



The United Nations
University

UNU/IIST

International Institute for
Software Technology

A Relational Model for Formal Object-Oriented Requirement Analysis in UML

Zhiming Liu, He Jifeng, Xiaoshan Li and Yifeng Chen

October 2003

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 Endowment Fund. As well as providing two-thirds of the endowment 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. Schools and 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's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU/IIST and its report series.

Chris George, Acting Director



The United Nations
University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

A Relational Model for Formal Object-Oriented Requirement Analysis in UML

Zhiming Liu, He Jifeng, Xiaoshan Li and Yifeng Chen

Abstract

This paper is towards the development of a methodology for object-oriented software development. The intention is to support effective use of a formal model for specifying and reasoning during the requirements analysis and design of a software development process. The overall purpose is to enhance the application of the Unified Modelling Language (UML) with a formal semantics in the Rational Unified Software Development Process (RUP). The semantic framework defines the meaning of some UML submodels. It identifies both the *static* and *dynamic* relationships among these submodels. Thus, the focus of this paper is the development of a semantic model to consistently combine a *use-case model* and a *conceptual class diagram* to form a system specification.

Keywords: *Object-orientation, UML, use-cases, conceptual models, requirement specification*

This paper is to occur in the proceedings of ICFEM 2003, 5-7 November, 2003, Singapore

Zhiming Liu is a research fellow at UNU/IIST, on leave from Department of 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.

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: hjjf@iist.unu.edu.

Xiaoshan Li is an Associate 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.

Yifeng Chen is a lecturer of the Department of computer Science at the University of Leicester. His research interest includes logics for computer science, programming and specification language designs, and formal specification and verification. E-mail: yc10@mcs.le.ac.uk.

Contents

1	Introduction	1
2	Models in Rational Development Process	2
3	Computational Model	3
4	Conceptual Model	4
4.1	Conceptual class diagram	4
4.2	Inheritance and well-formedness conditions	6
4.3	Object diagrams as system states	7
4.4	Conceptual model	8
4.5	Special classes and associations	9
5	Use-Case Model	10
5.1	System operations	11
5.2	Actors	11
5.3	System specification	12
6	Semantics	15
6.1	Expressions	15
6.2	Commands	16
6.3	Declarations	18
6.4	Call to a required method	19
6.5	System specification	20
7	Conclusion & Related Work	20
7.1	Conclusion	20
7.2	Related work	21

1 Introduction

Object orientation is now a popular approach in the software industries. UML [Gro99] is the de-facto standard modelling language for the development of software with a broad range of applications, covering the early development stages of requirements analysis and specification and with strong support for design and implementation [JBR99, DW98]. Driven by this trend, computer scientists are now intensifying the research to help better understanding and use of OO methods and UML, e.g. [Ken97, FELR98, BPP99, HR00, RCA01, Jür02].

A main feature of UML is that different modelling diagrams are used to represent a system from various views at different levels of abstraction. This however gives rise to the questions of whether different models used in a system development are consistent and how they are related. In [EKHG01], problems concerning consistency among the models for different views are classified as *horizontal consistency*, and those about models at different levels of abstraction as *vertical consistency*. Furthermore, consistency of each kind is divided into *syntactical consistency* and *semantic consistency*.

Conditions of syntactical consistency are expressed in UML in terms of the wellformedness rules of OCL (Object Constraint Language) [WK99]. Obviously semantic consistency requires syntactical consistency. Article [EKHG01] studies a particular kind of *behavioral consistency* among different statecharts of a system by translating them into Hoare's CSP. The work in [Egy01] deals with automated checking of horizontal syntactical consistency between a design class diagram and a sequence diagrams of a system at the design level.

In this paper, we provide a unified framework for formal specification for the models used in different activities in RUP [JBR99, Kru]. The framework enables us to identify the distinguishable features of different models and to relate and manipulate them as well. These models are UML requirement models including *use-case models* and *class diagrams*, and the design models that are *interaction diagrams* and *design class diagrams*. Our long term aim is to support formal use of UML in requirement specification and analysis, and transformation of requirement models to design models. When it is used within the incremental and iterative RUP, the method allows stepwise refinement and supports object-oriented and component-based software development. We believe this will help to change today's situation that OO software development in practice is usually done in a non-scientific manner based on pragmatics and heuristics. On the other hand, with the incorporation of our model into the incremental and iterative RUP, we hope to improve the use of formal methods in the development of large scale systems.

Our formalization follows the Unifying Theories of Programming of Hoare and He [HH98] and is based on the relational model for object-oriented programming in [HLL01]. It uses a simple set theory and predicate logic, rather than a particular formal specification language, such as Z or VDM.

