Formalization of the Use-Case Driven Requirement Analysis Process*

Zhiming Liu†,† Xiaoshan Li‡, and Jifeng He§

† Department of Mathematics and Computer Science
The University of Leicester, Leicester, LE1 7RH, UK.

‡ Faculty of Science and Technology
The University of Macau, Macau

§ International Institute for Software Technology
The United Nations University, Macau

Abstract

We have recently proposed a formalization of the use of UML in requirement analysis. This paper applies that formalization to a library system as a case study. We intend to show how the approach supports a use case-driven, step-wised and incremental development in building models for requirement analysis. The actual process of building the models shows the importance and feasibility of the formalization itself.

Keywords: *Conceptual Model, Use Cases, Object-Orientation, Incremental Model Building, UML.*

1 Introduction

Until the late 1990's, the functional and structural analysis, design and coding process described by the "water-fall model" had been the dominant discipline for software development. A clear advantage of this approach is that the stages that a development process passes through are well organized and the activities at different stages can be well managed. The main disadvantage of the approach is that it encourages or suggests that the whole system must be dealt with at each stage: the requirement specification and analysis for the system, the design of the whole system, and then the implementation of the whole system. The system validation including verification of the design and testing of the implementation of the system has to be considered for the system as a whole too. Such a development process does not support ease of maintenance. This is fine for small sized software development, but not feasible for large system. Although modular and compositional approaches have been proposed and indeed used, the problem of how to decompose of a system into components or modules and then compose them together to meet the whole system requirements has never been well solved.

Formal methods were born and growing up in those years mainly for justifying, better understanding, and more precise and rigorous use of techniques in parts of a "water-fall model" of a development process. So we have theories of formal specification, verification, refinement, decomposition and composition. These have helped in improving the quality of the system developed so that they are more correct and safer to use. On the other hand, formal methods have inherited the same disadvantages from the informal use of the "water-fall model" of the structural analysis approach, and they suffer even more seriously from these

*This work is partly supported by the British EPSCRC Grant GR/M89447. This is an extended and revised version of [LLH01]
†On leave from the University of Leicesterand working as a guest scientist at UNU/IIST, Macau.

disadvantages as a specification of the whole system at any level, e.g. the requirement level, in a formal notation is not understandable to most system engineers, not to mention about formal verification. This may be the main reason why the use of formal methods cannot be scaled up and widely accepted in large scale software developments.

Another difficulty to scale up the use of formal methods and to use it in industries might be due to the fact that most formal methods, including those for OO development [Jon94, Jon96, AC96, CN00], are developed in a bottom-up approach. 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 levels 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, CN00].

Our work in [LHL01] is to support the formal use of UML in OO system development processes and development of tools for consistency checking. In contrast to most work on formal methods for OO development that uses a bottom-up approach [Jon96, CN00], it follows the UML evolutionary approach in a development process to develop a semantics for UML, and aims to achieve simplicity and ease of understanding. The framework is based on the set theory and the notion of *pre* and *post* conditions. The method is expected to be usable within an incremental and iterative development process driven by use cases [JBR99]. Such a development process has shown promising in overcoming the disadvantages of the "water-fall model" of the traditional functional and structural development for a class of so called *software intensive systems*. This paper uses a library system as a case study to show how the formalization in [LHL01] for UML conceptual models and use cases can help to improve the use of formal methods in requirement analysis of large scale systems, as well as to enhance the use of UML itself in requirement analysis with a formal semantics.

The rest of the paper is organized in the following way. Section 2 gives a brief summary of [LHL01]; Section 3 presents the library case study; Section 4 illustrates the method by building the requirement models for the case study; and finally Section 5 concludes the paper with discussions.

2 Conceptual Model and Use Case Model

The main UML models to be produced at the requirement analysis are a *use-case model* and a *conceptual model*. The use case model consists of a set of use cases, each of which describes a service that the system is to provide for some kinds of users called *actors*. The use-case model describes the functional requirement.

The conceptual model describes a set of concepts by *class name*, how these classes are related by *associations*, and a set of assertions about the relationships among the associations.

On the one hand, the description of the use cases provides important information about what should be in the conceptual model. On the other hand, the effect of a use case can only be defined in the context of a conceptual model in terms what objects should be created and deleted, and which and how associations between objects including attributes of objects are changed. Such a close relation between the use case model and the conceptual model suggests that if we write out an specification of a use case, we should be able to build part of the conceptual model that is *adequate* [LHL01] for the definition of the use case. The conceptual model will be extended while further use cases are captured and defined. This is similar to the technique of *noun-phrase identification* for the creation of a conceptual model from a use case [Lar98, JBR99, Liu01]. To formalize these ideas, we assume two disjoint sets of names $CName$ and $AName$ for representing classes and associations, and introduce the following types and variables:

- for each $C \in CName$, assume a type called $ObjectTypeof(C)$ which is non-empty;

- we allow to declare a variable $x : ObjectTypeof(C)$;
- each $C \in CName$ is treated as a variable of type $C : \mathbb{P}ObjectTypeof(C)$;
- each $A \in AName$ is treated as a variable of type $A : \mathbb{P}(ObjectTypeof(C_1) \times ObjectTypeof(C_1))$ for some $C_1, C_2 \in CName$, and we use $A :< C_1, C_2 >$ as its shorthand;
- for each $A \in AName$, there is an $A^{-1} \in AName$ such that $(A^{-1})^{-1} = A$ and $A(c_1, c_2)$ iff $A^{-1}(c_2, c_1)$;
- for each variable $x : T$ of any type T, x' is variable that is distinct from x but of same type T as x,x' is called the primed version of x; primed variables are used to distinguish the values of variables in the the state after an joint action from those before the action.

