This paper adopts the second approach.

The concepts, their objects and the associations between them are identified by understanding the application domain. The creation of such a conceptual model reply on discussions with the domain experts, and with the end users of the system about their *use cases*. However, the effect or the semantics of a use case can only be defined in the context of a conceptual model in terms how an operation or event in a use case will create or delete an object, how to form or how to break an association between two objects.

2.1 Conceptual diagram

To define the syntax of conceptual diagrams, we introduce two set of names $CName$ and $AName$ which are names used to denote classes and associations. For each name, $A \in AName$, we assume there is a unique name $A^{-1} \in AName$, which is call the *inverse* of A. We define $(A^{-1})^{-1} = A$. For a set $\mathcal{A}^{-1} \subseteq ANmae$, we denote $\mathcal{A}^{-1}$ to be the set of inverses of the names in $\mathcal{A}$. Let **N** denote the set of all the natural numbers, and $\mathbb{P}$**N** be the power of **N**.

Definition 1 (Conceptual Diagram) A conceptual diagram is a tuple: $D = \langle \mathcal{C}, \mathcal{A}, \lhd\!\!\!-\!\!\!-, \mathcal{R} \rangle$, where

- $\mathcal{C}$ is a nonempty finite subset of $CName$, called the *classes* or *concepts* of the diagram D.
- $\mathcal{A}$ is a finite subset of $ANames$, which are called the *associations* of D.
- $\mathcal{R} \in [\mathcal{A} \cup \mathcal{A}^{-1} \longmapsto \mathcal{C} \times \mathbb{P}\mathbf{N} \times \mathbb{P}\mathbf{N} \times \mathcal{C}]$ is a function. For $\mathcal{R}(A) =< C_1, M_1, M_2, C_2 >$, we say A is an association between C_1 and C_2, and we denote this fact by $A :< C_1^{M_L}, C_2^{M_2} >$. M_1 and M_2 are the *multiplicities* of C_1 and C_2 respectively in this association. If $\mathcal{R}(A) =< C_1, M_1, M_2, C_2 >$, then $\mathcal{R}(A^{-1}) =< C_2, M_2, M_1, C_1 >$.
- $\lhd\!\!\!-\!\!\!- \subseteq \mathcal{C} \times \mathcal{C}$ is the generalization relation between classes. We use $C_1 \lhd\!\!\!-\!\!\!- C_2$ to denote that C_1 is a *generalization* or a *superclass* of C_2, and C_2 is a *specialization* or *subclass* of C_1. We require that the generalization is acyclic, i.e. there is not a sequence of classes $C_1, \ldots, C_n$ such that $C_i \lhd\!\!\!-\!\!\!- C_{i+1}$, for $i = 1, \ldots, n$, and $C_n \lhd\!\!\!-\!\!\!- C_1$. We also write $C_2 -\!\!\!-\!\!\!\rhd C_1$ for $C_1 \lhd\!\!\!-\!\!\!- C_2$.

For an association $A :< C_1^{M_1}, C_2^{M_2} >$, we define $lft(A) \triangleq C_1$ and $rht(A) \triangleq C_2$ as the *roles* of the association; $ml(A) \triangleq M_1$ and $mr(A) \triangleq M_2$ as the *multiplicities* of the two roles C_1 and C_2 respectively. The inverse A^{-1} of the an association A is to represent that the direction of navigation from one class to it related class by the association A^{-1} is the opposite to that of A.

Given two conceptual diagrams $D_i = \langle \mathcal{C}_i, \mathcal{A}_i, \lhd\!\!\!-\!\!\!-_i, \mathcal{R}_i \rangle$ where $i = 1, 2$:

$$D_1 \cup D_2 \triangleq \langle \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \lhd\!\!\!-\!\!\!-_1 \cup \lhd\!\!\!-\!\!\!-_2, \mathcal{R}_1 \cup \mathcal{R}_2 \rangle$$

We now define a notion of *well-declaredness* for a class $C \in CName$ or an associations $A : \langle C_1^{M_1}, C_2^{M_2} \rangle$ is in a conceptual diagram D, denoted by $WD_D(C)$ or $WD_D(A : \langle C_1^{M_1}, C_2^{M_2} \rangle)$ (or $WD_D(A)$ for short) respectively.

Assume some *data types* that are the types of the so called *pure-data values* in UML [BRJ99] with some built-in functions and primitive predicates. Examples of primitive types include types of natural numbers **N**, integers **Int**, booleans, **Bool** characters **char**, and strings. Each such a data type is a type in our system.

1. if T is a data type, $WD_D(T)$;
2. if $C \in \mathcal{C}$, then $WD_D(C)$;
3. $A \in AName$, $WD_D(A : \langle C_1^{M_1}, C_2^{M_2} \rangle)$ if $\mathcal{R}(A) = A : \langle C_1^{M_1}, C_2^{M_m} \rangle$;
4. $WD_D(\overline{N})$, if $WD_D(N)$ for each element $N \in \overline{N}$ that is a list of classes, associations and generalizations;
5. $WD_D(C_1 \lhd\!\!\!-\!\!\!- C_2)$ if $WD_D(C_1, C_2)$ and $(C_1, C_2) \in \lhd\!\!\!-\!\!\!-$;

6. if $A \in AName$, $WD_D(A)$ and $\mathit{lft}(A) \lhd\!\!\!\longleftarrow C$, then $WD_D(A : \langle C^{ml(A)}, \mathit{rht}(A)^{mr(A)} \rangle)$;
7. $A \in AName$ and $WD_D(A, \mathit{rht}(A) \lhd\!\!\!\longleftarrow C)$, then $WD_D(A : \langle \mathit{lft}(A)^{ml(A)}, C^{mr(A)} \rangle)$.

In UML drawing, an association $A :< C_1^{M_1}, C_2^{M_2} >$ can be depicted by the diagram in Figure 1. We only show either an association or its inverse, but not both in a diagram.

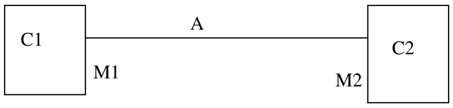

Figure 1: UML Association

Definition 1 allows more than one association between two classes. This is important because there are application in which two classes are related by two different associations. For example, in an application of air traffic control application, there may be two useful associations "Flies-from" and "Flies-to" between the classes Flight and Airport (see Figure 2).