In this paper, we focus on only conceptual aspects of object orientation. Most syntactical and semantic consistency conditions defined in this paper have straightforward algorithms for checking and hence support from necessary automated tools. We have started our effort to build such as tool [LLH03].

In the rest of this paper, Section 2 briefly discusses the activities and models in RUP that we intend to formalize. Section 3 introduces a computational model that is similar to the notation of action systems in [MP81]. Section 4 defines a syntax and a semantics for a conceptual model. Section 5 defines a syntax for use-case model using the relational model developed in [HLL01]. Section 6 gives a semantics for a use case and a *canonical form* of system specification. It shows how a use case behaves in the context of a conceptual model, and captures the consistency between the two models. Finally conclusions and discussion are given in

Section 7. Simple examples are used to illustrate the ideas and formalization.

2 Models in Rational Development Process

Requirements capture, analysis, design and modelling are the main technical activities in the early stage of a RUP cycle. Requirements analysis mainly involves the creation and analysis of *use-case models* and *conceptual models* [JBR99, Lar01, DW98].

A *use-case model* consists of a set of *use-case diagrams*, and a family of *use-case descriptions* in text, each describing one use case. Each use-case description specifies a required functional service that the system is expected to provide for certain kinds of users called *actors*. It describes a *use case* in terms of a pattern of interactions between the actors and the system. The use-case diagrams do not provide much semantic information. They only illustrate which actors use which use cases and which use cases are *included* as parts of a use case. Therefore, a use-case model specifies the systems's required functional services, the users of these services, and the dependency relationships among these services. A library system, for example, has use cases to, *Borrow a copy*, *Make a Reservation* and *Validate User Identification* for the actor called **User**. Both *Borrow a Copy* and *Make a Reservation* includes *Validate User Identification*.

A conceptual model 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*).

For example, the library system has **User**, **Loan**, **Publication** and **Copy**. They are associated so that a *user takes* (currently) a number of loans and a loan *borrow*s a copy of a publication. Different library systems have different conceptual models and provide different services. A “small” library does not allow a user to make a reservation, while a “big” library may provide this service. A conceptual class diagram is given in Figure 1. In addition to associations between concepts, a concept may have some *properties* represented by *attributes*.

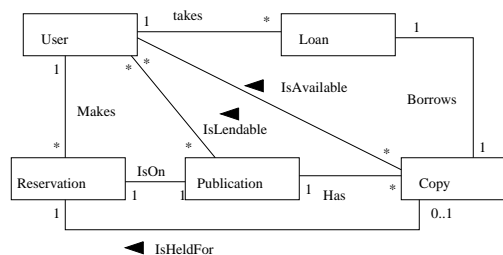


Figure 1: A conceptual class diagram

For example, **User** has a *name* as an attribute, and **Loan** has a *date* as an attribute.

We say a class diagram is conceptual at this level because it is not concerned with what an object does, how it behaves, or how an attribute is represented. The decision on these issues will be made during design when the *responsibilities* of a use case are decomposed and assigned to appropriate objects. Use case decomposition is carried out according to the *knowledge* that the objects maintain. *What an object can do depends on what it knows, though an object does not have to do all that 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 the design, can the directions of the associations (i.e. *navigation* and *visibility* from one object to another)

be determined. This indicates that an association at the conceptual level are simply sets of pairs of objects and has no direction or equivalently two directions. Therefore, a conceptual model is a *static model* of the *structure* of the application domain.

The relationship between a use-case model and a conceptual model is that the conceptual model specifies the environment, i.e. *the state space*, under which the use cases are to be carried out. A *state* is an *object diagram* that consists of a set of objects and a set of *links* between these objects. Each object and each link in a state must be respectively an instance of a class and an association declared in the conceptual model. An execution step in a use case transforms a state into another. A conceptual model is consistent with a use-case model if it is *adequate* to realize the functional services required by the use-case model.

In *system design*, a requirement specification is realized by a design specification consisting of a *design class diagram* and a family of *interaction diagrams*. The design class diagram models the *software structure* that realizes the conceptual model of the requirement specification. The interaction diagrams (i.e. collaboration diagrams or a sequence diagrams) model the interactions between objects and realize the use cases of the requirement specification. The creation and manipulation of these design models mainly involve use case decomposition and assignment of responsibilities to objects. The interaction diagrams must meet the requirements. This can be proved by showing that the use cases are indeed correctly realized by the interactions between objects. Experience [Lar01] shows that once a design class diagram is obtained, code can be easily produced from it. It is possible to develop a tool to help in transforming a design into a code of implementation.