These types have to be declared in a conceptual model before they can be used to define a use case.

2.1 Use cases

In [LHL01], a use case is defined to be a *parameterized joint action* of the following 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 or types of pure-data values, $\overline{ovar}$ denotes a list of typed variables that can be modified by the action and is called the *frame* of the action. We sometimes omit the frame and assume that only the variables with their primed versions occurring in the postcondition may be modified. We call $[\overline{pvar};\ \overline{ovar}]$ the *signature* of the action. 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 a first order predicate formula with free variables and primed variables of the above assumed types.

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. We may also include the *actors* of a use case in the parameter list.

2.2 Conceptual model

A conceptual model $M = \langle D, Inv \rangle$ is a pair of a conceptual diagram D and an assertion Inv about the objects and associations in D. A *conceptual diagram* is a tuple: $D = \langle \mathcal{C}, \mathcal{A}, \lhd\!\!\longleftarrow, \mathcal{R} \rangle$, 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 $\mathcal{R}(A^{-1}) =< C_2, M_2, M_1, C_1 >$.
- $\lhd\!\!\longleftarrow \subseteq \mathcal{C} \times \mathcal{C}$ is the generalization relation between classes. We use $C_1 \lhd\!\!\longleftarrow C_2$ 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 $C_i \lhd\!\!\longleftarrow C_{i+1}$, for $i = 1, \ldots, n$, and $C_n \lhd\!\!\longleftarrow C_1$. We also write $C_2 \longrightarrow\!\!\rhd C_1$ for $C_1 \lhd\!\!\longleftarrow C_2$.

A conceptual model $M = \langle D, Inv\rangle$ thus declares a finite set $\mathcal{C} \subset CName$ of *class variables*, a finite set $\mathcal{A} \subset AName$ of *association variables* with their types and multiplicity constraints, and a set of generalization relations between the declared classes. These variables are *global* from the system point of view and are constrained by the assertion Inv as well as a set of constraints enforced by the D itself [LHL01]. We denote by $\mathcal{I}_M$ the conjunction of all these state constraints [LHL01]. M can be treated as *big composite state variable* which takes values of *object diagrams* [LHL01] that are instances of D and satisfying Inv, and we denote by Σ_M the set of all such object diagrams.

A use case describes a functional requirement and specifies what it should do for the actors. Its effect should be described in terms what changes it can make on the state of a conceptual model. This means that we treat the variables and association variables as global and they can be modified by the system use cases. A conceptual model M is *adequate* to define a use case $Act[\overline{pvar};\ \overline{ovar}] \triangleq [\overline{pvar};\ \overline{ovar}] \bullet Pre \vdash Post$ if all the class variables and association variables are well declared in M and the pre-condition is satisfiable in the state space Σ_M. We can weaken adequateness of a conceptual model by removing the condition that the pre-condition is satisfiable in the state space Σ_M as it is impossible to be checked statically.

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

$$WD_M(\overline{pvar};\ \overline{ovar},\ Var(Pre),\ Var(Post)) \wedge \mathcal{I}_M \wedge Pre \wedge ok \Rightarrow Post \wedge \bigwedge_{v \notin \overline{ovar}} (v' = v) \wedge ok'$$

where $v \notin \overline{ovar}$ means that v is in the underlying class diagram and in the set of free variables of the action, but not in $\overline{ovar}$; *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].

3 Informal Description of a Library System

The library system is used to support the management of loans in a university library. Librarians maintain a catalogue of publications which are available for lending to users. There may be many copies of the same publication. Publications and copies may be added to and removed from the library. Registered users can borrow the available copies in the library. When a copy has been borrowed by a user, it is on loan and is not available for lending to other users. When all copies of a publication have been borrowed, users can make a reservation for the publication. However, a user may not place more than one reservation for the same publication. When a copy is returned to the library, the loan will be put into the loan record in the library. After a copy is returned, it may be put back on the shelf, or alternatively, held for a user who has reserved the corresponding publication of the copy.

From above informal description of the system requirements, [Ken97] identified the following use cases which should be supported by the system.

1. Librarians can maintain the library, such as add and remove publications, copies, and users.
2. A library lends copies to users.
3. User can make a reservations and remove reservations.
4. User can return the copies.

These use cases were organized into the use-case diagram in [Ken97, LLH01]. From the analysis of these use case *informally*, paper [Ken97] also produced a conceptual diagram and listed out a number of state assertions about the class diagram. The diagram was represented in [LLH01] with some names changed only.

However, there are a number of questions need to be addressed:

1. How much noise is contained in the conceptual model, i.e. how can we justify that all the classes and association are relevant for the realization of the use cases?
2. How can we justify that the conceptual model is adequate for the use cases?
3. Which part is relevant to a use case?
4. How is this conceptual model produced?
5. How can we justify that the lists state constraints or invariants against the use uses and vice versa?

We shall attempt to discuss these questions in the next section when we actually develop the models.

4 Requirement Analysis of the Library System

From the signature of an joint action and the types of the variables in its pre and post conditions, we can extract the classes and their association that are needed to realize or define the effect of the use cases. This will lead to the creation of a conceptual class diagram. Analysis of the conceptual diagram and the use cases will derive the state constraints [LHL01] required for the conceptual model. We now define a number of use cases one by one and at same time to develop a conceptual model step-by-step.

