Consistency Checking of UML Requirements

Xiaoshan Li
Faculty of Science and Technology
University of Macau, Macao, China
E-mail: xsl@umac.mo

Zhiming Liu and Jifeng He
International Institute for Software Technology
United Nations University, Macao, China.
E-mail: {lzm, hjf }@iist.unu.edu

Abstract

This paper discusses how to check consistency of UML requirements model which consists of a use case model and a conceptual class model with system constraints. Based on a given semantics, the requirements consistency can be defined and checked formally. The consistency among use cases and constraints are classified into five types. A system operation of interaction between actor and system is formally defined as a pair of pre and post conditions. An atomic use case is described as one system operation, and a composed use case may be defined as several system operations described by an activity diagram. Thus, each use case can also be modelled as a pair of pre and post conditions by composing the pre and post conditions of system operations by introducing a sequence composition operation. Requirement consistency can be logically checked based on the semantics. A simple library system is used as a case study to illustrate the feasibility of the method.

Keywords: *Consistency Checking, Formal Requirement Specification, Requirement Analysis, UML*

1 Introduction

Once a requirement specification is given, an important question to ask is whether the requirement is correct. The activity to answer this question is called requirements validation in requirements engineering [8, 17]. We are concerned with two aspects about requirements correctness. The first is that the requirements are what the customer really wants. The other is that the requirements must be *consistent*.

Requirements consistency checking is to one part of requirements validation activity. As we all know, fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error [8]. Although requirements informal reviewing and prototyping are useful validation techniques, however, some implicit errors and conflicts in requirements should be checked formally based on the system requirement specification.

Currently, UML [1] and Rational Unified Process (RUP) [6] are widely used to model and develop software systems in software industry. Therefore, the work on how to check the consistency of requirements which are modelled by UML use cases and constraints, is necessary and useful. The requirements consistency can be classified into five types which include consistency of each use case itself, consistency of each constraint itself, consistency of constraints, consistency between use cases and constraints, consistency between use cases.

As the first activity of software development process, requirements engineering is to formulate the different stakeholders' informal needs into the formal requirements specification. Thus, it makes requirements validation difficult. However, the conflicts should be eliminated in the final formal requirements specifications. Many books on requirements engineering and software engineering [8, 18, 17] discussed requirements consistency checking informally using reviewing method with conflict identification table and interaction matrix. Although Heitmeyer's paper [4] discussed consistency checking formally, it used SCR (Software Cost Reduction) requirements table method for specifying real-time embedded systems. Most of consistency checking works on UML models focus on the lower design models which are sequence diagrams and state diagrams. J. H. Hausmann's paper [2] used graph transformation method to detect of conflicting functional requirements in a use case driven approach, however, the constraint was not included.

We presented a formal model in [10, 13] that relates a UM to a CM in the framework of Hoare and He's Unifying Theory of Programming[5]. However, the requirements consistency problem didn't discussed in [10]. In this paper, the consistency among use cases and constraints are classified into five types. A system operation of interaction between actor and system is formally defined as a pair of pre and post conditions. This comes from the method of from B. Meyer's "Design by Contracts" [14, 16]. An atomic use case is described as one system operation, and a composed use case may be defined as several system operations de-

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

scribed by an activity diagram. Thus, each use case can also be modelled as a pair of pre and post conditions by composing the pre and post conditions of system operations by introducing a sequence composition operation. Requirement consistency can be logically checked based on the semantics. A simple library system is used as a case study to illustrate the feasibility of the method.

A model of requirements consists of a *use-case model* (UM) and a *conceptual class model* (CM). A UM consists of a use case diagram and textual descriptions of use cases. A CM for an application is a class diagram consisting of *classes* (also called *concepts*), and *associations* between classes. A class represents a set of *conceptual objects* and an association determines how the objects in the associated classes are related (or *linked*). A library system is used as a case study [10] to show how the formalization in [12] for UML conceptual model 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 method is expected to be usable within an incremental and iterative development process driven by use cases [6], but it didn't present how to check the consistency between use cases and constraints.

The rest of the paper is organized in the following way. Section 2 gives a brief summary of formal model of UML requirements [12]; Section 3 defines five types of consistencies formally and discusses how to check each type of requirements consistencies. Section 4 first gives the formal requirements model of a library case study, and then checks the consistencies to illustrates the feasibility of the method. And finally Section 5 concludes the paper with some discussions.

2 Conceptual Model and Use-Case Model

The main UML models to be produced at the requirement analysis are a *use-case model*(UM) and a *conceptual model* (CM). 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 requirements. A main part of the CM is a conceptual class diagram (CCD) that describes a set of concepts by *class names*, and how these classes are related by *associations*. This section summarizes the formal syntax and semantics of these modelling notations and for details we refer the reader to [12].

We assume an infinite set $\mathbf{O}$ of *objects*, two infinite and disjoint sets of *class names* $CName$ and *association names* $AName$, that are disjoint with $\mathbf{O}$. For each $A \in AName$ there is a distinct name $A^{-1} \in AName$ (for reverse navigation), and $(A^{-1})^{-1} = A$. Let $\mathbf{N}$ denote the set of all the natural numbers, and $\mathbb{P}\mathbf{N}$ be the power set of $\mathbf{N}$. And M_1 and M_2 are the multiplicities of an association A, with the syntactic forms such as $*$, 1, 0..1, and 1..$*$ in UML, and in semantic level, M_1 and M_2 are sets such as $\mathbf{N}$,$\{1\}$, $\{0, 1\}$ and $\mathbf{N}^+$.