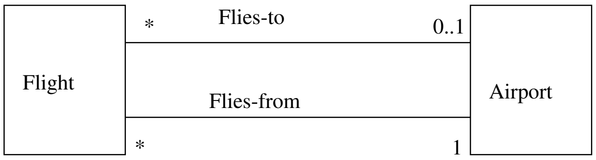

Figure 2: UML Multi-associations

In Figure 2, the star "*" represents whole set of natural numbers to mean that there may be 0 or any number of flights fly to or from an airport. The two associations differ because they related different flights with different airports. Also, a flight must take off from one airport, but it may never arrive at any airport (landing in a sea or crashing on a mountain). We shall use the UML conventions [BRJ99] to represent the multiplicities of an association. Figure 3 is a conceptual diagram, denoted by D_1, of a library application used in [Ken97].

2.2 Semantics of conceptual diagrams

To define semantics for a conceptual model, we need a very simple type system:

- Any class name $C \in CName$, $\mathit{InstanceTypeof}(C)$ is a type.
- Each data type is a type.
- If T_1, T_2 are types, then $T_1 \times T_2$ is a type, called the type of the pairs of elements in the two types; and $T_1 \longmapsto T_2$ is a type, called the type of functions from T_1 to T_2.
- If T is a type, $\mathbb{P}T$ is a type, called the types of the power set of P.

We denote by $\mathcal{T}$ the set of all types constructed in the above type system. With these types, we allow ordinary operations on sets, such as $\cup$ and $\cap$; on relations, such as *composition*; and on functions such as *function composition*. We define a special function $\sharp : \mathbb{P}T \longmapsto \mathbf{N} \cup \{\infty\}$, which returns the number of

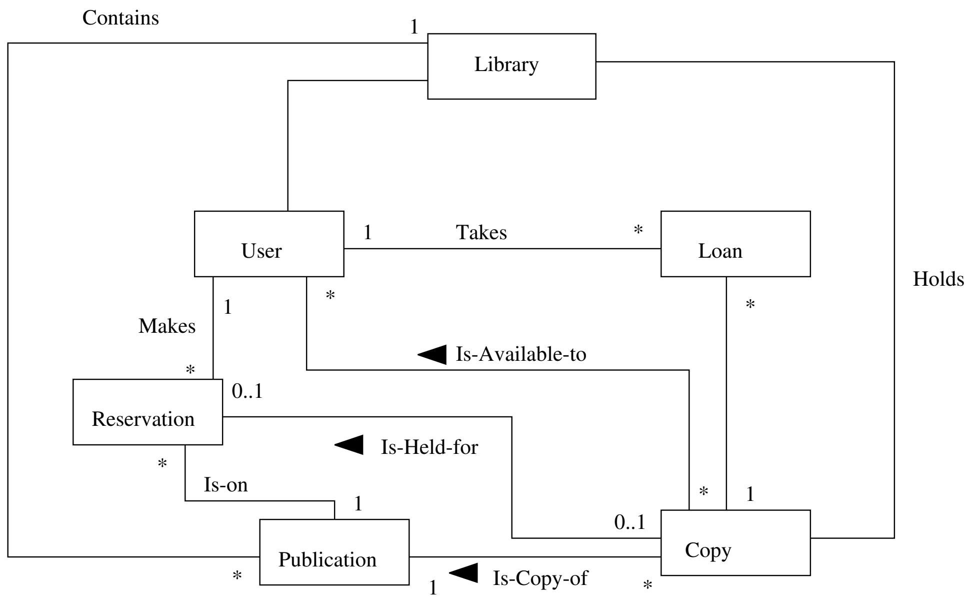

Figure 3: A Conceptual Model of a Library Application

elements of a set. For a relation $A \in \mathbb{P}(T_1 \times T_2)$, we use $A(t_1)$ for $t_1 : T_1$ (or $t_2 : T_2$) to the set of elements in T_2 (respectively in T_2) that are related to t_1 (respectively to t_2) by A. So $A(c_1)$ is of type $\mathbb{P}T_2$ and $A(t_2)$ is of type $\mathbb{P}T_1$.

By introducing $InstanceTypeof(C)$, we have made an important distinction between classes and types [AC96]. We indicate by $InstanceTypeof(C)$ the type of class C. With is distinction, we can avoid the confusion of using C as both a type and a variable.

We assume a set V of *variables*, for such a variable v and a type T, we use $v : T$ to denote that variable v is of type T. Using these variables and the operations and functions of the type system, we can define *well-typed* terms, and we denote by $WT(t)$ that t is well typed. For a class C, we shall use $c : C$ as a syntactic shorthand for $c : InstanceTypeof(C)$.

We use the following conventions.

- For each class $C \in CName$ declared in a conceptual model, C will be treated as a variable of type $\mathbb{P}InstanceTypeof(C)$ that records the set of objects of class C in any given system state.
- For each $A \in AName$ declared in a conceptual model, A is a variable of type

$$\mathbb{P}(InstanceTypeof(lft(A)) \times InstanceTypeof(rht(A))$$

The semantics of each *data type* and each of the type operation take its classical meaning. The semantics of $InstanceTypeof(C)$ for each $C \in CName$ is a non-empty set of objects in $\mathcal{O}$. We will use a relational symbol $=$, which represents equality tests at all types. Given a conceptual diagram D, $t_1 = t_2$ is *well-formed in D* iff t_1 and t_2 are well-typed on the same type. For a diagram $D = \langle \mathcal{C}, \mathcal{A}, \Longleftarrow, \mathcal{R} \rangle$, we define an *object diagram* S which also called a *state* of D as an assignment of of the entities in D to values in types:

- $S : \mathcal{C} \longmapsto \mathbb{P}\mathcal{O}$ such that $S[C] \in \mathbb{P}InstanceTypeof(C)$;

- S assigns $\mathcal{R}(A)$ for each $A \in \mathcal{A}$ as relation $S[A] \in \mathbb{P}(S[\mathit{lft}(A)] \times S[\mathit{rht}(A)])$ such that

$$\forall c_1 \in S[\mathit{lft}(A)], c_2 \in S[\mathit{rht}(A)] \bullet \sharp S[A](c_2) \in \mathit{ml}(A) \wedge \sharp S[A](c_1) \in \mathit{mr}(A))$$

- the generalization relation is the superset relation: if $WD_D(C_1 \lhd\!\!\!- C_2)$, then $S[C_1] \supseteq S[C_2]$;

All states of a diagram D have to satisfy the following axioms, where $-\!\!\rhd^*$ is the reflexive and transitive closure of $-\!\!\rhd$: for all classes C, C_1, C_2, and associations A in D,