Use case LendCopy This use case is about how the library can lend a copy of a publication to a user. Obvious, a user u and a copy c are participants in this action, and a loan ℓ should be created for user u and copy c. However, there may be some publications that are not allowed to be borrowed by some users. This use case can be formally specified as

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{LendCopy}[c : Copy, u : User] \triangleq} \\
\textbf{ovar} : & & Loan : \mathbb{P}\,ObjectTypeof(Loan); & \\
 & & \text{Borrows} :< Loan, Copy >; & \\
 & & \text{Takes} :< User, Loan >; & \\
 & & \text{IsAvailable} :< Copy, User >; & \\
\textbf{Pre} : & & c \in Copy \wedge u \in User & c \text{ and } u \text{ exist} \\
 & \wedge & \text{IsLendableTo}(\text{IsCopyof}^{-1}(c), u) & u \text{ is allowed to borrow } c \\
 & \wedge & \text{IsAvailable}(c, u) & c \text{ is available to } u \\
\textbf{Post} : & & \exists \ell : ObjectTypeof(Loan) \bullet \ell \notin Loan & \\
 & \wedge & Loan' = Loan \cup \{\ell\} & \text{create a new loan} \\
 & \wedge & \text{Borrows}' = \text{Borrows} \cup \{< \ell, c >\} & \text{record } c \text{ on the new loan} \\
 & \wedge & \text{Takes}' = \text{Takes} \cup \{< u, \ell >\} & \text{record } u \text{ on the new loan} \\
 & \wedge & \text{IsAvailable}' = IsAvailable - \bigcup_{u \in User} \{< c, u >\} & \text{make } c \text{ unavailable to any user}
\end{array}
$$

We can extract the classes and associations from this definition and make them into a conceptual diagram in Figure 1, denoted by D_1.

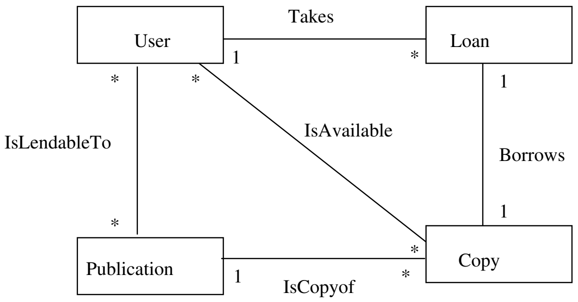

Figure 1: Conceptual model for $LendCopy$ use case

From the definition of $LendCopy$ use case, a copy can only be borrowed in one loan and a loan can be taken by one user, but a user may use $LendCopy$ again and again to take more loans. These imply the multiplicities of the associations in the diagram. The following state assertions are preserved by the use case:

(I_{11}).	$\text{IsAvailable}^{-1}(User) \cap \text{Borrows}(Loan) = \emptyset$	a copy on loan cannot be available
(I_{11}).	$\text{Takes}(User) = \text{Borrows}^{-1}(Copy)$	each loan records a user and a copy

where we have adopted the convention that $\sharp S$ is the number of elements in set S, if $R \subseteq T_1 \times T_2$ and S is a subset of T_1, $R(S)$ is the set of elements that are related to those in set S. Define $I_1 \triangleq I_{11} \wedge I_{12}$ and let $LM_1 = \langle D_1, I_1 \rangle$ be the conceptual model constructed for $LendCopy$ use case.

Use case AddPublication A library may add a new publication into the system and this service is provided by a use case called $AddPublication$. We thus need to introduce a class $Library$ to represent the library concept. Assume there is only one library object, denoted by lib. Then there is an *aggregation* association relation *Contains* between $Library$ and $Publication$. The use case $AddPublication$ can be specified as:

AddPublication$[p : Publication, lib : Library] \triangleq$

ovar	:	$Publication : \mathbb{P}ObjectTypeof(Publication)$;	
		Contains $:< Library, Publication >$;	
Pre	:	$p \notin Publication$	p is not currently in the lib
Post	:	$Publication' = Publication \cup \{p\}$	create the publication
	$\wedge$	$\text{Contains}' = \text{Contains} \cup \{< lib, p >\}$	p now belongs to lib

Only two classes, $Library$ and $Publication$, and one association $Contains :< Library, Publication >$ are needed to define this $AddPublication$ use case. The extension of the conceptual diagram D_1 in Figure 1 with these newly introduced classes and association is the class diagram in Figure 2, denoted by D_2. We formally defined in [LHL01] how to extend a conceptual model. If all publications are added by applications of this $AddPublication$ use case, we have the following two state constraints:

(I_{21}).	$Library = \{lib\}$	there is only one library
(I_{22}).	$Contains(lib) = Publication$	all publications are registered

Adding these two constraints onto D_2 with those constraints of D_1, we obtain the conceptual model $M_2 = \langle D_2, I_2 \rangle$, where $I_2 \triangleq I_1 \wedge I_{21} \wedge I_{22}$.