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

- $\mathcal{C}$ is a nonempty finite subset of $CName$, called the *classes* or *concepts* of the diagram D.
- $\mathcal{A}$ a partial function:

  $$\mathcal{C} \multimap [AName \multimap \mathbb{P}\mathbf{N} \times \mathbb{P}\mathbf{N} \times \mathcal{C}]$$

  such that $\mathcal{A}(C_1)(A) = \langle M_1, M_2, C_2 \rangle$ iff $\mathcal{A}(C_2)(A^{-1}) = \langle M_2, M_1, C_1 \rangle$. We use $A : \langle C_1^{M_1}, C_2^{M_2} \rangle$ as a shorthand for $\mathcal{A}(C_1)(A) = \langle M_1, M_2, C_2 \rangle$, and require that only finitely many association names defined for a pair of class names.
- $\lhd\!\!- \subseteq \mathcal{C} \times \mathcal{C}$ is the generalization relation between classes, we use $C_1 \lhd\!\!- C_2$ to denote $(C_1, C_2) \in \lhd\!\!-$. We require that the generalization is acyclic.

2.1 Semantics of a CCD

A CCD models the conceptual objects and their associations in an application domain. In a given application domain, a class $C \in \mathcal{C}$ models a set $\mathbf{C} \subset \mathbf{O}$ of *domain objects*, and an association $A \in AName$ such that $A : \langle C_1^{M_1}, C_2^{M_2} \rangle$ models an relation between the objects in $\mathbf{C}_1$ and $\mathbf{C}_2$, and thus A is interpreted as a subset $\mathbf{A}$ of $\mathbf{C}_1 \times \mathbf{C}_1$. The generalization symbol $\lhd\!\!-$ is semantically defined as the superset relation between the classes of objects, i.e. $\mathbf{C}_1 \supseteq \mathbf{C}_2$ if $C_1 \lhd\!\!- C_2$ and the "superseteq" relation, $\supseteq$, between classes is the semantics of the reflexive and transitive closure $\lhd\!\!-^*$ of $\lhd\!\!-$. We say that $\mathbf{C}_1$ is a *subclass* of $\mathbf{C}_2$ if $C_2 \lhd\!\!-^* C_1$.

For a relation $R \subseteq S_1 \times S_2$, we use $R(s_1)$ for s_1 in S_1 (or S_2) to denote the set of elements in S_2 (or S_1 respectively) that are associated with s_1 by R. We use $\mid S \mid$ to denote the cardinality of a set S; we allow to write $R(s_1, s_2)$ or $s_1 R s_2$ to denote $(s_1, s_2) \in R$. The semantic denotations of the class and association names satisfy the following conditions:

1. $\mathbf{A} \subseteq \mathbf{C}_1 \times \mathbf{C}_2$;
2. $\mathbf{A} \circ \mathbf{A}^{-1} \subseteq Id$, where $\circ$ is the *composition* operation on relations, and Id is the identity relation;
3. $\mathbf{C}_1 \supset \mathbf{C}_2$ if $C_1 \lhd\!\!- C_2$;
4. $\forall c_1 : \mathbf{C}_1\ c_2 : \mathbf{C}_2 \bullet \mid \mathrm{A}(c_1) \mid \in M_2 \wedge \mid \mathrm{A}(c_2) \mid \in M_1$;

5. for all C_1 and C_2 in $\mathcal{C}$, either $\mathbf{C}_1 \cap \mathbf{C}_2 = \emptyset$ or one of $\mathbf{C}_1$ and $\mathbf{C}_2$ is the subclass of another.

Note that Conditions 3&5 implies that as in Java, multiple inheritance is not allowed in this formalization.

In addition to the types assigned to the names in a CCD, we assume a collection of *types of pure data values/objects* that are used as values of objects' attributes, and a set Var of variable names. We write $x : \mathbf{T}$ for a declaration of either a state or a logical variable x with its type $\mathbf{T}$.

When the dynamic aspects of a system is considered, a CCD represents a state of the system which is a snapshot recording the existing objects of each class and the current links between objects by the associations at a moment of time. Therefore, a CCD declares

- a *system state variable* C for each $C \in \mathcal{C}$ that records the *set* of existing objects of type $\mathbf{C}$, and the type of C is $\mathbb{P}\mathbf{C}$.
- a *system state variable* A for each $A \in AName$ such that $A : \langle C_1^{M_1}, C_2^{M_2} \rangle$ recording the current links between the existing objects recorded in C_1 and C_2; its type is thus $\mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2)$.

2.2 An object diagram is a system state

An object diagram for a CCD is a particular instance of the pattern that is modelled by the CCD.

Definition 2 (**Object Diagram**) Given a CCD D, an *object diagram* $\mathcal{O}_D$, which also called a *state* of D, is an evaluation function that assigns:

- each $C \in \mathcal{C}$ a set $\mathcal{O}_D[C] \in \mathbb{P}\mathbf{C}$;
- each A such that $A : \langle C_1^{M_1}, C_2^{M_2} \rangle$ a relation $\mathcal{O}_D[A] \in \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2)$.

that satisfies the following four assertions

1. $A \subseteq C_1 \times C_2$;
2. $A \circ A^{-1} \subseteq Id$, where Id is the identity relation on the domain of A.
3. $C_1 \supseteq C_2$ if $C_1 \lhd\!\!- C_2$;
4. $\forall c_1 : \mathbf{C}_1\ c_2 : \mathbf{C}_2 \bullet \mid A(c_1) \mid \in M_2 \wedge \mid A(c_2) \mid \in M_1$;
5. for all C_1 and C_2 in $\mathcal{C}$, either $C_1 \cap C_2 = \emptyset$ or one of $\mathbf{C}_1$ and $\mathbf{C}_2$ is the subclass of another.