1. $\mathcal{I}_D^1 \triangleq \emptyset \subset \mathit{InstanceTypeof}(C) \wedge C \subseteq \mathit{InstanceTypeof}(C) \subset \mathcal{O}$
2. $\mathcal{I}_D^2 \triangleq (A \subseteq \mathit{lft}(A) \times \mathit{rht}(A) \wedge \forall c_1 : \mathit{lft}(A), c_2 : \mathit{rht}(A) \bullet \sharp \mathit{fst}(A)(c_2) \in \mathit{ml}(A) \wedge \sharp \mathit{snd}(A)(c_1) \in \mathit{mr}(A))$
3. $\mathcal{I}_D^3 \triangleq C_1 \lhd\!\!\!- C_2 \Rightarrow C_1 \supset C_2$
4. $\mathcal{I}_D^4 \triangleq C_1 \cap C_2 = \emptyset \vee \exists C : C_1 -\!\!\rhd^* C \wedge C_2 -\!\!\rhd^* C$
5. $\mathcal{I}_D^5 \triangleq (A : \langle C_1^{M_1}, C_2^{M_2} \rangle \Leftrightarrow A : \langle C_2^{M_2}, C_1^{M_1} \rangle) \wedge (\forall c_1 : C_1, c_2 : C_2 \bullet A(c_1, c_2) \Leftrightarrow A^{-1}(c_1, c_1))$
6. $\mathcal{I}_D^6 \triangleq A : \langle C_1^{M_1}, C_2^{M_2} \rangle \wedge C_1 \lhd\!\!\!- C \Rightarrow A : \langle C^{M_1}, C_2^{M_2} \rangle$

Property $\mathcal{I}_D^1$ ensures that a diagrams does not contain meaningless class names; property $\mathcal{I}_D^2$ characterizes the multiplicities of the roles in an association; property $\mathcal{I}_D^3$ says that a class cannot be a generalization of itself; property $\mathcal{I}_D^4$ says that only classes which are subclasses of a same classes may contain common objects; $\mathcal{I}_D^5$ asserts that an association is of no direction at the requirement analysis model; and $\mathcal{I}_D^6$ ensures that a subclass inherits the associations and thus attributes of its superclass. We denote the conjunction of the above six properties by $\mathcal{I}_D$ ($\mathcal{I}$ when D is known).

Note that UML does not have the restriction required by the semantics of a conceptual model that an association name cannot be used to for different association. However, without this condition, the formal semantics is difficult to define. That is why we require $\mathcal{I}_D^1$. UNL does not either have the restriction required by $\mathcal{I}_D^4$. However, if we require that an association name cannot be used for two different association but without the restriction enforced by $\mathcal{I}_D^4$, it will be difficult for us to describe the same relation between a common object in two classes with objects in a third class. Further, $\mathcal{I}_D^4$ helps to produce better models by using the abstraction analysis pattern [MMMNS00]. An example UML conceptual model of such a case is shown in Figure 4 which shows that an owner may be also a designer of a project. This can be better modelled modelled as either of the diagrams in Figure 5, which are proper diagrams according to our definition. We can of course make $\mathcal{I}_D^4$ even strong that all classes must be disjoint, and thus a specialization is a partition of a super class. However, we prefer this to be enforced by the modeller using state constraints.

Definition 2 (**Semantics of a Conceptual Diagram**) The semantics of a conceptual diagram is the set of all the states of D, denoted by Σ_D

2.3 State constraints

A conceptual model enforces constraints on what classes of objects, what associations between them can exist in the system, and how many objects in one class can be associated with an object in another. In general, assertions about the objects and associations can be written in set theory and first order predicate logic.

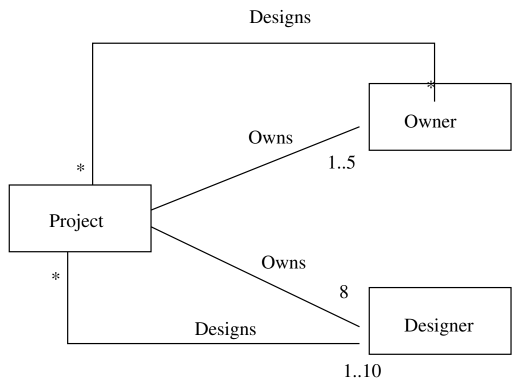

Figure 4: An Example of UML diagram with non-disjoint classes

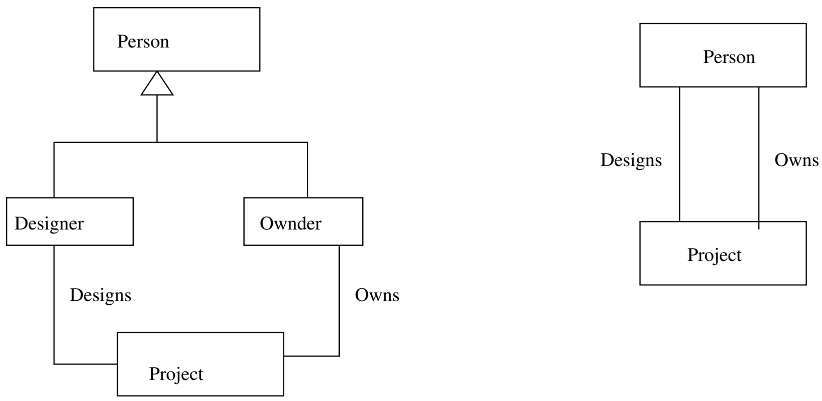

Figure 5: Better models for Figure 4

However, the library system shows that only classes, associations, and their multiplicities are not enough to express all the constraints that the application requires. In particular, multiplicity restrictions do not allow relationships between associations to be expressed. For example, we should require that a copy $c : \mathit{Copy}$ that "is-held" for a reservation $r : \mathit{Reservation}$ be a copy of the publication that is reserved by the reservation r. To describe constraints of this kind, we need to introduce the notion of *state constraints* which are invariants of the system during its execution. To describe state constraints, we allow to write $t_1 A t_2$ for $A(t_1, t_2)$ when $A \in AName$, $WD_D(A)$, $t_1 : \mathit{lft}(A)$, and $t_2 : \mathit{rht}(A)$.

Definition 3 (**State Constraint**) Given a conceptual diagram D, a *well-formed state constraint* of D is a formula F built from the terms of types declared in D and relational symbols with their names declared in D by using the first order connectives and quantification over typed variables.