Use case AddCopy Use case $AddCopy$ is used to add a new copy of a publication to the library after its corresponding publication has already been created. If there is not the corresponding publication of the new

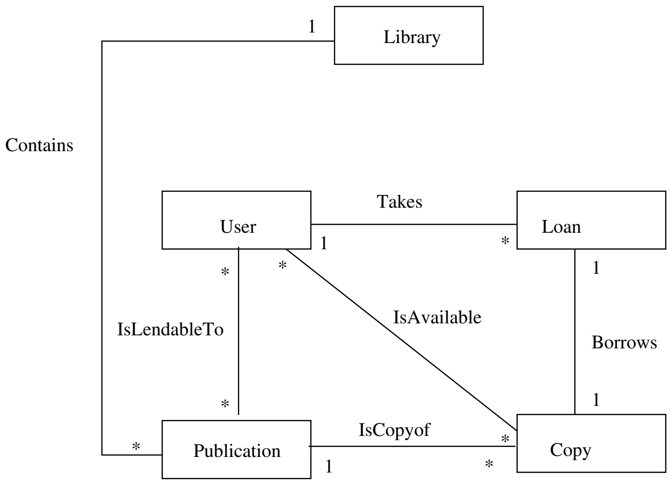

Figure 2: Conceptual model after adding use cases of $AddPublication$ and $AddCopy$

copy, we should first call use case $AddPublication$ to create the publication, and then carry out $AddCopy$ use case. $AddCopy$ is therefore defined as follows.

AddCopy$[c : Copy, p : Publication] \triangleq$			
ovar	:	$Copy : \mathbb{P}ObjectTypeof(Copy)$;	
		IsCopyof $:< Copy, Publication >$;	
		IsAvailable $:< Copy, User >$;	
Pre	:	$c \notin Copy \wedge p \in Publication$	c is new but p exists
Post	:	$Copy' = Copy \cup \{c\}$	create c and add it in
	$\wedge$	IsCopyof$'$ = IsCopyof $\cup \{< c, p >\}$	record that c is a copy of p
	$\wedge$	IsAvailable$'$ = IsAvailable $\cup \{< c, u >: u \in User\}$	make c available

In [Ken97], an association relation *Holds* between class $Library$ and class $Copy$ is used. In fact, it is an association derivable from association relations Contains and IsCopyof:

$$\text{Holds} = \text{Contains} \circ \text{IsCopyof}^{-1}$$

Some software practitioners suggest that derivable associations should not be shown in a conceptual diagram to keep the model simpler [Lar98], while others says that they are better to be shown. We feel that instead of showing these derivable associations, they should be defined in text because the definition should be given even if they are shown. Conceptual diagram D_2 in Figure 2 is still adequate for the definition of $AddCopy$. However, we need to notice the use case preserves the constraint imposed by the many-to-one multiplicities of $IsCopyof$ and the following property that each copy must be a copy of a publication in the system:

$$(I_{31}).\ Copy = \text{IsCopyof}^{-1}(Publication)$$

Add this state constraint into conceptual model M_2 to obtain a conceptual mode M_3. Similarly, we can define use case $AddUser(u : User)$ which introduces an association relation *Registers* between classes $Library$ and $User$. We add association *Regesters* to M_3 and get a new model denoted by M_4

Use case MakeReservation When a user u wants to borrow a publication p and there is no copy of this publication available to him or her, the system should allow the user to make a reservation r on the publication p. When a copy of p is returned, it should be held for the reservation u. Therefore, use case $MakeReservation$

should introduce a new class $Reservation$ and three associations *Makes, IsOn,* and *IsHeldfor* among classes $User$, $Publication$ and $Copy$.

The above informal description is taken from the client's requirement in Section 2 about making a reservation. It suggests that the use case for making a reservation should be considered together with the use case for returning a borrowed copy. This implies that sometimes a group of use cases should be analysed together. Such use cases are called *tightly coupled use cases* and they together with their conceptual model should be documented in a UML *package*. Use case $MakeReservation$ can be defined formally as follows.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{MakeReservation}[u : User, p : Publication] \triangleq} \\
\textbf{ovar} & : & Reservation : \mathbb{P} ObjectTypeof(Reservation); & \\
 & & \text{Makes} :< User, Reservation >; & \\
 & & \text{IsOn} :< Reservation, Publication >; & \\
\textbf{Pre} & : & p \in Publication \land u \in User & u \text{ and } p \text{ exist} \\
 & \land & IsLendableTo(p, u) & p \text{ is lendable to } u \\
 & \land & \neg\exists c \in Copy \bullet (\text{IsCopyof}(c, p) \land \text{IsAvailble}(c, u)) & \text{no copy of } p \text{ available} \\
 & \land & \neg\exists r \in Reservation \bullet \text{Makes}(u, r) \land \text{IsOn}(r, p) & u \text{ not already reserved } p \\
\textbf{Post} & : & \exists r : ObjectTypeof(Reservation) \bullet (r \notin Reservation \land & \\
 & & Reservation' = Reservation \cup \{r\}) & \text{make a new reservation} \\
 & \land & \text{Makes}' = \text{Makes} \cup \{< u, r >\} & \text{record } u \text{ in the reservation} \\
 & \land & \text{IsOn}' = \text{IsOn} \cup \{< r, p >\} & \text{record } p \text{ in the reservation}
\end{array}
$$

Notice that more than one reservation can be made on one publication, but not by the same user. Adding the association introduced by this use case to the conceptual diagram in M_4, we get the class diagram D_5 in Figure 3.