Note that Condition 4 can be replaced by the type condition $A \subseteq \mathbf{A}$, and that Condition 5 can be derived from the type condition Condition 5 in the previous page. However, we may prefer checking these conditions as state invariants rather than type checking. We call the conjunction of the four assertions in Definition 2 the *generic invariant* of CCDs, and denote is as $\mathcal{I}$.

Definition 3 (**Semantics of a CCD**) The *semantics* of a CCD D is the set of all the states of D, denoted by Σ_D

Only classes, associations, and their multiplicities are not enough to express all the constraints that application requires. We need to introduce the notion of *state constraints* which are invariants of the system during its execution. In general, a *well defined state constraint* on a CCD D is a first order predicate formula with types and free variables declared in D.

Definition 4 (**CM**) A CM is a pair $M = \langle D, Inv \rangle$, where D is a CCD and Inv is the state constraint of whole system that is well-defined on D. And we define $\mathcal{I}_M \triangleq \mathcal{I} \wedge Inv$.

Definition 5 (**Semantics of a CM**) The semantics of a CM $M = \langle D, Inv \rangle$ is the set, denoted by Σ_M, of all the object diagrams of D that also satisfy Inv.

2.3 Use cases

In [12], an atomic use case is defined to be a *parameterized joint action* of the following form:

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

where Act is an action name, $\overline{pvar}$ a list of parameters typed with classes or data types, and $\overline{ovar}$ denotes a list of typed variables that can be modified by the action and is called the *frame* of the action. 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. The postcondition $Post$ of the action describes the values of the post-state after the action is carried out.

To describe a postcondition, we introduce a primed version x' of each state variable x in Var that always has the same type as that is declared for x and represents the final value of state variable x after an action is carried out. An assertion P containing primed state variables (as well as unprimed ones) is to be interpreted as a relation over pairs $(\mathcal{O}, \mathcal{O}')$ of states of Var such that P is true of $(\mathcal{O}, \mathcal{O}')$ if $P[\mathcal{O}[Var]/Var, \mathcal{O}'[Var]/Var']$ holds. We sometimes omit the frame and assume that only the variables with their primed versions occurring in the postcondition may be modified.

A use case may involve a number of atomic use cases. As suggested in [12], composite use cases made up from atomic use cases, such as sequential composition, guarded choice, iteration and parallel composition, are defined in a notation in [5]. These are enough to model the *extend* and *include* relationships between uses cases in UML, and the repetition of a sequence of actions that is not defined in use case model of UML.

2.4 Semantics of use cases

The semantics of a use case has to be defined under a CM $M = \langle D, Inv \rangle$ as its environment. We only define the semantics for an atomic use case, and the semantics for a composite use case is defined by the semantics of the compositions given in [5].

Definition 6 (**Well definedness**) Given a use case $Act \triangleq [\overline{pvar};\ \overline{ovar}] \bullet Pre \vdash Post$ and a CM $M \triangleq \langle D, Inv \rangle$. We define $WD_M(Act)$ to be *true* if all the types used in the signature of Act are declared in D and all the free variables in Pre and $Post$ are declared in either in D or in the signature of Act.

Definition 7 (**Semantics of a use case**) Given a use case $Act \triangleq [\overline{pvar};\ \overline{ovar}] \bullet Pre \vdash Post$ and a CM $M \triangleq \langle D, Inv \rangle$. The semantics $[\![Act]\!]_{M(s,s')}$ of Act under M is defined as

$$WD_M(Act) \Rightarrow$$
$$(Pre(s) \wedge ok \Rightarrow Post(s, s') \bigwedge_{x \notin \overline{ovar}} (x' = x) \wedge ok')$$

where ok is a logical state variables which represents that the program is in a proper (i.e. an *ok*) state s to start the execution of the action [5] and terminate at the state s'.

As for the different composition of use cases, it will be defined and discuss in part of consistency between use cases in next section.

3 Requirements Consistency Checking

Based on the formal semantics of UML requirements model given in Section 2, requirements consistency can be defined precisely. In this section, we will present four types of requirements consistencies among use cases and constraints. By introducing the dependent degrees of use cases and constraints, we develop a method on checking requirements consistency formally.

Obviously, the task of requirements consistency checking is to check whether there is a conflict between two related requirements. Traceability tables are used in [8] to show requirement dependencies. According to UML requirements model, we can divide dependence relationships between two use cases into "Dependency" and "Non-Dependency", which are decided by the relationship of read and write variable sets of two use cases. The definition is given as follows.

Definition 8 The read-variable set of a use case U or an invariant I contains these variables appearing in specification formula, denoted as $Rd(U)$ or $Rd(I)$. And the write-variable set of a use case U, denoted as $Wt(U)$, contains of these variables appearing in its post condition with prime, whose values are modified when accessing the use case.

Definition 9 Two kinds of dependent relationships between two use cases U_1 and U_2 can be defined as follows:

Dependency : if $Wt(U_1) \cap Rd(U_2) \neq \emptyset$ or $Wt(U_2) \cap Rd(U_1) \neq \emptyset$.

Non-Dependency: if $Wt(U_1) \cap Rd(U_2) = \emptyset$ and $Wt(U_2) \cap Rd(U_1) = \emptyset$.

Similarly, we can define the two kinds of impact relationships between a use case U and a constraint I. A constraint can be viewed as a query use case whose write-variable set always is empty.

Definition 10 Two kinds of impact relationships between a use case U and a constraint I.

Impact: if $Wt(U) \cap Rd(I) \neq \emptyset$.

Non-Impact: if $Wt(U) \cap Rd(I) = \emptyset$.

Requirement conflicts can only happen on the use cases and constraints with "Dependence" and "Impacted" cases. The requirements consistency of UML requirements model can be classified into five types, which are listed as follows.