3 Computational Model

We use a notation similar to a transition or action system [MP81] to combine the two models together to model a system. A *system* is defined by a tuple $(\alpha, \Phi, \Theta, P)$ where

- α denotes the set of program variables known to the program.
- P is a set of *operations*, each of which is a predicate that relates the initial values of program variables. The predicate is of the form $p(x) \vdash R(x, x')$ (called a design in [HH98]):

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

where x and x' represent the initial and final values of x respectively; ok asserting that the operation is started well and ok' means that the operation terminated; $p(x)$ is called the precondition, and $R(x, x')$ the post-condition or the transition relation.

- Θ is a predicate over α , called the *initial condition* and defines the initial state(s) of the system.
- Φ is a predicate over α , called the *invariant*. It must be true in any initial state and preserved by each operation in P .

An action only changes a subset of variables declared in α . The *normal form* of a design is thus a *framed design* of the form $V : (p \vdash R)$, that denotes $p \vdash R \wedge (\underline{w}' = \underline{w})$, where V and \underline{w} are subsets of α , and $\underline{w} = \alpha - V$. When there is no confusion, we will omit the frame in a design by assuming that a variable x can be changed by a design only if its primed version x' occurs in the design.

The above model has to take into account the following OO aspects.

1. A use case is composed from a number of operations, while a conceptual model determines the following variables on which a use case operates:
 - for every concept, a *class variable* that takes values of sets of objects of the concept;
 - for every object of a concept, a variable for each attribute of the concept;
 - for every association, an *association variable* that take values of sets of links (i.e. pairs) between objects of the associated concepts.
2. Due to the inheritance mechanism, the effect of a use case on a variable depends on its current type during execution, rather than its originally declared type.
3. As in imperative languages, a state of a variable is its current value. An object is represented as a finite tuple that records its *identity*, current type, and the values of its attributes. As an object has no attributes of object types in a conceptual model, there is no recursive nesting needed here. Association variables are used to represent links between objects, which may be realized as object attributes in the later design and implementation.

In summary, a model of an OO requirement is a system $\mathbf{S} = (\alpha, \Phi, \Theta, P)$ where

- P consists of a set of use cases.
- α identifies the variables on which the use cases in P operate and it is determined by the conceptual class diagram and the input and output variables of the program.
- The invariant Φ formally models the invariant constraint. The pair (α, Φ) thus gives the formalization of the conceptual model.
- Θ is a condition to be established when starting up the system.

In the following sections, we formulate these four components of a system specification.

4 Conceptual Model

A conceptual model is a pair $CM = (\mathcal{D}, \Phi)$, where \mathcal{D} is a *class diagram* and Φ is the state constraint on the classes and associations enforced by \mathcal{D} .

4.1 Conceptual class diagram

A conceptual class diagram \mathcal{D} of an application identifies the environment in which the use cases operate. This environment consists of four parts:

1. The first part provides the *static* information on classes and their inheritance relationships:

- **CN**: the finite set of classes identified in the diagram. We use bold capital letters to represent arbitrary classes and types.
- **super**: the partial function which maps a class to its *direct* superclass, i.e. $\mathbf{super}(\mathbf{C}) = \mathbf{D}$ if \mathbf{D} is the direct superclass of \mathbf{C} .

2. The second part describes the structure of each class and for $\mathbf{C} \in \mathbf{CN}$, it includes

attr(\mathbf{C}): the set of $\{\langle a_1 : \mathbf{T}_1 \rangle, \dots, \langle a_m : \mathbf{T}_m \rangle\}$ attributes of \mathbf{C} , where \mathbf{T}_i stands for the type of attribute a_i of class \mathbf{C} , and will be referred by $type(\mathbf{C}.a_i)$.

As in [AM02], the type of an attribute is assumed to be a built-in *simple data type*, such as the natural numbers \mathbf{Nat} . We use \mathbf{DT} to denote the set of these assumed data types. Each class \mathbf{C} defines a type, also denoted by \mathbf{C} . We allow the construction of a type from the direct product of two types, and the power set $\mathbb{P}(\mathbf{T})$ of a type \mathbf{T} .

3. The third part identifies the relationships among the classes:

AVar: the finite set of associations captured in the diagram and declared as *association variables*

$$\{A_1 : \mathbb{P}(\mathbf{C}_{11} \times \mathbf{C}_{12}), \dots, A_m : \mathbb{P}(\mathbf{C}_{m1} \times \mathbf{C}_{m2})\}$$

The type of each A_i , referred by $type(A_i)$, is the powerset $\mathbb{P}(\mathbf{C}_{i1} \times \mathbf{C}_{i2})$. We use \mathbf{AN} to denote the list A_1, \dots, A_m of the association names in **AVar**. The separation of the treatments of attributes and associations supports a more flexible design of the interactions or connection between objects [FM96, HR00, AM02]. For simplicity, we only deal with binary associations. General relations among classes can be modelled in the same way.

