The United Nations University

UNU/IIST

International Institute for Software Technology

Using Transition Systems to Unify UML Requirement Models

LIU Zhiming, LI Xiaoshan and HE Jifeng

October 2002

UNU/IIST Report No. 263

UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2003

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Using Transition Systems to Unify UML Requirement Models

LIU Zhiming, LI Xiaoshan and HE Jifeng

Abstract

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

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

LIU Zhiming is a research fellow at UNU/IIST, on leave from Department of Mathematics and Computer Science at the University of Liecester, Liecester, England where he is lecture in computer science. His research interests include theory of computing systems, including sound methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems, and formal techniques for OO development. His teaching interests are Communication and Concurrency, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. E-mail: Z.Liu@iis.unu.edu.

LI Xiaoshan is an Assistant Professor at the University of Macau. His research areas are interval temporal logic, formal specification and simulation of computer systems, formal methods in system design and implementation. E-mail: xsl@umac.mo.

HE Jifeng is a senior research fellow of UNU/IIST. He is also a professor of computer science at East China Normal University and Shanghai Jiao Tong University. His research interests include the Mathematical theory of programming and refined methods, design techniques for the mixed software and hardware systems. E-mail: hjif@iist.unu.edu.

Contents

1 Introduction

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

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

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

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

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

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

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

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

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

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

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

2 Conceptual Model

One of the main artifacts to produce in an OO analysis is a conceptual class diagram. Such a diagram captures the physical *concepts* and their *relations* in the system's application domain. In UML, a concept is represented by a *class* with a given name. An instance of a concept is called an *object* of the corresponding class. A relation between two concepts are denoted by an

association. In addition to associations between concepts, a concept may have some *properties* represented by *attributes*. For example, **Account** has a *balance* as an attribute and **Customer** has a *name* as an attribute. There are two approaches to deal with attributes. The first is to introduce *types of pure data values* [BRJ99] and then to represent attributes as *component fields* of classes. Alternatively, these types of pure data values can be treated as classes and attributes as associations [LLH01]. In this paper, we follow the former approach.

We must understand that at the requirement level, a class simply represents a set of objects. A system requirement specification is concerned with what the system does as a whole rather than what an individual object does, how an object behaves, or how an attribute of an object is represented. The decision on the later issues will be made during design. A use case is designed by decomposing its *responsibilities* and assigning them to appropriate objects [Lar98]. Use case decomposition and responsibility assignment are carried out according to the *knowledge* that the objects maintain[1]. *What an object can do depends on what it knows, though an object does not have to do all what it can do*. What an object knows is determined by its attributes and associations with other objects. Only when the responsibilities of the objects are decided in design, can the directions of the associations (i.e. *navigation* and *visibility*) and the methods of the classes be determined. This indicates that an association has no direction or equivalently two directions and a class has no methods. This nature of conceptual models enables us to avoid from recursive definition of objects and method calls and to keep the theory simple [HLL01].

2.1 Conceptual class diagram

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

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

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

- $\mathcal{C}$ is a nonempty finite subset of **CName**, called the *classes* or *concepts* of Δ.
- Ass is a partial function $Ass : \mathcal{C} \pfun (\mathbf{AName} \pfun \mathbb{P}\mathbf{N} \times \mathbb{P}\mathbf{N} \times \mathcal{C})$ such that

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

  where $\mathbb{P}\mathbf{N}$ is the powerset of **N**.

  If $Ass(C_1)(A) = \langle M_1, M_2, C_2 \rangle$, then A is called an *association* between C_1 and C_2, M_1 and M_2 are called the cardinalities of C_1 and C_2 in A. An association A is in general denoted

[1] This is the main idea of the design pattern called *Expert Pattern*.

by $A : (C_1, M_1, M_2, C_2)$. We use $AssN(C_1, C_2)$ to denote the set of all the associations between C_1 and C_2.

- Att is a partial function $Att : \mathcal{C} \rightarrow\!\!\!\!\circ\!\!\!\!\rightarrow (\mathbf{attrName} \rightarrow\!\!\!\!\circ\!\!\!\!\rightarrow \mathcal{T})$. We use $C.a : \mathbf{T}$ to denote $Att(C)(a) = \mathbf{T}$, and call a an *attribute* of C and $\mathbf{T}$ the *type* of a. We use $attV(C)$ to denote the set $\{a : \mathbf{T} \mid Att(C)(a) = \mathbf{T}\}$ of all the attributes of C.
- $\lhd\!\!\!- \subseteq \mathcal{C} \times \mathcal{C}$ is the *direct generalization relation* between classes. We use $C_1 \lhd\!\!\!- C_2$ to denote $(C_1, C_2) \in \lhd\!\!\!-$ and say that C_1 is a *direct superclass* of C_2, and C_2 is a *direct subclass* of C_1.