1. consistency of each use case itself.
2. consistency of each constraint itself.
3. consistency of constraints.
4. consistency between use cases.
5. consistency between use cases and constraints.

For type 1 and 2, the consistency of a use case or constraint means that its specification should be well-defined in syntax. That is to say, $WD_M(U)$ and $WD_M(I)$ should be "true" on CM M for use case U and constraint I. We also say that a CM M is *adequate* for defining an atomic use case U if $WD_M(U)$ is "true" and Pre is satisfiable by the state space Σ_M. M is adequate for defining a composite use case U if it is adequate for defining all the atomic use cases of U. And each constraint I should be satisfiable, too.

Furthermore, the consistency of all constraints means that for a CM, the system state constraint $\mathcal{I}_M$ should be satisfiable by the state space Σ_M.

Consistency between use cases: We should check the consistencies between two or more use cases when we do composition of use cases. Three basic kinds of compositions are sequential, parallel or conditional choice, shown in figure 2. UML activity diagram is used to describe the workflow of use cases. These compositions are required by system application business operation rules [15]. However, whether the

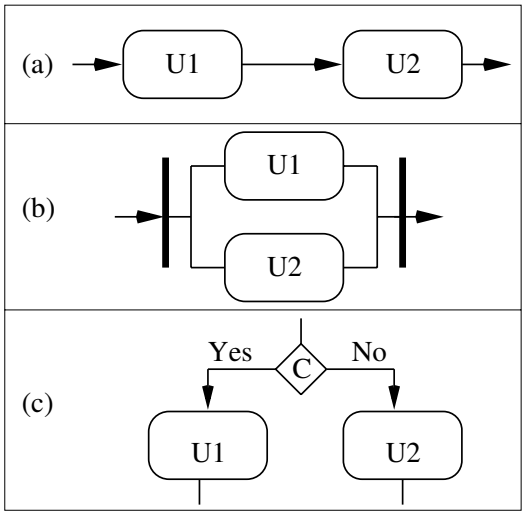

Figure 1. Three kinds of activity compositions

compositions are allowed without any conflicts on semantics, is the requirements consistency problem. If there does not exist a pair of system states which make the composition satisfiable, then it is a conflict. The semantics of sequence, parallel and conditional choice compositions shown in figure 2 can be defined formally as follows.

$$[\![U_1;\ U_2]\!]_{M(s,s')} \triangleq \exists m.([\![U_1]\!]_{M(s,m)} \wedge [\![U_2]\!]_{M(m,s')} \wedge (Post_1(s,m) \Rightarrow Pre_2(m)))$$

$$[\![U_1 \mid\mid U_2]\!]_{M(s,s')} \triangleq [\![U_1]\!]_{M(s,s')} \wedge [\![U_2]\!]_{M(s,s')}$$

$$[\![\text{If } C \text{ Then } U_1 \text{ Else } U_2]\!]_{M(s,s')} \triangleq C(s) \wedge [\![U_1]\!]_{M(s,s')} \vee \neg C(s) \wedge [\![U_2]\!]_{M(s,s')}$$

where use cases U_1 and U_2 are $(Pre_1 \vdash Post_1)$ and $(Pre_2 \vdash Post_2)$, and

$$\begin{array}{rl} & (Pre_1 \vdash Post_1) \mid\mid (Pre_2 \vdash Post_2) \\ = & (Pre_1 \wedge Pre_2 \vdash Post_1 \wedge Post_2) \end{array}$$

The above semantics model is also suitable to system operations in activity diagram if U_1 and U_2 are system operations (activities or actions). Obviously, we have the following theorem from above semantics of compositions;

Theorem 1 If U_1 and U_2 are "Non-Dependency", then their sequence and parallel compositions have no consistency conflicts.

Thus, we can only pay our attention to check the consistency between two use cases with "Dependency".

Another kind consistency among two or more use cases is that they cannot be enabled at the same time with some particular input parameters. It means that two preconditions can not be satisfied at that situation. It can be defined as follows:

$$\exists s, \overline{pvar}_1, \overline{pvar}_2 \bullet (Pre(U_1(\overline{pvar}_1))(s) \wedge Pre(U_2(\overline{pvar}_2))(s) = false)$$

Consistency between use cases and constraints: For this type of consistency, it requires any executions of use cases should preserve the system constraint $\mathcal{I}_M$. It is equivalent to the parallel composition with the constraint that can be considered as a query use case (operation)[14, 16] without modifying any variables. We specify an atomic use case in the form as follows.

$$Act \triangleq [\overline{pvar};\ \overline{ovar}] \bullet (Pre \vdash Post) \mid\mid (\mathcal{I}_M \vdash \mathcal{I}'_M)$$

where $\mathcal{I}'_M$ is obtained from $\mathcal{I}_M$ by replacing each variable in $\mathcal{I}_M$ by its primed version.

When a use case has an "Impact" relationship with a constraint, it may possibly destroy the constraint invariant after its execution. However, by *theorem 1*, the consistency of a use case with those 'Non-Impact' constraints does not need to check.

4 Requirement Analysis and Consistency Checking of the Library System

This section first uses the formalization to carry out the requirement analysis for the library application, and then check the requirements consistency formally. We intend to illustrate the feasibility of the formalization.

4.1 Informal description of a simple library system

The simple 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. Each user can at most borrow 10 copies. 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. 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, [7] identified the following use cases which should be supported by the system.

1. Librarian maintains the library, such as adds and removes publications, copies, and users.

2. Library lends a copy to a user, and user returns a copy.

3. User makes and removes a reservation.

4.2 The specification and analysis

The relationship between the use-case model and the conceptual model is very close in an application. On the one hand, the identification and description of the use cases provide important information about what should be in the CM. 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 is similar to the technique of *noun-phrase identification* for the creation of a conceptual model from a use case [9, 6].