The constraints about the library system that a copy $c : \mathit{Copy}$ that "is-held" for a reservation $r : \mathit{Reservation}$ be a copy of the publication that is reserved by the reservation r can be described as

$$\forall c : \mathit{Copy}, r : \mathit{Reservation}, p : \mathit{Publication} \bullet c \text{ Is-Held-for } r \wedge r \text{ Is-on } p \Rightarrow c \text{ Is-Copy-of } p$$

This constraints can be written more abstractly in terms of the algebra of relations:

$$\text{Is-Held-for} \circ \text{Is-on} \subseteq \text{Is-Copy-of}$$

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

2.4 Conceptual models

Now we can define a conceptual model which is a conceptual diagram plus a state constraint that are well-formed in the diagram.

Definition 4 (**Conceptual Model**) A conceptual model $CM = \langle D, Inv \rangle$ where D is a conceptual diagram and Inv is a state constraint that is well-formed in D

For a conceptual model $CM = \langle D, Inv \rangle$, we use $diag(CM)$ to denote for D and $inv(CM)$ for Inv. With Definition 4 for a conceptual model, we can reason about state properties of a conceptual model. Given a conceptual model $CM = \langle D, Inv \rangle$ and a well-formed state constraint F of D, we write $CM \vdash F$ iff $\mathcal{I} \wedge Inv \Rightarrow F$, meaning that F can be proven from $\mathcal{I}$ and Inv in the relational calculus.

Using this proof system, we can reason about the properties of conceptual models and relationships between conceptual models. For example, we can define transformations between conceptual diagrams that have to preserve some state constraints.

Then the semantics of a state constraint is defined based on the semantics of the conceptual diagram and gives a truth value in $\{true, false\}$.

2.5 Object models at conceptual level

For a conceptual model $CM =< D, Inv >$, a *state space* of CM is a state of D, i.e. an object diagram of D, that satisfies Inv as well. Such an object model contains objects of classes and instances of associations that are declared in CM. Each object model of CM must satisfy the constraints $\mathcal{I}$ and $inv(CM)$ of CM. In this way, we treat an object model as a value of a conceptual model in the same way as we used to treat a a state of a program variable. We have also treated a conceptual model as a template and an object model of it as an instance in the same way that we treat an object as an instance of a class.

Definition 5 (**Semantics of a Conceptual Model**) The semantics of a conceptual model CM is the set, denoted by Σ_{CM}, of all the object models of CM.

Therefore, Σ_{CM} is the *system state space* captured by CM, and an object diagram in this state space is called a *system state*. We use S to an arbitrary system state space. An object model[1] of the upper model, denoted by CM_1, in Figure 10 is shown in Figure 6. Notice that this object diagram is not an instance of the bottom model, denoted by CM_1, in Figure 10 as customer c_1 has two accounts when CM_2 does not allow a customer to more than one account.

An object in an object diagram, except for those that are of types of pure-data values and thus used as attributes of other objects, has a state consisting of its *links* with other objects. These links are instances of associations in the given conceptual model. For an object model S of a CM and an association A in which class C is involved in CM, we use $A(c)$ is the set of objects that are associated with c by A in a system state S.

For a conceptual model $CM = \langle \mathcal{C}, \mathcal{R}, \mathcal{A}, \lhd\!\!\longrightarrow \rangle$, we can write a textual description in the following form:

```
Class Model CM::
    Class C1       /*For each C1 in CM is not a super class*/

       association A11: Powerset of C_{11} /*if A11:<C1, C11>*/
```

[1] Notice that we have omitted the association names.

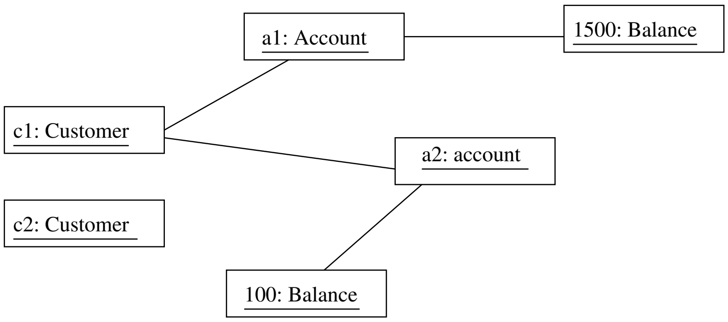

Figure 6: An example of object model

```
      ........
     association A1k: Powerset of C_{1k} /*if A11:<C1, C1k>*/
      ........
  Class C2 is a subclass of C /* For each C2 and each C in CM such that
                              /* C2is a direct specialization of C in CM

     association A21: Powerset of C_{11} /*if A11:<C1, C11>*/

      ........                        /*These are associations of
                                       /*of the subclasses

     association A2m: Powerset of C_{1m} /*if A11:<C1, C1m>*/

  Invariants: F1; F2;......; Fn

END CM
```

Both of the associations and their inverses in a conceptual model must be declared in the text.

2.6 Associative classes

UML allows *associative classes*, classes that in fact models associations. An example of this kind of classes is shown in Diagram (a) of Figure 7.

The Class $JobContract$ is about the association $Employs$. It has an attribute $Salary$, that can be represented as an association as well. As a convention, we shall allow to write attributes inside the box of a class when we draw class diagrams. Such a class and association can be modelled by a separate class and the decomposition of the association into two associations as shown in Diagram (b) of Figure 7. If we do so, we also need to add state constraints about these two new associations so that the original association can be faithfully represented. In this example, we need the following state constraints:

$$\forall c : Company, con : JobContract \bullet c \text{ Has } con \Rightarrow \exists! p : People \bullet con \text{ Is-for } p$$
$$\forall con : JobContract, p : People \bullet con \text{ Is-for } p \Rightarrow \exists! c : Company \bullet c \text{ Has } con$$

The same association class can be modelled by the diagram in Figure 8. With this diagram, we need the constraints:

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

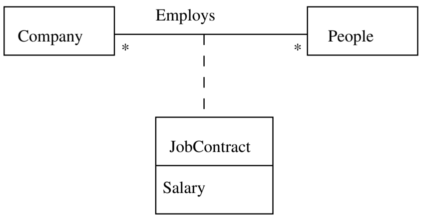

Diagram (a)

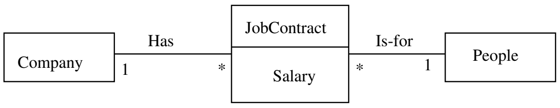

Diagram (b)

Figure 7: Representation of Associative Class

In general, an associative class $A : \langle C_1^{M_1}, C_2^{M_2} \rangle$ in UML can be modelled by adding a class $AClass$ and