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

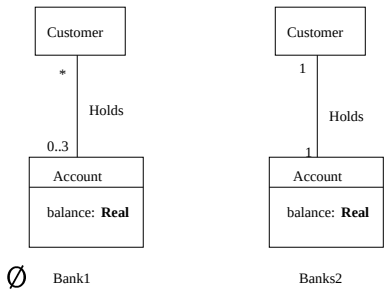

Figure 1: An example of class diagram

2.2 Semantics of class conceptual diagrams

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

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

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

SUBT-3: $\mathbf{C}_1 <: \mathbf{C}_2$ if $C_2 \lhd\!\!\!- C_1$ is contained in Δ.

SUBT-4: $\mathbf{C_1} <: \mathbf{C_3}$ if $\mathbf{C}_1 <: \mathbf{C}_2$ and $\mathbf{C}_2 <: \mathbf{C}_3$.

The meaning of $\mathbf{C}_1 <: \mathbf{C}_2$ is defined as the set inclusion $\mathbf{C}_1 \subseteq \mathbf{C}_2$.

In general, for two list of types $\underline{\mathbf{T}}_1 = \mathbf{T}_{11} \ldots \mathbf{T}_{1n}$ and $\underline{\mathbf{T}}_2 = \mathbf{T}_{21} \ldots \mathbf{T}_{2m}$, let

$$\underline{\mathbf{T}}_1 <: \underline{\mathbf{T}}_2 \stackrel{def}{=} m = n \wedge \forall i : 1..n \bullet (\mathbf{T}_{1i} <: \mathbf{T}_{2i})$$

We require that a class diagram satisfies the following conditions.

1. The generalization is acyclic:

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

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

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

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

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

   $$W_3(\Delta) \stackrel{def}{=} \left(\begin{array}{ll} & \mathbf{C}_1 <: \mathbf{C}_2 \\ \wedge & C_1 \not\equiv C_2 \end{array} \right) \Rightarrow \pi_1(attV(C_1)) \cap \pi_1(attV(C_2)) = \emptyset$$

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

   $$W_4(\Delta) \stackrel{def}{=} \left(\begin{array}{ll} & \mathbf{C}_1 <: \mathbf{C}_2 \\ \wedge & C_1 \not\equiv C_2 \end{array} \right) \Rightarrow \left(\begin{array}{l} \forall C \in \mathbf{CName} \bullet \\ (AssN(C_1, C) \cap AssN(C_2, C) = \emptyset) \end{array} \right)$$

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

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

A class diagram Δ is *well-formed* if it satisfies

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

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

1. **CVar** $\stackrel{def}{=} \{C : \mathbb{P}\mathbf{C} \mid C \in \mathcal{C}\}$ in which each C records the current set of objects of class $\mathbf{C}$ existing in the system.

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

We call variables in $\mathbf{CVar} \cup \mathbf{AVar}$ variables in Δ too. In general, we use $type(x)$ to denote the type of a variable or an expression x, and $type(\underline{x})$ to denote the list of types of $type(x)$.

2.3 Object diagrams as system states

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

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

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

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

  Let $\mathbf{Att}$ be a state variable that takes values of sets of the form

  $$\{o.a_1 : \mathbf{T}_1, \ldots, o.a_n : \mathbf{T}_n\}$$

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

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

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

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

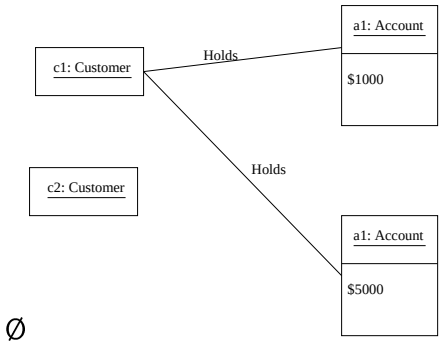

Figure 2: An example of an object model

2.4 State assertions

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

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

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

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

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

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

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

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

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

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

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

$$\left(\begin{array}{l}\forall c \in Copy, \\ \forall r \in Reservation, \\ \forall p \in Publication\end{array}\right) \bullet (IsHeldFor(c,r) \wedge IsOn(r,p) \Rightarrow Has(p,c))$$

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

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

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

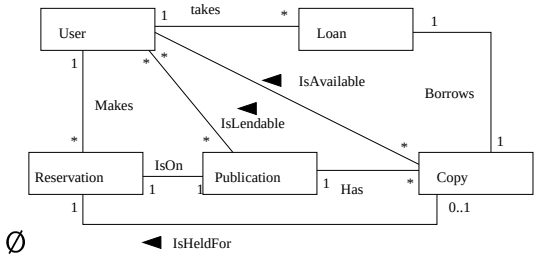

Figure 3: A class diagram for a library system

2.5 Conceptual models

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

A state property ψ of a conceptual model can be reasoned about by proving the implication $\theta \wedge Inv \Rightarrow \psi$ in the relational calculus. We denote by $CM \models \psi$ that CM satisfies ψ. This also allows us to define transformations between conceptual diagrams that preserve a state constraint.