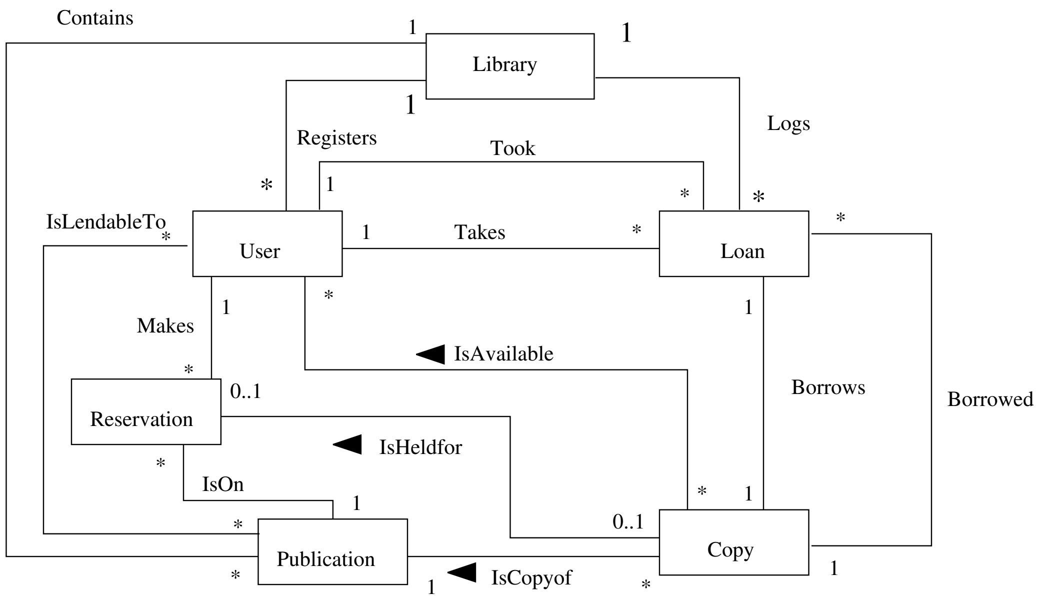

Figure 3: Conceptual model after adding MakesReservation use case

Use case ReturnCopy The use case for returning a borrowed copy can be defined as follows. Notice that in its post condition, a copy is made to be held for one reservation if there is any. We also need to introduce two associations *bf Took* and **Borrowed** to record the completed loans when a copy is returned.

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\textbf{ReturnCopy}[c : Copy] \triangleq} \\
\textbf{ovar} & : & Copy : \mathbb{P}\, ObjectTypeof(Copy); & \\
& & Loan : \mathbb{P}\, ObjectTypeof(Loan); & \\
& & \text{Takes} :< User, Loan >; & \\
& & \text{Borrows} :< Loan, Copy >; & \\
& & \text{Took} :< User, Loan >; & \\
& & \text{Borrowed} :< Loan, Copy >; & \\
& & \text{IsHeldfor} :< Copy, Reservation >; & \\
& & \text{Logs} :< Library, Loan >; & \\
& & \text{IsAvailable} :< Copy, User >; & \\
\textbf{Pre} & : & c \in Copy & \text{the copy exists in the system} \\
& \wedge & \exists \ell \in Loan \bullet \text{Borrows}(\ell, c) & \text{the copy is on loan} \\
\textbf{Post} & : & \textbf{Let } \ell = \textbf{Borrows}^{-1}(c) \textbf{ and } u = \textbf{Takes}^{-1}(\ell) \textbf{ in} & \\
& & Loan' = Loan - \{\ell\} & \text{remove the loan and} \\
& \wedge & \text{Takes}' = \text{Takes} - \{< u, \ell >\} & \text{break the link} \\
& \wedge & \text{Borrows}' = \text{Borrows} - \{< \ell, c >\} & \text{break the link} \\
& \wedge & \text{Logs}' = \text{Logs} \cup \{< lib, \ell >\} & \text{Log the completed loan} \\
& \wedge & \text{Took}' = \textbf{Took} \cup \{< u, \ell >\} & \text{record the link} \\
& \wedge & \text{Borrowed}' = \text{Borrowed} - \{< \ell, c >\} & \text{record the link} \\
& \wedge & \underline{\textbf{if}}\ U = \text{Makes}^{-1}(\text{IsOn}^{-1}(\text{IsCopyof}(c))) \neq \varnothing & \text{if the } p \text{ of } c \text{ is reserved} \\
& & \underline{\textbf{then}}\ \text{IsHeldfor}' = \text{IsHeldof} \cup \{< c, choice(U) >\}) & \text{hold } c \text{ for one reserver} \\
& & \underline{\textbf{else}}\ \text{IsAvailable}' = \text{IsAvailable} \cup \bigcup\limits_{u \in User} \{< c, u >\} & \text{make } c \text{ available to anyone}
\end{array}
$$

Notice that the use of *choice* function means that no decision is made in the postcondition about for which reservation that the copy should be held for if more than one reservation is made for the publication. Such a decision should be made when the use case is to be designed and implemented. The newly introduced associations are related to the existing ones in the following way:

(I_{51}). $\text{Took}(User) = \text{Borrowed}^{-1}(Copy) = \text{Logs}(lib)$
(I_{52}). $\text{Takes} \cap \text{Took} = \emptyset$
(I_{53}). $\text{Borrows} \cap \text{Borrowed} = \emptyset$

Define the set of current loans as

$$CurrentLoan \triangleq Loan - \{\ell : \ell \in Logs(lib)\}$$

Use case CollectReservation When a copy is held for a reservation, the user of the reservation will go to collect the copy. In [Ken97], collecting the copy held for a reservation is part of use case $LendCopy$. However, we found no justification of doing so. We prefer to introducing a separate use case for this purpose. Of course it can be combined with $LendCopy$ use case in the design.