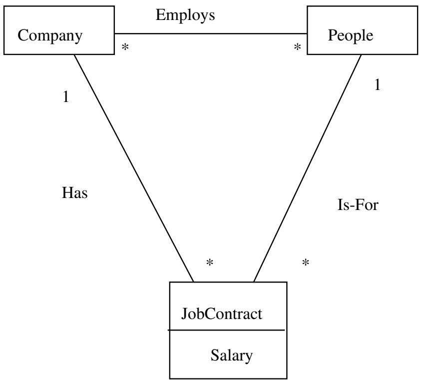

Figure 8: Representation of Associative Class

two associations, $A_1 : \langle C_1^{\{1\}}, AClass^{M_1} \rangle$ and $A_2 : \langle AClass^{M_2}, C_2^{\{1\}} \rangle$ such that

$$A_1 \circ A_2 = A$$

This decomposition can be depicted by Figure 9. We allow the use of the UML associative classes, but we define its semantics as that of Diagram (a). In fact associative classes will be implemented by decomposing the associative class into an addtional class and two two associations. This decomposition also changed the many-to-many association into one-to-many associations that are much easier to realize than many-to-many associations. This treatment of associative classes can be even used in applications where some classes are needed to relate any number of classes. In this way, $AClass$ can have attributes and associations with other classes.

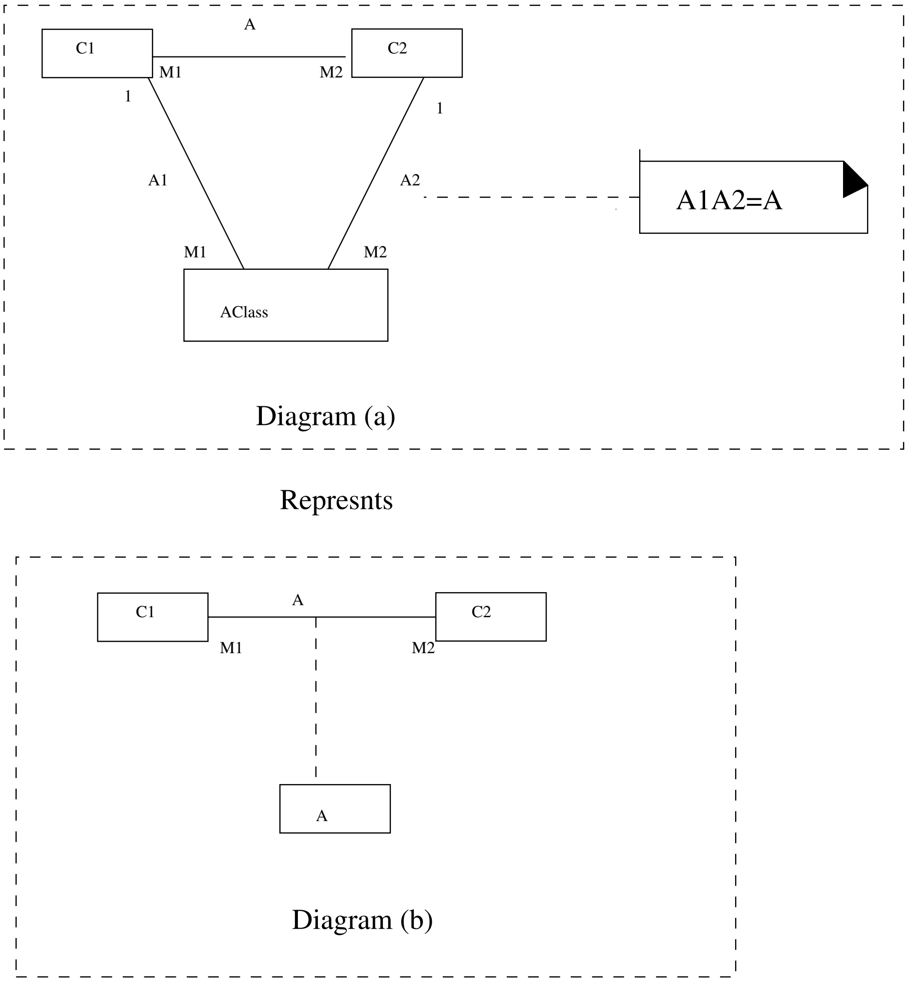

Figure 9: General Representation of Associative Class

2.7 Refining a conceptual model

We can define the a *refinement* relation between two conceptual models.

Definition 6 We say a conceptual model CM_1 refines conceptual model CM_2 if $\Sigma_{CM_1} \supseteq \Sigma_{CM_2}$

The refinement relations is obviously reflexive and transitive.

Theorem 1 *For two conceptual models* $CM_1 = \langle D_1, Inv_1 \rangle$ refines *another* $CM_2 = \langle D_2, Inv_2 \rangle$, $CM_1 \sqsupseteq CM_2$ *if*

1. *any class declared in* D_2 *is declared in* D_1*,*
2. *any association declared in* D_2 *is declared in* D_1*,*
3. $Inv_1 \Leftarrow Inv_2$*, and*
4. $\mathcal{I}^2_{D_1} \Leftarrow \mathcal{I}^2_{D_2}$*, for all* $A \in AName$ *declared in* D_2*.*

The refinement of CM_2 by CM_1 is in fact an enlargement of the allowed state space and thus CM_1 supports more services than CM_2. However, note that this is more a static relation than a dynamic relation. An example of refinement is shown in Figure 10. Let us denote the conceptual model on the top as $Bank_1$ and denote the model at the bottom in the figure. We can see that everything declared in $Bank_2$ is declared in $Banks_1$, also $Bank_1$ allows a customer to have no account, but $Bank_2$ does not allow such customers. An account in $Bank_1$ may have up to five holders (or owners) to allow joint accounts, but $Bank_2$ does not serve customers who want to have joint accounts; $Bank_1$ allows a customer to hold a number of accounts, but $Bank_2$ does not provide this service.

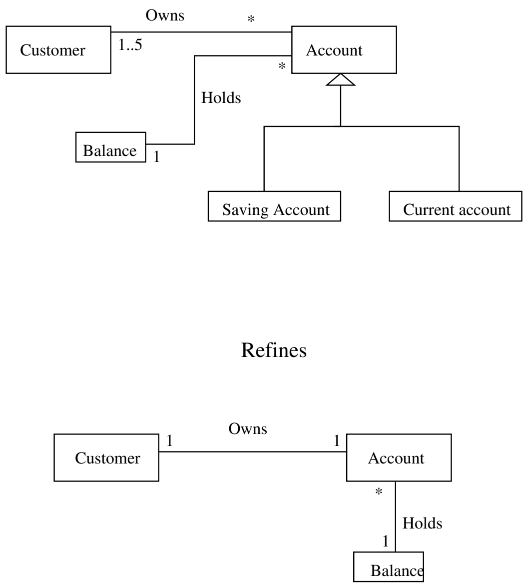

Figure 10: An example of refinement

The above definition of refinement is too strong to be useful enough. We sometimes want to have a refinement by defining some classes and associations of CM_2 in terms of those in CM_1 by using set operations. We denote such a refinement by $CM_1 \sqsupseteq_m CM_2$, where m defines a mapping which defines the classes and associations of CM_2 from those of CM_1 in terms of set operations. After the definition, CM_2 is transformed into a model $m(CM_2)$. Then $CM_1 \sqsupseteq_m CM_2$ iff $CM_1 \sqsupseteq m(CM_2)$. This conforms to the general refinement mapping [AL91].