2.6 Associative classes, abstract classes and aggregations

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

$$Has \circ IsFor = Employs$$

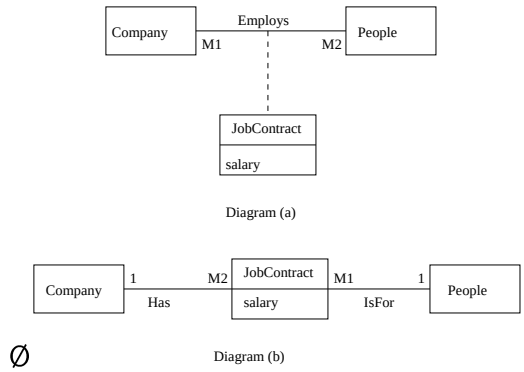

Figure 4: Representation of Associative Class

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

A class $C \in$ **CName** is an *abstract class* if there are $C_1, \ldots, C_k \in$ **CName** such that $C \lhd\!\!- C_i$ and the following condition is an invariant of the system:

$$C = \bigcup_{i=1\ldots k} C_i$$

The relation that a class C is an *aggregation* of a class C_1 can be treated as a general association. Most of the properties of an aggregation association are design-oriented and more about the visibility of the *whole* C to the *part* C_1. There are two kinds of aggregations, *composite aggregations* and *shared aggregations*. C is a composite aggregation of C_1 if an object in C_1 can only be a part of *one* object in C. In the case when C is a shared aggregation of C_1, an object

of C_1 can be shared by a number of objects of C as their part. These properties can be specified in terms of cardinalities as shown earlier.

There are some behavioural properties about aggregation such as parts cannot exist with out a whole. If we use $IsPartOf$ to denote a general aggregation, this property can be specified as.

$$\forall o_1 : \mathbf{C}_1 \bullet (o_1 \in C_1 \Rightarrow \exists o \in C \bullet IsPartOf(o_1, o))$$

This implies that all the parts must be removed when their whole is removed.

3 Use-Case Model

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

3.1 System operations

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

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

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

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

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

where $\underline{w}$ contains all the system's variables but those in X.

A system operation operates on the following variables.

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

We have **V** to be the set that contains the variables identified in items 1-4. Let **U** be the set of variables that contains the actual parameters and control variables identified in items 5 and 6, and **P** the set of formal parameters required for the specification of the system operations. We define $\Gamma \stackrel{def}{=} \mathbf{V} \cup \mathbf{U}$, and $\Omega \stackrel{def}{=} \mathbf{V} \cup \mathbf{P}$. Now a system operation is defined in the form

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

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

The semantics of operation op is defined as a design

$$op \stackrel{def}{=} X : (P \vdash R)$$

where, P and R only contain variables in Γ, and X is the set of variables that can be modified by op.

3.2 Actors operations

An actor's operation may modify actual value parameters in **val** and read values from actual result parameters in **res**. It of course may modify and read a control variable. We call these operations that only access variables in **U** *control operations*. A control operation has no effect on the object diagrams of the conceptual model. Such an operation is in general a guarded design of the form

$$g \longrightarrow X : (Pre \vdash Post) \stackrel{def}{=} g \wedge (X : Pre \vdash Post)$$

This operation can only take place in a state that assigns g to $true$. Remember that such an operation only contains variables in **U**.

Actors have to call system operations to carry out a use case. It is possible that one or more actors call an operation for a number of times as well as different system operations. To define system operation calls by actors, we introduce a set **Actor** of names to represent the set of individual actors (or users) involved in the system. A *call* to a system operation $op[\underline{x} : \mathbf{T}_1;\ \underline{y} : \mathbf{T}_2] :: \mathbf{Pre} : P;\ \mathbf{Post} : R$ by an actor u is an operation of the form $op_u(\underline{val};\ \underline{res})$:

$$\begin{array}{lcl} op_u(\underline{val};\ \underline{res}) & \stackrel{def}{=} & W(op(\underline{val};\ \underline{res})) \Rightarrow op[\underline{val}/\underline{x};\ \underline{res}/\underline{y}] \\ W(op_u(\underline{val};\ \underline{res})) & \stackrel{def}{=} & (type(\underline{val}) <: \mathbf{T}_1) \wedge (\mathbf{T}_2 <: type(\underline{res})) \end{array}$$

where $op[\underline{val}/\underline{x};\ \underline{res}/\underline{y}]$ is obtained from the design of $op[\underline{x} : \mathbf{T}_1;\ \underline{y} : \mathbf{T}_2]$ by substituting $\underline{x}$ and $\underline{y}$ with $\underline{val}$ and $\underline{res}$ respectively.