$\textbf{CollectReservation}[u : \mathit{User}, r : \mathit{Reservation}] \triangleq$

$$
\begin{array}{lll}
\textbf{ovar} & : & \mathit{Reservation} : \mathbb{P}\mathit{ObjectTypeof}(\mathit{Reservation}); \\
 & & \mathit{Loan} : \mathbb{P}\mathit{ObjectTypeof}(\mathit{Reservation}); \\
 & & \text{Makes} :< \mathit{User}, \mathit{Reservation} >; \\
 & & \text{IsOn} :< \mathit{Reservation}, \mathit{Publication} >; \\
 & & \text{IsHeldfor} :< \mathit{Copy}, \mathit{User} >; \\
\textbf{Pre} & : & r \in \mathit{Reservation} \wedge u \in \mathit{User} \\
 & \wedge & \exists c \in \mathit{Copy}, p \in \mathit{Publication} \bullet \text{IsHeldfor}(c, u) \wedge \text{IsCopyof}(c, p) \wedge \text{IsOn}(r, p) \\
 & & \text{/**}u \text{ made the reservation } r \text{ and } r \text{ is on a } p \text{ of } c \text{ **/} \\
\textbf{Post} & : & \exists \ell : \mathit{ObjectTypeof}(\mathit{Loan}) \bullet \ell \notin \mathit{Loan} \\
 & \wedge & \mathit{Loan}' = \mathit{Loan} \cup \{\ell\} \qquad \text{make a new loan} \\
 & \wedge & \text{Borrows}' = \text{Borrows} \cup \{< \ell, c >\} \qquad \text{record } c \text{ in loan } \ell \\
 & \wedge & \text{Takes}' = \text{Takes} \cup \{< u, \ell >\} \qquad \text{record } u \text{ in loan } \ell \\
 & \wedge & \text{Reservation}' = \text{Reservation} - \{r\} \qquad \text{remove the reservation } r \\
 & \wedge & \text{Makes}' = \text{Makes} - \{< u, r >\} \qquad \text{break link with the removed object } \ell \\
 & \wedge & \text{IsOn}' = \text{IsOn} - \{< r, p >\} \qquad \text{break link with the removed object } \ell \\
 & \wedge & \text{IsHeldfor}' = \text{IsHeldfor} - \{< c, u >\} \qquad \text{break link with the removed object } \ell
\end{array}
$$

All above three use cases modify variables $\mathit{Reservation}$ and its related associations. The definition for use case $\mathit{CollectReservation}$ also indicates sometimes when we remove an object, all links with this object should be removed too. In fact, this is ensured by the formal definition of conceptual models in [LHL01]. We can introduce an operation $\mathit{Destroy}(o : C)$ that removes object o from C together will all links to o. However, we explicitly write out the removal of each link for the sake of clarity in this paper. The following state constraints should be preserved to be conjoined into the conceptual model M_5:

(I_{54}).	IsHeldfor $\circ$ IsOn $\subseteq$ IsCopyof	the copy that is held for a reservation is a copy of the publication that is reserved
(I_{55}).	$\mathit{Reservation} = \text{Makes}(\mathit{User}) \wedge$ $\mathit{Reservation} = \text{IsOn}^{-1}(\mathit{Publication})$	every reservation made by a user must be one on a publication in the library.

Association $\mathit{IsHeldfor}$ is related to association $\mathit{IsAvailable}$ and $\mathit{Borrows}$ such that not copy that is held for a reservation should be available, and any copy that is currently on a loan cannot be held for a reservation:

(I_{11}). $\text{IsAvailable}^{-1}(\mathit{User}) \cap \text{Borrows}(\mathit{Loan}) = \emptyset$
(I_{56}). $\text{IsAvailable}^{-1}(\mathit{User}) \cap \text{IsHeldfor}^{-1}(\mathit{Reservation}) = \emptyset$
(I_{57}). $\text{Borrows}(\mathit{Loan}) \cap \text{IsHeldfor}^{-1}(\mathit{Reservation}) = \emptyset$
(I_{58}). $\text{IsAvailable}^{-1}(\mathit{User}) \cup \text{Borrows}(\mathit{Loan}) \cup \text{IsHeldfor}^{-1}(\mathit{Reservation}) = \mathit{Copy}$

Recall that I_{11} was already given after the definition of $\mathit{LendCopy}$ use case requiring that no book currently on loan $\mathit{IsAvailable}$. The other three properties say that a copy cannot be available if is held for a reservation and vice versa; a copy on loan cannot be held for a reservation and vice versa; and every copy is either available, or on loan or held for a reservation.

Then we have if a copy is available to a user, it is then available to any user. The analysis of these properties helps a lot when we design the use cases. For example, we can safely introduce a boolean attribute $\mathit{available}$ to Copy to avoid adding links between a copy to all the users.