Given two conceptual models CM_1 and CM_2, we can compose them by the *union* operation, denoted by $\cup$, to build a larger model:

$$CM_1 \cup CM_2 \triangleq \langle diag(CM_1) \cup diag(CM)_2, inv(CM_1) \vee inv(CM_2) \rangle$$

However, we must make sure that the classes with a same name in the two diagrams represent the same *domain concepts*; associations between same classes with a same association name in the two diagrams represent the same *domain association*, and their multiplicities must be consistent as well. Otherwise, renaming of classes and/or associations are needed before the union. The union can only be done when the resulting diagram is well-formed according to the Definition 1. The *union* defined above can also used to extend a diagram. For example, we can extend diagram D_1 in Figure 3 with the diagram in Figure 11, denoted by D_2. Notice that the union of two models CM_1 and CM_1 is a refinement of both CM_1 and CM_2.

In D_2, class $User$ is specialized into two subclasses which are those of students and members of staff who use the library; and class $Publication$ are divided into three subclasses of periodicals, books and

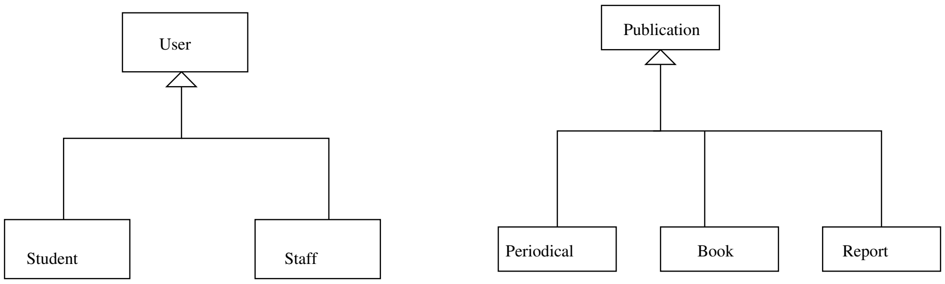

Figure 11: Adding Generalization

unpublished reports. According to Definition 1, $D_1 \cup D_2$ is a well-formed conceptual diagram.

3 Use-Case Model

Given a conceptual model CM, an object diagram represents a snapshot of the system at a moment of time. An *atomic use case* will change the system from one state into another by creating new objects, deleting old objects, forming links between objects or breaking links between objects[2]. Such an atomic use case is equivalent to a system operation in [Lar98]. We shall use the notion of *joint action* in [DW98] to represent an atomic use case. Such an action takes parameter of types of classes and specified in terms of its pre- and post-conditions. The pre- and post conditions are to be defined in terms of the state space of a conceptual model. For a variable $x : T$ of any type T, we assume a *primed version* $x' : T$ of the same type.

Each system state S assigns a value to each variable $v : C$ possibly **null** and the value of $img(A).v$ is determined by S for the value that S has assigned to v . We use the unprimed variables to represent values in the present state, and the primed variables to describe the post state of an action. We use formulas in the first order predicate calculus with terms of the type system. If $\overline{x}$ is the set of all free variables in a formula $F(\overline{x}, \overline{x}')$ of this logic, this formulas is interpreted over pairs (S, S') of system states:

$$[\![F]\!] \triangleq [\![F]\!](S[\overline{x}], S'[\overline{x}'])$$

where $[\![F]\!]$ is semantics of the formula F, $S[\overline{x}]$ is to evaluate $\overline{x}$ in S, and $S'[\overline{x}']$ gives the values of the primed variables. However for this to be defined, S and S' should be states of a same conceptual model CM, and $CM \vdash \overline{x}, \overline{x}'$. For a formula F, let $var(F)$ be the set of all typed variables and primed variables that occur in F, and $free(F)$ is the set of typed *free* variables and free primed variables in F. We say that a list of typed variables $\overline{x}$ is well declared in a conceptual model CM, denoted by $WD_{CM}(\overline{x})$, if all the types are well declared in CM. A joint operation Act can be of the form

$$Act[\overline{pvar};\ \overline{ovar}] \triangleq [\overline{pvar};\ \overline{ovar}] \bullet Pre \vdash Post$$

where Act is an action name that together with the defining symbol can sometimes be omitted, $\overline{pvar}$ a list of parameters typed with classes that are not types of pure-data values, $\overline{ovar}$ denotes a list of returned typed values, and the *precondition* Pre of Act specifies the values of the variables in the current state S of the system. It is thus a first order predicate formula with free variables of the above assumed types, without using any primed variables. The postcondition $Post$ of the action describes the values of the post-state after the action is carried out. It is therefore is a first order predicate formula with free variables and primed

[2]These also include modification of objects' attributes as they are represented as links too.

variables of the above assumed types. We also require that all free variables, excluding those C's in Pre and $Post$ should be declared in $\overline{pvar}$ or $\overline{ovar}$:

$$(free(Pre) \cup free(Post)) - \{C, C' \mid C \in CName\} \subseteq \{v, v' : v \in \overline{pvar} \cup \overline{ovar}\}$$

Please note that the parameters of an action represents objects that might be different from one occurrence to another. Some of the parameters may be distinguished as (*participants*) which do not have different semantic meanings from the other parameters at the requirement. However, in the design, a participant of a use case will play roles to control and coordinate the behaviour of other objects. At the requirement, we may also include the *actors* of a use case in the parameter list, but their states are not usually to be described in the pre and post condition. They will not be realized as software classes in the system design *unless access control needs to be designed and implemented because of security reasons*.

Atomic use cases can be used to make up a bigger use case U defining

- *sequential composition*, U_1 ; U_2;
- choice among the a number of use cases which may be conditionally guarded: $[]_{i=1}^{n} g_i \rightarrow U_i$ where g_i is a condition on some attributes (associations at this level) of objects in the current state or input parameters of U_i.
- *iteration* **do** $[]_{i=1}^{n} g_i \rightarrow U_i$ **od**

These are enough to model the *extends* and *uses* relationships between uses cases in UML.