This definition implies that it is the caller's responsibility to ensure the correctness of the types of the actual parameters. If a call is not well-formed, the system behaves chaotically. Of course, a system operation can be *refined* to handle unwell-formed calls as exceptions.

For concurrent applications, an actor u can make anumber of calls simultaneously:

$$op^1_u(\underline{val}_1;\ \underline{res}_1) \wedge op^1_u(\underline{val}_2;\ \underline{res}_2)$$

In general, we allow to conjoin two actors actions by conjunction $\wedge$. Such an operation is called an *joint operation*.

A call to a system operation by an actor can be conditional and in the form of

$$g \longrightarrow op_u(\underline{val};\ \underline{res})$$

where g only contains variables in **U**. A non-conditional call is a special conditional call in which the guard is $true$.

In summary, an operation of an actor is either a control operation, a conditional call or an joint action to a system operations. The use cases of a system are then specified by giving a set **Actor** of actors, a set of system operations **OP** and a set of actors operations (also called *actions*) **UA**.

Some readers may wonder how sequential composition of two actors operations specified. This can be done by using a control variable ℓ:

$$P_1;\ P_2 \stackrel{def}{=} \{(\ell = c_1) \longrightarrow P_1 \wedge (\ell' = c_2), (\ell = c_2) \longrightarrow P_2 \wedge (\ell' = c_3)\}$$

where c_1, c_2 and c_3 represent the *labels* of commands, and it is required that $\ell = c_1$ initially.

For example, we can specify system operations $FindUser[Id : \mathbf{N};\ u : \mathbf{User}]$, $FindCopy[Id_1 : \mathbf{N};\ c : \mathbf{Copy}]$, $Loan[u : \mathbf{User}, c : \mathbf{Copy}]$ for a library system [LLH01]. Actor librarian L can carry out the use case $LendCopy_L(i_1, i_2)$ by the protocol

$$(i_1' \in \mathbf{N}) \wedge (i_2' \in \mathbf{N});$$
$$(FindUser_L(i_1;\ u);\ (u \neq null) \longrightarrow FindCopy_L(i_2;\ c);$$
$$(c \neq null) \longrightarrow Loan_L(u, c)$$

This use case defines the behaviour of the process of lending a copy to a user. The librarian first inputs the identifications of the user and the copy. He/she then calls the system operation $FindUser$ which returns the user object if found otherwise *null*. If the user is found, the librarian then calls the system operation $FindCopy$ which returns the copy if found otherwise *null*. If the copy is found, the librarian calls the system operation $Loan$ that creates a loan and record the user and copy on the loan.

This specification is only correct if no other actor tries to lend the same copy to another user at the same time. In that case $FindCopy_L$ and $Loan_L$ should be carried out together atomically as

$$(u \neq null) \longrightarrow FindCopy_L(i_2;\ c) \wedge Loan_L(u, c)$$

3.3 Constructing a system specification

Given a conceptual model $CM = (\Delta, Inv)$, a set of systems operations **OP**, a set **Actor** of actors, and a set **UA** of actors operations, we construct

- The set Γ of variables that the actors operations operate on.
- An initial condition $Init$ which define the set of states from which the system can start to work.

The system is then specified by the transition system $\mathcal{S} \stackrel{def}{=} (\Gamma, Inv, \mathbf{OP}, Init, \mathbf{UA})$.

$\mathcal{S}$ is well formed if every operation call in **UA** is a call to an operation in **OP** from an actor in **Actor**:

$$W(\mathcal{S}) \stackrel{def}{=} \forall op_u (op_u \in \mathbf{UA} \Rightarrow op \in \mathbf{OP} \wedge u \in \mathbf{Actor})$$

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

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

Each execution of $\mathcal{S}$ in an interleaving of a set of *scenarios* of use cases. The invariant properties Inv can be prove by showing that for each $op_u \in \mathbf{UA}$,

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

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

From the work [HMP94] about the definition of Hoar's CSP specification by transition systems, we can also write the specification of use cases in the notation of CSP.