On the other hand, the effect of a use case can only be defined in the context of a CM in terms what objects should be created and deleted, and which and how associations between objects including attributes of objects are changed. We can rigorously check whether a given CM is adequate to realize a use case, and extend the CM by adding classes and associations or softening the state constraint to accommodate a use case. The CM will be extended while further use cases are captured and defined. This provides supports to an *incremental and iterative process* for the requirement analysis.

We now specify and analyze the use cases that were identified in Section 4.1. The classes and associations can be extracted from analyzing use case one by one and at same time to develop a conceptual model step by step. Here we omit it. For the details, please refer to [10]. The conceptual diagram is shown in Figure 2, denoted by D.

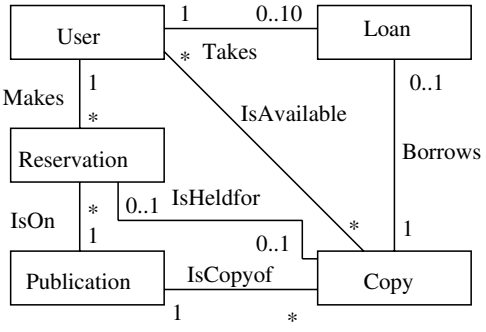

Figure 2. Conceptual model for library system

Use case LendCopy This use case is about how the library can lend a copy of a publication to a user. Obviously, 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. This use case can be formally specified as

$$
\begin{array}{lll}
LendCopy \triangleq \textbf{pvar } c : \textbf{Copy}, u : \textbf{User}; & & \\
\textbf{Pre} & : & c \in Copy \wedge u \in User \wedge IsAvailable(c, u) \\
& \wedge & \mid Takes(u) \mid < 10 \\
\textbf{Post} & : & \exists \ell : \textbf{Loan} \bullet \ell \notin Loan \\
& \wedge & Loan' = Loan \cup \{\ell\} \\
& \wedge & Borrows' = Borrows \cup \{< \ell, c >\} \\
& \wedge & Takes' = Takes \cup \{< u, \ell >\} \\
& \wedge & IsAvailable' = IsAvailable - \\
& & \{< c, u > \mid u \in User \wedge IsAvailable(c, u)\}
\end{array}
$$

The four preconditions say that c and u are known by the system, c is available to u, and the total number of loans that user u takes is less than 10; and the post conditions assert that a new loan is created to record the loan of c and u, and that c becomes unavailable.

Use case AddPublication The library may add a new publication into the system and this service is provided by a use case called $AddPublication$. The use case $AddPublicaiton$ can be specified as:

$$
\begin{array}{lll}
AddPublication \triangleq \textbf{pvar } p : \textbf{Publication} & & \\
\textbf{Pre} & : & p \notin Publication \\
\textbf{Post} & : & Publication' = Publication \cup \{p\}
\end{array}
$$

The precondition requires that p is not currently in the library; and the postcondition asserts that the publication has been created and that p now belongs to $Publication$.

Use case AddUser Similarly, we can define use case $AddUser$ as follows.

$$
\begin{array}{lll}
AddUser \triangleq \textbf{pvar } u : \textbf{User} & & \\
\textbf{Pre} & : & u \notin User \\
\textbf{Post} & : & User' = User \cup \{u\} \\
& \wedge & IsAvailable' = IsAvailable \cup \\
& & \{< c, u > \mid c \in Copy \wedge IsAvailable(c) \neq \emptyset\}
\end{array}
$$

Use case AddCopy Use case $AddCopy$ adds 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 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:

$AddCopy \triangleq$ **pvar** c : **Copy**, p : **Publication**;
Pre : $c \notin Copy \wedge p \in Publication$
Post : $Copy' = Copy \cup \{c\}$
$\wedge$ $IsCopyof' = IsCopyof \cup \{< c, p >\}$
$\wedge$ $IsAvailable' = IsAvailable \cup$
$\{< c, u > | \, u \in User\}$

Use case MakeReservation When a user u wants to borrow a publication p and there is no copy of this publication available, the system allows the user to make a reservation r on the publication p. Therefore, use case $MakeReservation$ introduces a new class $Reservation$ and two associations *Makes* and *IsOn* among classes $User$, $Publication$ and $Copy$. Use case $MakeReservation$ can be defined formally as follows.

$MakeReservation \triangleq$ **pvar** u : **User**, p : **Publication**;
Pre : $p \in Publication \wedge u \in User$
$\wedge$ $\neg \exists c \in Copy \bullet$
$IsCopyof(c, p) \wedge IsAvailble(c, u)$
$\wedge$ $\neg \exists r \in Reservation \bullet$
$Makes(u, r) \wedge IsOn(r, p)$
Post : $\exists r$: **Reservation** $\bullet$ $(r \notin Reservation \wedge$
$Reservation' = Reservation \cup \{r\})$
$\wedge$ $Makes' = Makes \cup \{< u, r >\}$
$\wedge$ $IsOn' = IsOn \cup \{< r, p >\}$

Notice that more than one reservation can be made on one publication, but not by the same user.

Use case ReturnCopy If a publication is reserved, a copy of it should be held for the reservation when it is returned. We thus need to introduce an association *IsHeldfor*. The use case for returning a borrowed copy can be defined as follows.