A use case can only be defined in the context, or *environment* of a conceptual mode CM, denoted by $CM :: U[\overline{pvar};\ \overline{ovar}]$. We define the semantics $[\![CM :: U[\overline{pvar};\ \overline{ovar}]]\!]$ by defining the semantics of the joint actions, and then the semantics composite use cases are defined in the traditional way.

The semantics $[\![CM :: [\overline{pvar};\ \overline{ovar}] \bullet Pre \vdash Post]\!]$ of a joint action is defined as

$$WD_{CM}(\overline{pvar};\ \overline{ovar}, Var(Pre), Var(Post)) \wedge Pre \wedge ok \Rightarrow Post \wedge ok'$$

where *ok* is a logical state variables which represents that the program is in a proper (an *ok*) state to start the execution of the action. This semantics means an action can be properly carried out only when the current state is a proper state to start the execution of the action, all variables are declared (an all terms should be well typed), and the precondition holds in the current state. If this is true, the execution of the action transforms the current state into a state that is related with the current state by $Post$ and the execution will properly terminate, otherwise we cannot say anything about what the action does – *chaos* [HH98].

We say that a conceptual model CM is *adequate* for an use case U if

1. for U as an atomic use case $[\overline{pvar};\ \overline{ovar}] \bullet Pre \vdash Post$, CM is adequate for U if
   (a) every type used in the action is declared in CM, and
   (b) the Pre condition is *satisfiable* in the state space Σ_{CM}, i.e. $\mathcal{I} \wedge inv(CM) \wedge Pre \neq false$.
2. For U as a composite use case, CM is adequate for U if CM is adequate for all atomic use cases of U.

The notion of *adequateness* of a conceptual model with regard to a use case intends to support incremental development of a requirement model.

Theorem 2 *If $CM_1 \sqsupseteq CM_2$ and CM_2 is adequate for a use case U, then CM_1 is adequate for use case U as well.*

3.1 Examples of use cases

This subsection gives some small examples of use cases. We us the two models $Bank_1$ on the top and $Bank_2$ at the bottom in Figure 10. Under $Bank_2$, we can specify a use case that allows a customer to withdraw a certain amount of money from his/her account.

$$\begin{array}{l} Withdraw_1[c : Customer, b : Balance;\ out : Balance] \triangleq \\ Pre\ :\ c \in Customer \\ Post\ :\ (Has(Owns(c))' = Has(Owns(c)) - b) \wedge (out' = b) \end{array}$$

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

$$\begin{array}{l} Withdraw_2[c : Customer, a : Account, b : Balance;\ out : Balance] \triangleq \\ Pre\ :\ c \in Customer \wedge a \in Account \wedge Owns(c, a) \\ Post\ :\ (Has(a)' = Has(a) - b) \wedge (out' = b) \end{array}$$

Model $Bank_1$ can support a "withdraw" use case that behaves different from the above one:

$$\begin{array}{l} Withdraw_3[c : Customer, b : Balance;\ out : Balance] \triangleq \\ Pre\ :\ c \in Customer \wedge \exists a \in Account \bullet Owns(c, a) \\ Post\ :\ (Has(a)' = Has(a) - b) \wedge (out' = b) \end{array}$$

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

$$Bank_2 \vdash c \in Customer \Leftrightarrow c \in Customer \wedge \exists a \in Account \bullet Owns(c, a)$$

We can see that $Bank_1$ supports the a use case 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, CM_1 supports more use cases than CM_2. Under $Bank_1$, the transfer use case can be written as:

$$\begin{array}{l} Transfer[c : Customer, from,\ to : Accounts, b : Balance] \triangleq \\ Pre\ :\ c \in Customer \wedge Owns(c, from) \wedge Owns(c, to) \\ Post\ :\ (Has(from)' = Has(from) - b) \wedge (Has(to)' = Has(to) + b) \end{array}$$

Figure 12 shows the effect of $Withdraw_1$ on a state of $Bank_2$, and Figure 13 illustrates the effect of $Transfer$ use case on a state of $Banks_1$. We can think of other use cases on $Banks_1$, such as $Joint(c : Customer, a : Account)$ with links customer c to account a by association $Owns$, $Pay(c_1, c_2 : Customer, a_1, a_2 : Account, b : Balance)$ which allows customer c_1 to transfer b amount of money from his/her account c_1 to account a_1 of customer c_1.

4 Conclusion & Discussion

We have given a semantics to the conceptual models and use-cases models of UML that are the main models to be built during the requirement analysis. The introduction of type $InstanceTypeof(C)$ is important to allow us to define class names and association names as variables, and an object diagram as a state of a conceptual model. The set of all object diagrams satisfying the state constraints enforced by a conceptual

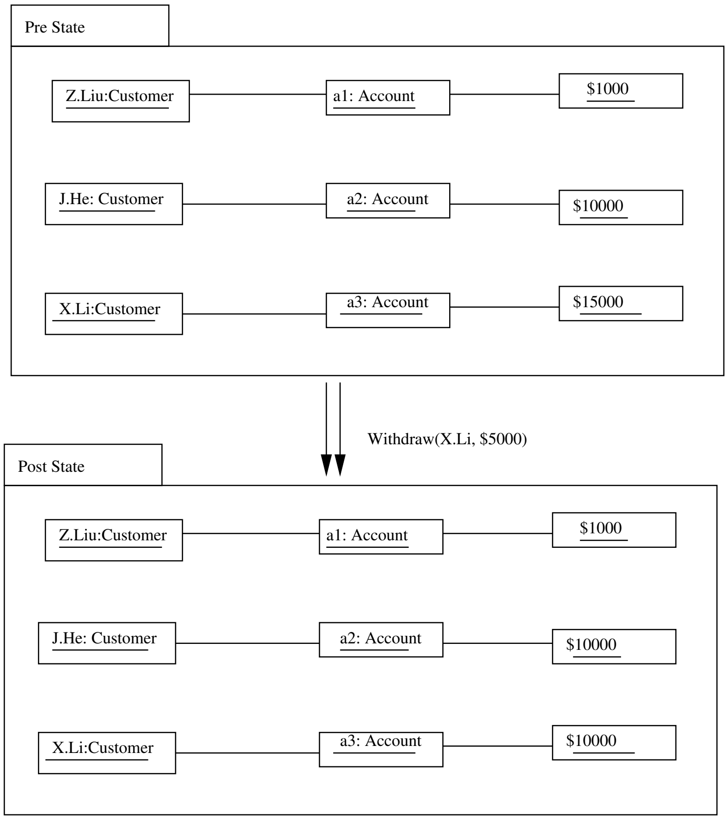

Figure 12: Effect of Withdraw Use Case

model is defined as the semantics of the model. Under such this semantic definition for a conceptual model, a use case is then defined as an action that transforms one state into another state of the conceptual model. A use case can thus be specified in terms as it pre and postconditions. Therefore the semantics of use cases is tightly related with that of the conceptual model. The semantic definitions are easy to understand as they are entirely based on the well established *state-based* semantic approach [HH98]. The formalization provides a clearer understanding about the meaning of associative classes, and enforces with axioms on conceptual models principles for better modelling using the generalization hierarchy.