Let each operation call $op_u(\underline{val};\ \underline{res})$ be an event triggered by actor u, and let $\mathcal{D}(op_u(\underline{val};\ \underline{res}))$ denote the design of $op_u(\underline{val};\ \underline{res}))$ defined in this section. Define a CSP command

$$op_u(\underline{val};\ \underline{res}) \longrightarrow (\mathcal{D}(op_u(\underline{val};\ \underline{res}));\ out_u!(\underline{res}))$$

The meaning of this command is that once an interaction op_u between actor u and the system occurs, the system carries out the operation according to the design of the operation to change its state and outputs results to the actor.

Such a CSP command can be guarded by a Boolean expression too in the form

$$b \wedge op_u(\underline{val};\ \underline{res}) \longrightarrow (\mathcal{D}(op_u(\underline{val};\ \underline{res}));\ out_u!(\underline{res}))$$

Then sequential composition ";", external choices "[]", non-deterministic choice "$\sqcap$", and recursion can be defined to form a specification of a use case as a CSP process. The use-case model is then given as the interleaving composition of the specified use cases in the form

$$U_1 \,|||\, \ldots \,|||\, U_m$$

where each U_i, $i = 1, \ldots, U_m$, is a CSP process specification of a use case.

3.4 Examples of use cases

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

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

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

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

Model $Bank_1$ supports a "withdraw" operation that behaves different from the above one:

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

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

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

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

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

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

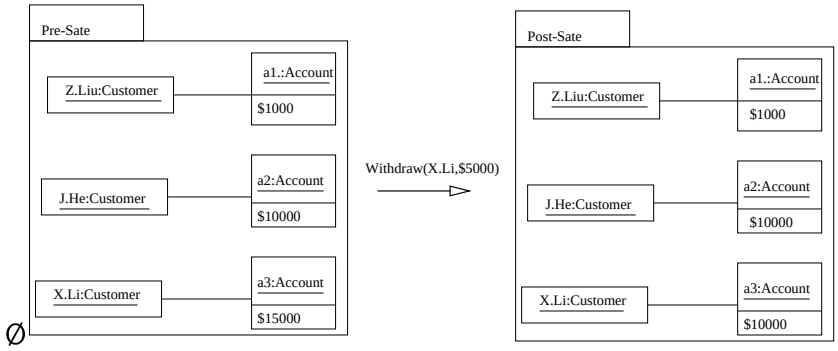

Figure 5: Effect of Withdraw Use Case

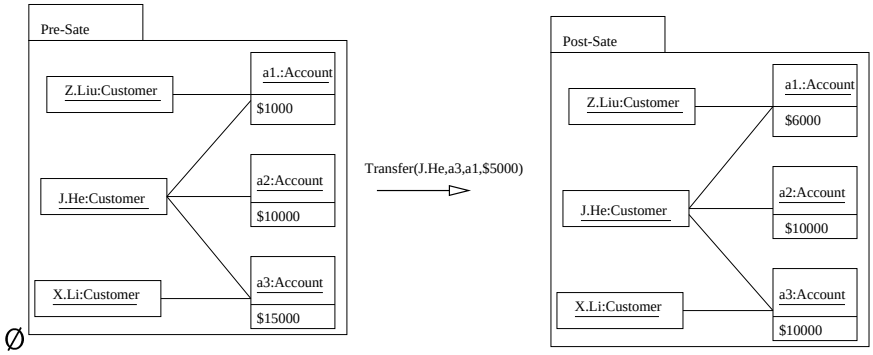

Figure 6: Effect of Transfer Use Case

4 Conclusion & Discussion

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

The advantage of using a well-established model is that we do not have to develop or study new semantics and tools for verification. Methods and tool for specification and verification of transition system are well-established, e.g. [Lam91, MP91, CES86, EGL92]. A system specification can also be written in terms of Hoar's CSP that can be defined in terms of a transition system [HMP94]. Furthermore, this model is already extended to deal with real-time and fault-tolerance [HMP91, AL92, LJ99]. The same methods can be used to deal with real-times requirements on use cases. Also, the model of transition systems is isomorphic to that of the statecharts which is a part of UML.

The main difference between our work and that in [pG99, EKHG01, Egy01] is that we study

formal semantic relationships between different models of UML, rather than only formalization of individual diagrams. The paper [HR00] also treats a class as a set of objects and an association as a relation between objects. However, it does not consider use cases. Our work also shares some common ideas with [BPP99] in the treatment of use cases. However, we have a different understand about a conceptual model and have addressed the clear relationships between the UML models. We have also provided a working procedure for building a system specification from UML models.

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

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