$ReturnCopy \triangleq$ **pvar** c : **Copy**
Pre : $c \in Copy \wedge \exists \ell \in Loan \bullet Borrows(\ell, c)$
Post : **Let** $\ell = Borrows^{-1}(c)$ **and**
$u = Takes^{-1}(\ell)$ **and**
$r = IsOn^{-1}(IsCopyof(c))$ **in**
$Takes' = Takes - \{< u, \ell >\}$
$\wedge$ **if** $r \neq \emptyset$ **then** $IsHeldfor' =$
$IsHeldfor \cup \{< c, choice(r) >\}$
else $IsAvailable' = IsAvailable$
$\cup \{< c, u > | \, u \in User\}$

Use case CollectReservation When a copy is held for a reservation, the user of the reservation will go to collect the copy. In [7], 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.

$CollectReservation \triangleq$ **pvar** u : **User**, r : **Reservation**
Pre : $r \in Reservation \wedge u \in User$
$\wedge$ $\exists c \in Copy, p \in Publication \bullet$
/* u made the reservation r and r is on p of c */
$IsHeldfor(c, r) \wedge Makes(u, r) \wedge$
$IsCopyof(c, p) \wedge IsOn(r, p)$
Post : $\exists \ell$: **Loan** $\bullet$ $\ell \notin Loan$
$\wedge$ $Loan' = Loan \cup \{\ell\}$
$\wedge$ $Borrows' = Borrows \cup \{< \ell, c >\}$
$\wedge$ $Takes' = Takes \cup \{< u, \ell >\}$
$\wedge$ $Reservation' = Reservation - \{r\}$
$\wedge$ $Makes' = Makes - \{< u, r >\}$
$\wedge$ $IsOn' = IsOn - \{< r, p >\}$
$\wedge$ $IsHeldfor' = IsHeldfor - \{< c, r >\}$

All above three use cases modify variables $Reservation$ and its related associations.

Use case RemoveReservation Sometimes a user or librarian needs to cancel a reservation from the library. It can be defined as follows.

$RemoveReservation \triangleq$ **pvar** u : **User**, r : **Reservation**
Pre : $r \in Reservation \wedge u \in User \wedge Makes(u, r)$
$\wedge$ $\exists p \in Publication \bullet IsOn(r, p)$
Post : $Reservation' = Reservation - \{r\}$
$\wedge$ $Makes' = Makes - \{< u, r >\}$
$\wedge$ $IsOn' = IsOn - \{< r, p >\}$
$\wedge$ $IsHeldfor' = IsHeldfor -$
$\{< c, r > | \, c \in Copy \wedge IsHeldfor(c, r)\}$

Use case RemoveCopy This use case is to remove a copy c from the library. Before accessing this service, the copy should not be in available state. That is to say, if the copy c is borrowed or is held for a reservation, use case $ReturnCopy$ or $RemoveReservation$ should be accessed first. The formal definition of $RemoveCopy$ can be defined as follows.

$RemoveCopy \triangleq$ **pvar** c : **Copy**
Pre : $c \in Copy \wedge IsAvailable(c) \neq \emptyset$
$\wedge$ $\exists p \in Publication \bullet < c, p > \in IsCopyof$
Post : $Copy' = Copy - \{ c\}$
$\wedge$ $IsCopyof' = IsCopyof - \{< c, p >\}$
$\wedge$ $IsAvailable' = IsAvailable -$
$\{< c, u > | \, u \in User \wedge IsAvailble(c, u)\}$

Use case RemovePublication This use case should first call $RemoveCopy$ to remove all the copies belong to the publication p, and call $RemoveReservation$ to remove all the reservations on the publication p. The formal specification of use case $RemovePublication$ can be defined as follows.

$$
\begin{array}{lcl}
RemovePublication \triangleq \textbf{pvar } p : \textbf{Publication} & & \\
\textbf{Pre} & : & p \in Publication \\
& \wedge & IsCopyof^{-1}(p) = \emptyset \wedge IsOn^{-1}(p) = \emptyset \\
\textbf{Post} & : & Publication' = Publication - \{\, p\}
\end{array}
$$

Use case RemoveUser This use case is to remove a user u from the library. Similarly to use cases $RemoveCopy$ and $RemovePublication$, the precondition of $RemoveUser$ requires that the user should return all the copies he or she borrowed from the library, i.e. delete all links in association $Takes$. Therefore, the user should first call use case $ReturnCopy$ for returning all the copies and use case $RemoveReservation$ for cancelling all the reservations before accessing use case $RemoveUser$. The definition is given as follows.

$$
\begin{array}{lcl}
RemoveUser \triangleq \textbf{pvar } u : \textbf{User} & & \\
\textbf{Pre} & : & u \in User \wedge Takes(u) = \emptyset \wedge Makes(u) = \emptyset \\
\textbf{Post} & : & User' = User - \{u\} \\
& \wedge & IsAvailable' = IsAvailable - \\
& & \{< c, u > |\ c \in Copy \wedge IsAvailable(c, u)\}
\end{array}
$$

4.3 Constraints on system stable states

In general the system state constraints are on association relations, where multiplicities of associations are basic constraints. The following eight state constraints should be preserved under executions of use cases on system stable states, and be conjoined into the conceptual model M of library system. For example, $AddCopy$ 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_1) $\quad Copy = IsCopyof^{-1}(Publication)$

Similarly, the other constraints are listed as follows:

(I_2) a copy is either on loan or held for a reservation, or available

$$
\begin{array}{rcl}
Copy & = & IsAvailable^{-1}(User) \\
& \cup & Borrows(Loan) \\
& \cup & IsHeldfor^{-1}(Reservation)
\end{array}
$$

We thus have if a copy is available to a user, it is then available to any user. The association $IsHeldfor$ is related to association $IsAvailable$ and $Borrows$ by the following invariants:

(I_3) no book currently on loan $IsAvailable$

$$IsAvailable^{-1}(User) \cap Borrows(Loan) = \emptyset$$

(I_4) a copy held for a reservation is not available

$$IsAvailable^{-1}(User) \cap IsHeldfor^{-1}(Reservation) = \emptyset$$

(I_5) a copy on loan cannot be held for a reservation

$$Borrows(Loan) \cap IsHeldfor^{-1}(Reservation) = \emptyset$$

(I_6) each user cannot borrow more than 10 copies, that is a generic invariant of $\mathcal{I}$ on the multiplicity of association $Takes$ on $Loan$

$$\forall u \in User \bullet \mid Takes(u) \mid \leq 10$$

(I_7) the copy that is held for a reservation is a copy of the publication that is reserved

$$IsHeldfor \circ IsOn \subseteq IsCopyof$$

(I_8) every reservation made by a user must be on a publication in the library

$$Reservation = Makes(User) \wedge Reservation = IsOn^{-1}(Publication)$$

4.4 Consistency checking of library requirements

Up to now we have given the formal requirements model of library system with 11 use cases and 8 invariant constraints. Following the method of consistency checking presented in Section 3, we can check the requirements consistency of library system.

Obviously, the well-definedness of use cases and constraints can be easily checked according to the syntax of formal specification by analysts manually or supported by CASE tools.

From semantic viewpoint, any use case and constraint should be satisfiable. It means that for a use case U and system constraint I, we can find a model s (an object diagram) to make $Pre(s) \wedge \mathcal{I}_M(s)$ be $true$ on the conceptual model M. For the library system, it is not difficult to check each use case by giving a simple object diagram. Similarly, all constraints should be satisfied under the object diagram, system state at the same time. For example, $LendCopy(u, c)$ with precondition requires at the current system state s, c is a copy, u is a registered user, c is available, and u takes the loans less than 10. Obviously, it is

easy to give an object diagram that makes the precondition be "true". Meanwhile at current state s, all constraints $I_{1..8}$ are satisfied to be "ture".

Here we focus on checking consistencies between two use cases as well as preserving with constraints.

Consistency of two use cases For the library system, we only concern about the sequence composition of two use cases, such as whether $ReturnCopy$ can follow $LendCopy$. $ReturnCopy$ and $LendCopy$ are high dependent. The formal definition can be defined as follows.

$$\forall u, c \bullet (LendCopy(u, c);\ ReturnCopy(c))$$

According to the specification of $LendCopy$ and $ReturnCopy$, we can prove that the above composition is valid by using the method in Section 3. The proof key point is to check whether the postcondition of $LendCopy(u, c)$ implies the precondition of $ReturnCopy(c)$.

Similarly, we can check the complex sequence compositions of $MakeReservation(u, p)$ following by $ReturnCopy(c)$ and then $CollectReservation(u, r)$ under the condition of non other reservation on the publication the p of c and user u makes the reservation r.

And $LendCopy(u, c)$ and $MakeReservation(u, p)$ cannot be enabled at same time if c is a copy of p. We can prove the following formula is valid.

$$\neg(IsCopyof(c, p) \wedge Pre(LendCopy(u, c)) \\ \wedge Pre(MakeReservation(u, p)))$$

where we use $Pre(U)$ to denote the precondition of use case U. Because $Pre(LendCopy(u, c))$ contains $IsAvailable(u, c)$ which declares that copy c is available, however, $Pre(MakeReservation(u, p))$ contains $\neg IsAvailable(u, c)$, therefore, it makes the formula valid.

Consistency between use cases and constraints For the library system, the conceptual model contains 5 classes and 7 associations which are closely related. We use $LendCopy$ as an example to illustrate how to check its consistency with constraints $I_1..I_8$.

First we can give read and write variable sets of $LendCopy$ from its pre and post conditions.

$$Rd(LendCopy) = \{Copy, User, Loan, Takes, \\ IsAvailable, Borrows\}$$
$$Wt(LendCopy) = \{Loan, Takes, Borrow, \\ IsAvailable\}$$

Then from the read-variable sets of 8 constraints $I_{1..8}$, we can find that only 5 constraints from I_2 to I_6, are high-dependent with $LendCopy$. Use case $LendCopy$ preserves the constraints means the following formula must be valid.

$$Pre(LendCopy(u, c)) \wedge I_2 \wedge \cdots \wedge I_6 \Rightarrow \\ Post(LendCopy(u, c)) \wedge I'_2 \wedge \cdots \wedge I'_6$$

Before checking the consistency between $LendCopy$ and 5 related constraints one by one, we must make clear how the use case modifies the four variables in $Write$-set. From its postcondition, the following equations can be got.

$$E1.\quad Loan' = Loan \cup \{\ell\}$$
$$E2.\quad Borrows' = Borrows \cup \{< \ell, c >\}$$
$$E3.\quad Takes' = Takes \cup \{< u, \ell >\}$$
$$E4.\quad IsAvailable' = IsAvailable - \\ \{< c, u > |\ u \in User \wedge IsAvailable(c, u)\}$$

Check with I_2: From $E1$ and $E2$, we know that a new object ℓ is added to set $Loan'$, which is linked to copy c, so $Borrow'(Loan')$ increases one more element of copy c than $Borrow(Loan)$. But from $E4$, the copy c is removed from $IsAvailable^{-1}(User)$ so that $IsAvailable'^{-1}(User)$ decreases one element of copy c. And $IsHeldfor^{-1}(Reservation)$ and $Copy$ do not change. Therefore, we have

$$Copy = IsAvailable^{-1}(User) \cup Borrows(Loan) \\ \cup IsHeldfor^{-1}(Reservation)$$
$$Copy = Copy'$$
$$IsAvailable^{-1}(User) = IsAvailable'^{-1}(User') - \{c\}$$
$$Borrows(Loan) = Borrows'(Loan') \cup \{c\}$$
$$\frac{IsHeldfor^{-1}(Reserva.) = IsHeldfor'^{-1}(Reserva.')}{Copy' = IsAvailable'^{-1}(User') \cup Borrows'(Loan') \\ \cup IsHeldfor'^{-1}(Reservation')}$$

So, $Pre(LendCopy) \wedge I_2 \Rightarrow Post(LendCopy) \wedge I'_2$

Check with I_3 ***&*** I_4: From above analysis of I_2, obviously we have I'_3 and I'_4 can be guaranteed.