The formalization supports building up a model step by step. It faithfully reflects the informal way of using UML for requirement analysis. The consistency between two models can be easily checked. The refinement between two conceptual models is easy to prove, and the refinement between two use-case case models follows the traditional refinement calculus [Bac88, HHS86, Mor94, BvE98, HH98] for imperative programs. The main difference between our work with that in [pG99] is that we are providing with formal relationships between different models used in UML during a development process, rather than only formalizations of the different models. Compared with [Jon94, Jon96, CN00], the development of our framework is in a top-down style and this helps a lot for ease of understanding of the the formalization. This is only a starting point of our research toward a formal use of UML in OO systems development. The very next step is to define semantics for UML design models, the refinement of use-cases into interactions between objects, and refinement between design models in UML. We aim to develop whole framework in an incremental manner so that the complexity will not become overwhelming.

Acknowledgements: The first author and the third author would like to thank Yifeng Chen for giving us a

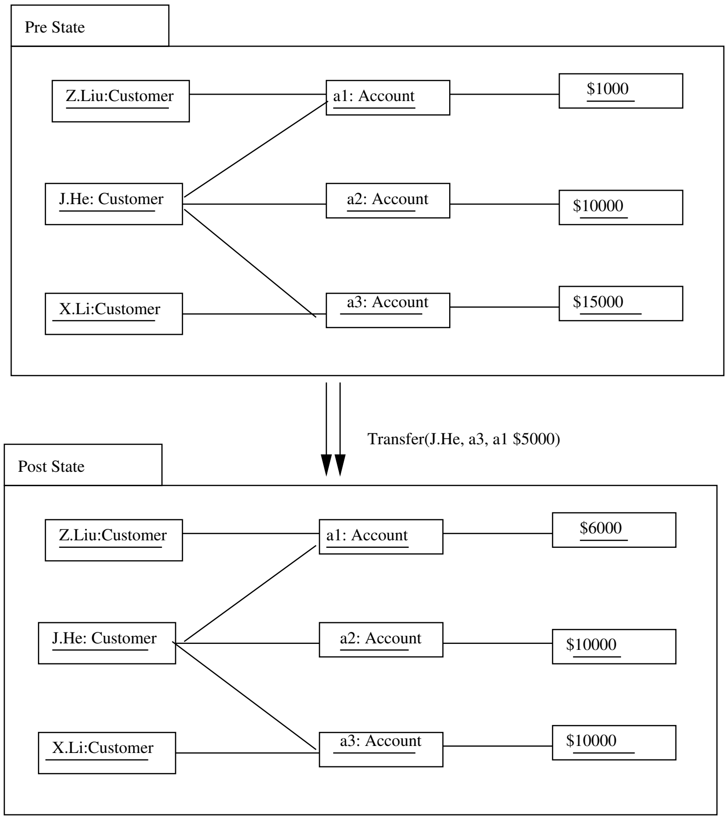

Figure 13: Effect of Transfer Use Case

talk on He and Hoare's *Unifying Theories of Programming*. We wish to thank Chris George, Wang Yanjie and Wang Zhuo for the discussions about how to use UML in their applications. Those discussions have provided us with some examples and ideas used in this paper.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Sprginger, 1996.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 83(2):253–284, 1991.

[Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[BKS98] M.M. Bonsangue, J.N. Kok, and K. Sere. An approach to object-orientation in action systems. In J. Jeuring, editor, *Mathematics of Program Construction,* LNCS 1422, pages 68–95. Springer, 1998.

[BRJ99] G. Booch, J. Runbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[BvE98] R.J.R. Back and J. von Eright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science, Springer-Verlag, 1998.

[CN00] A. Cavalcanti and D.A. Naumann. A weakest precondition semantics for an object-oriented language of refinement. Technical Report CS Report 9903, Stevens Institute of Technology, Hoboken, NJ 07030, February 2000.

[DW98] D. D'Souza and A.C. Wills. *Objects, Components and Framework with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[Fow97] M. Fowler. *UML Distilled – Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[HH98] J.F. He and C.A.R. Hoare. *Unifying theories of programming*. Prentice-Hall International, 1998.

[HHS86] J.F. He, C.A.R. Hoare, and J.W. Sanders. Data refinement. In *Lecture Notes in Computer Science 213*, pages 187–196. Springer-Verlag, 1986.

[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.

[JBR99] I. Jacobson, G. Booch, and J. Runbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Jon94] C.B. Jones. Process algebra arguments about an object-oriented design notation. In A.W Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice-Hall, 1994.

[Jon96] C.B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, 1996.

[Ken97] S. Kent. Constraint diagrams: Visualising invariants in object-oriented models. In *OOPSLA97*. ACM Press, 1997.

[Lar98] C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 1998.

[Liu00] Z. Liu. Lecture notes for mc206 software engineering and system development. 2000.

[LW95] X. Liu and D. Walker. Confluence of processes and systems of objects. In P.D. Mosses, M. Nielsen, and M.I. Schwartzback, editors, *Theorey and Practice of Software Development, Lecture Notes in Computer Science 915*, pages 217–231. Springer-Verlag, 1995.

[Mey97] B. Meyer. *Object-oriented Software Construction (2nd Edition)*. Prentice Hall PTR, 1997.

[MMMNS00] L. Mathiassen, A. Munk-Madsen, P.A. Nielsen, and J. Stage. *Object Oriented Analysis and Design*. Forlaget Marko, Aalborg, Demark, 2000.

[Mor94] C.C. Morgan. *Programming from Soecifications, 2ed.* Ptentice Hall, 1994.

[pG99] The pUML Group. The precise uml web site: /http://www.cs.york.ac.uk/puml. 1999.

[PS99] R. Pooley and P. Steven. *Using UML: Software Engineering with Objects and Component*. Addison-Wesley, 1999.

[RJB99] J. Runbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.