Now we would like to question ourselves whether every user in the system should be a registered user, i.e. whether the following condition should be an invariant of the system:

$(I_?)$. $\mathit{User} = \text{Registers}(\mathit{lib})$

This depends on what the use case $\mathit{RemoveUser}$ does. It is an invariant If we require $\mathit{RemoveUser}$ will remove the entire record of the user, include his/her historical loan records:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{RemoveUser}[u : User] \triangleq} \\
\textbf{ovar} & : & User : \mathbb{P}ObjectTypeof(User); & \\
& & Loan : \mathbb{P}ObjectTypeof(Loan); & \\
& & \text{Registers} :< Library, User >; & \\
& & \text{Took} :< Copy, User >; & \\
& & \text{Logs} :< Library, Loan >; & \\
& & \text{Borrowed} :< Loan, Copy >; & \\
\textbf{Pre} & : & u \in User & u \text{ is in the system} \\
& \wedge & \text{Takes}(u) = \emptyset & u \text{ does not hold any item on loan} \\
\textbf{Post} & : & User' = Copy - \{u\} & \text{Remove } u \\
& \wedge & Loan' = Loan - \text{Took}(u) & \text{remove all loans of } u \\
& \wedge & \text{Took}' = \text{Took} - \{< u, l >: l \in Loan\} & \text{Break all links of } u \\
& \wedge & \text{Borrowed}' = \text{Borrowed} - \text{Took}(u) \times Copy & \\
& \wedge & \text{Logs} = \text{Log} - Library \times \text{Took}(u) &
\end{array}
$$

This is the same as $Destroy(u : Cuser)$

However, consider $RemoveUser(u : User)$ as to remove u from the association $Registers$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{RemoveUser}_1[u : User] \triangleq} \\
\textbf{ovar} & : & \text{Registers} :< Library, User > & \\
\textbf{Pre} & : & u \in Register(lib) & u \text{ is registered} \\
& \wedge & \text{Takes}(u) = \emptyset & u \text{ does not hold any item on loan} \\
\textbf{Post} & : & \text{Registers}' = Registers - \{< lib, u >\} &
\end{array}
$$

Then, this definition of $RemoveUser_1$ does not preserve property $I_?$, and the set of users in the system is divided into two sets of *active* users and *inactive* users. Similarly, we can reconsider property I_{22} that asserts ath all publications are registered and again depends on the requirement of $RemovePublication$. If we insist that I_{22} is required, $RemovePublication$ has to be defined as $Destroy(p : Publication)$.

4.1 Introducing subclasses

Now consider how we can introduce subclasses $StaffUser$ and $StudentUser$ of $User$, subclasses $Book$, $Periodical$ and $Report$ of $Publication$. In terms of conceptual models, we can add the diagram in Figure 4 to the diagram in model M_5 that we have created for the library so far.

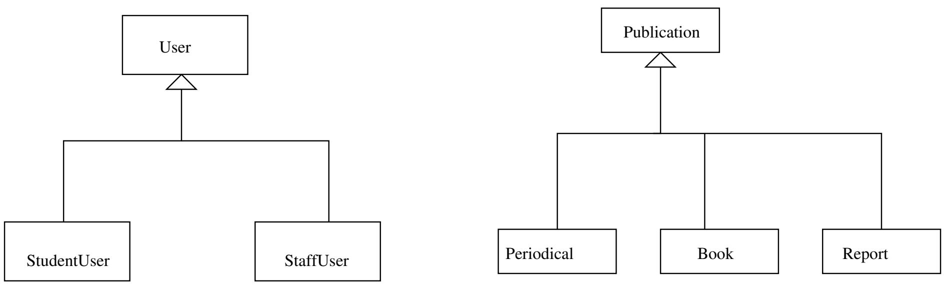

Figure 4: Subclasses in the library system

However, we have to reconsider the use cases $LendCopy$ and $MakeReservation$ to impose conditions on which kinds of users are allowed to borrow what kinds of publications. This can be done by constraining or defining the association $IsLendableTo$, that was already introduced when gave the definitions to those two use cases. The association $IsLendableTo$ was not purely for the introduction of subclasses at this stage, because it is entirely justifiable that the library should have the right to make a policy about who can borrow what, and the policy can be changed from time to time. We did not want to rewrite the specification of the use cases when this policy is changed, but only to change the definition. For example, the library may decide that only registered users can use the library, and a $StaffUser$ can borrow any kind of publications but a $StudentUser$ can only borrow $Book$s. This policy can be defined by

$$(I_{59}).\ \text{IsLendableTo} \triangleq \begin{array}{ll} & \{< p, u >: p \in Publication \wedge u \in Rigisters(lib) \\ \wedge & (p \in Book \vee u \in StaffUser)\} \end{array}$$

Of course, these subclasses could have been captured when we considered use cases for lending a copy and making a reservation. Or we can go back to redefine these use cases when we found they are needed, but we prefer to the way that we have proposed here, the other alternatives do not make the $IsLendableTo$ condition not unnecessary. For example, when we allow a user to borrow at most 5 items, we can also redefine this condition. Another approach in dealing with situations when some concept similar to a policy has to be considered is to introduce a class called $Policy$ and use a parameter x of this type in the use cases like $LendCopy$ and $MakeReservation$ so that a copy will be lent to a user or a user reserves a publication according to a given policy.

In general, when subclasses of a class are introduced, the use cases in which this *direct* super class is involved should be re-considered. This is because in the design stage the part of the work of the use case carried out by a method of the super class may be overwritten in the subclasses. Some people [Lar98] suggest *generalization-specialization* hierarchy is more useful in the design rather than in the requirement analysis. We think, it is still important for the analysis of the system structure and state space. New use cases may be introduced for subclasses of a class. Introducing a super class for some classes does not affect the definitions of the use cases at the requirement analysis, though combinations of smaller use cases into an abstract or big ones can be carried done [LHL01].