Check with I_5: $IsHeldof$ association does not changed for execution $LendCopy$. From the precondition of $LendCopy$, we know c is available in pre-state. From the precondition of $MakeReservation$, we know that c is not in $IsHeldof^{-1}(Reservation)$ because of all copies held for reservation are not available. Therefore, although $Borrows'(Loan')$ increases one copy c, it still makes I_5 valid at the post state, i.e., I'_5 holds.

Check with I_6: From $E3$ equation, we can get that only $Take'(u)$ increases one element ℓ for user u, and for other user v, $Takes'(v)$ does not change. And because one precondition of $LendCopy$ is $\mid Takes(u) \mid < 10$, so we have

$$I_6\quad \forall u \in User \bullet \mid Takes(u) \mid \leq 10$$
$$User = User'$$
$$u \in User \wedge \mid Take(u) \mid < 10$$
$$\underline{Takes'(u) = Take(u) \cup \{\ell\}}$$
$$I'_6\quad \forall u \in User' \bullet \mid Takes'(u) \mid \leq 10$$

Similarly, we can check other use cases consistent with the constraints from I_1 to I_8.

5 Conclusion & Discussion

We have formally analyzed 5 types of consistency for UML requirements models and developed methods for systematic checking of them. The library system is used as a case to illustrate the feasibility of the method. An important issue in use case analysis is to identify "conflicts" between use cases. Conflicts can be formal identified by checking the state invariants to see whether a newly introduced use case may violate a state invariant; or to check whether the post condition of a use case causes the precondition of another use case to be false after its execution. The formalization supports systematic and precise checking on the consistency among use cases to rule out conflicts in the early stage of system development.

A use case is defined in terms of its pre and post conditions, 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. In this paper, we focus on the consistent aspects of the formal model of requirements. As for validating the functional aspects of requirements, we also develop a prototype generator tool which can generate prototype Java source code automatically from this formal model of requirements [11]. Although this kind formal description of use cases and state constraints may be not easy to understand to some requirement engineers without equipped discrete mathematics and formal method training, however, it is necessary for the understanding of the functional requirements of the system precisely, especially for consistency checking.

As for some use cases are described as the composition of several system operations which is given in an activity diagram. We can specify each system operation as a pair of pre and post conditions, and then compose them as same as handling the composition of use cases in section 3. The itself consistency of the composed use case is to check whether its system operations can be composed in semantic level according to the activity diagram.

Of course, Object Constraint Language (OCL) in [19] and Simple Contract Language (SCL) in [16] can express the assertions of contracts. 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 [5, 3] is adequate for the treatment of refinement and consistency checking between use cases in design models as well.

Future work includes evaluating the method by applying it to requirements analysis of large practical systems.

Acknoledgement: First author thanks Dr. T. Tse from Hong Kong University for his question on requirements consistency when presenting [10] at COMPSAC'2001.

References

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

[2] J. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use-case driven approach: a static analysis technique based on graph transformation. In *Proc of the 24th International Conference on Software Engineering*, pages 105–115, 2002.

[3] J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *Proc of the 2nd Asian Symposium on Programming Language and Systems (APLAS 2004), LNCS 3302*, pages 415–436, Taiwan, November 2004. Springer.

[4] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transations on Software Engineering and Methodology*, 5:231–261, 1996.

[5] C. Hoare and J. He. *Unifying theories of programming*. Prentice-Hall International, 1998.

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

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

[8] G. Kotonya and I. Sommerville. *Requirements Engineering: process and techniques*. John Wiley, 1998.

[9] C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.

[10] X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *Proc of the 25th Annual International Computer Software & Applications Conference (COMPSAC01)*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.

[11] X. Li, Z. Liu, J. He, and Q. Long. Generating a prototype from a UML model of system requirements. In *Proceedings of 1st International Conference on Distributed Computing & Internet Technology(ICDCIT 2004), in LNCS 3347*, pages 255–265, Bhubaneswar, India, December 2004. Springer.

[12] Z. Liu, J. He, and X. Li. Toward a formal use of UML for software requirement analysis. In H. Arabnia, editor, *The Proceedings of PDPTA'2001 International Conference*, pages 27–33, Las Vegas, USA, June 2001. CSREA.

[13] Z. Liu, X. Li, and J. He. Using transition systems to unify UML models. In C. George, editor, *The Proceedings of 4th International Conference on Formal Engineering Methods (ICFEM2002), in LNCS 2495*, pages 535–547, Shanghai, China, October 2002. Springer-Verlag.

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

[15] J. Odell. *Advanced Object-Oriented Analaysis & Design Using UML*. Cambridge University Press, 1998.

[16] R. Plosch. *Contracts, Scenarios and Prototypes: an Integrated Approach to High Quality Software*. Springer, 2004.

[17] I. Sommerville. *Software Engineering (7th Edition)*. Addison-Wesley, 2004.

[18] I. Sommerville and P. Sawyer. *Requirements Engineering: a good practive guide*. John Wiley, 1997.

[19] J. Warmer and A. Kleppe. *the Object Constraint Language: precise modeling with UML*. Addison-Wesley, 1999.