If we start the system by a $StartUp$ use case that creates the lib object of $Library$. We can then apply $AddPublication$, $AddUser$ and $AddCopy$ for a number of times before the system can serve the other use cases. All the state constraints listed in this sections should be invariants of the system after its starting up.

4.2 About the state constraints

The state constraints have been imposed one by one after we introduce each use case, but we did not carry out the verification of these state constraints. However, everytime when a use case is introduced, we should prove that it preserves each state constraint I by proving

$$Pre \wedge I \Rightarrow Post \wedge I'$$

for the newly introduced use cases, where I' is the predicate formula obtained from I by replacing each free variable with it primed version. On the other hand, for each newly introduced state constraint I, we should verify that I is preserved by each use case that has been defined. Otherwise, the system will be in danger of violating some requirements. For example, if required that $I_?$ be an invariant that all users in the system must be registered but defined the use case for removing a user by $RemoveUser_1$ without checking $I_?$ against it, I_1 will not be met by the system. This shows the iterative ckecking on the use cases and state constraints are needed.

Writing out these state constraints is a very important part of the requirement analysis, as they will be crucial for the design and implementation of the system. For example, we can decide to use a boolean attribute to represent $IsAvailable$ association as if a copy is available to one user it is the available to any user; we can introduce a boolean variable $active$ to represent the state of a user.

5 Conclusion & Discussion

Use the library system in [Ken97], this paper has demonstrated a use case-driven, incremental and iterative requirement analysis supported by a simple formal semantic model of the conceptual model and use cases proposed in our recent work [LHL01]. The difference between the work presented here from that in [Ken97] is that we aim to provide a formal justification for the informal requirement analysis process used in [Lar98, JBR99, Liu01]. We have shown how a conceptual model can be derived by writing the formal specification of the use cases, one-by-one. By doing so, we have identified the classes, associations and state constraints systematically with formal justifications. The classes and associations identified are not and do not have to be entirely the same as those in [Ken97], and the differences were justified. We have identified a set of state constraints (i.e. invariants) which we believe together with the conceptual diagram itself are enough for designing the identified use cases.

A use case is defined in terms of its pre and postconditions, where the post condition is mainly about what new objects created, old objects deleted, new links added to associations and old links deleted from associations. The preconditions and postconditions of use case and state constraints are written in relational algebra and quite easy to understand. However, writing them out is very important for the understanding of the functional requirement of the system. We believe that any software engineer equipped with discrete mathematics should be able to do the requirement analysis in this semantic model.

When more associations are introduced among old classes by some new use cases, use case which have already defined may need to be reconsidered and slightly modified, and more constraints about old associations may have to be introduced. However, the use cases and constraints are easy to be located to only those that are affected by the newly introduced use cases. The conceptual model and the use case model constructed this way can be guaranteed to be consistent. All the constraints introduced during the development of the use cases and conceptual model are invariants of the system and will be preserved by the use cases. The invariants will be very useful in the system design in the next stage of the development.

When a use case is to be refined further, more classes and associations will be introduced. For example, if we want to record the time of a loan and it return time of the borrowed item, we need to introduce the concept of time into the conceptual diagram. On the other hand, more use cases, such as $RemoveCopy$ can be defined under an existing conceptual model.

The experience we have learnt through this formal analysis is that a formal method can be better used in such a OO use case driven, incremental and iterative analysis process than in the functional and traditional structural analysis; and its use does help to pin-points the main difficulties that are likely to be encountered in the later development stages. Wrting out the formal definitions of the use cases and checking them against the state constraints in an interative way have helped us to discover quite a few implicitly assumed state constraints, and to correct several mistakes about the state constraints. For example, the introduction and understanding of $LendableTo$ association, the problems with regard to use cases of removing a publication and removing a user as well as the problems of $I_?$ and I_{22}. Taking class names, association names as state variables, and the conceptual class diagram as a big system variable has enabled us to avoid from introducing new semantic notions and theories for object-oriented requirement analysis and the very classical state-based relational semantics [HH98] is adequate.

The work in [LHL01] and this paper is only a starting point of our ongoing 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 a whole framework in an incremental manner so that the complexity will not become overwhelming.

In this paper, we deliberately overloaded notations in typing to shrink the size of the paper. For example, we used $c : C$ sometimes for $c : ObjectTypeof(C)$ and $A :< C1, C_2 >$ to denote $\mathbb{P}(ObjectTypeof(C_1) \times$

$ObjectTypeof(C_2)$. We need to clear notation used in the typing system.

References

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

[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.

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

[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.

[LHL01] Z. Liu, J. He, and X. Li. Towards a formal use of UML for software requirement analysis. Technical Report UNU/IIST Report No 228, UNU/IIST: International Institute for Software Technology, The United Nations University, P.O. Box 3058, Macau, 2001. A short version is submitted for presentation at PDPTA2001, June, Las Vegas, USA.

[Liu01] Z. Liu. Object-oriented software development in uml. Technical Report UNU/IIST Report No. 228, International Institute for Software Technology, The United Nations University (UNU/IIST), P.O. Box 3058, Macau, SAR, P.R. China, March 2001.

[LLH01] X. Li, Z. Liu, and J. He. Formal and use use-case driven requirement analysis in UML. Technical Report UNU/IIST Report No 230, UNU/IIST: International Institute for Software Technology, The United Nations University, P.O. Box 3058, Macau, 2001. A short version is submitted for presentation at COMPSAC2001, October, USA.