4. For each class name $\mathbf{C} \in \mathbf{CN}$, there is one state variable, denoted by C , whose value records the objects of class \mathbf{C} currently existing in the system:

$$\mathbf{CVar} \stackrel{def}{=} \{C : \mathbb{P}(\mathbf{C}) \mid \mathbf{C} \in \mathbf{CN}\}$$

The type of C , denoted by $type(C)$, is $\mathbb{P}(\mathbf{C})$.

The multiplicities of the roles of an association will be specified in the invariant of the conceptual model.

Example The formalization of class diagram in Figure 1 is given as follows, where every class is a subclass of class **Object** and we omit **attr**(\mathbf{C}) when \mathbf{C} has no attributes:

$$\begin{aligned} \mathbf{CN} &= \{\mathbf{User}, \mathbf{Loan}, \mathbf{Copy}, \mathbf{Publ}, \mathbf{Resv}\} \\ \mathbf{super}(\mathbf{C}) &= \mathbf{Object}, \text{ for all } \mathbf{C} \in \mathbf{CN} \\ \mathbf{attr}(\mathbf{User}) &= \{\langle Id : \mathbf{String}, name : \mathbf{String} \rangle\} \\ \mathbf{attr}(\mathbf{Loan}) &= \{\langle date : \mathbf{Date} \rangle\} \\ \mathbf{AVar} &= \{\mathit{Takes} : \mathbb{P}(\mathbf{User} \times \mathbf{Loan}), \mathit{Borrows} : \mathbb{P}(\mathbf{Loan} \times \mathbf{Copy}), \\ &\quad \mathit{IsAvailable} : \mathbb{P}(\mathbf{Copy} \times \mathbf{User}), \\ &\quad \mathit{IsLendable} : \mathbb{P}(\mathbf{Publ} \times \mathbf{User}), \mathit{Has} : \mathbb{P}(\mathbf{Publ} \times \mathbf{Copy}), \\ &\quad \mathit{IsHeldFor} : \mathbb{P}(\mathbf{Copy} \times \mathbf{Resv}), \mathit{Makes} : \mathbb{P}(\mathbf{User} \times \mathbf{Resv})\} \\ \mathbf{CVar} &= \{\mathit{User} : \mathbb{P}(\mathbf{User}), \mathit{Copy} : \mathbb{P}(\mathbf{Copy}), \mathit{Loan} : \mathbb{P}(\mathbf{Loan}), \\ &\quad \mathit{Publ} : \mathbb{P}(\mathbf{Publ}), \mathit{Resv} : \mathbb{P}(\mathbf{Resv})\} \end{aligned}$$

A refinement of the model allows us to add more details, such as attributes and associations [HLL02].

4.2 Inheritance and well-formedness conditions

Every attribute of a class in a conceptual model is inherited by its subclasses. Formally speaking, we require

$$\mathbf{super}(\mathbf{C}) = \mathbf{D} \Rightarrow \mathbf{attr}(\mathbf{C}) \supseteq \mathbf{attr}(\mathbf{D})$$

Thus, when drawing or describing a class diagram, we do not repeat the attributes of a class in its subclasses.

A class diagram is *well-formed* when following conditions are met:

1. The function **super** does **not** cause circularity:

$$NoCirc \stackrel{def}{=} \mathbf{super}^+ \cap Id = \emptyset$$

where we abuse the notation by treating **super** as a binary relation, with *Id* denoting the identity relation and the definition using relation composition “;”.

$$\begin{aligned} \mathbf{super}^+ &\stackrel{def}{=} \bigcup_{n \geq 1} \mathbf{super}^n, & \mathbf{super}^1 &\stackrel{def}{=} \mathbf{super} \\ \mathbf{super}^{n+1} &\stackrel{def}{=} \mathbf{super}^n; \mathbf{super} \end{aligned}$$

We use $\mathbf{N} < \mathbf{M}$ to denote that \mathbf{N} is a subclass of \mathbf{M} .

2. An association only relate classes in the diagram, i.e. :

$$WFAsso \stackrel{def}{=} \forall (A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2)) \in \mathbf{AVar} \bullet (\mathbf{C}_1 \in \mathbf{CN} \wedge \mathbf{C}_2 \in \mathbf{CN})$$

3. The association names are all distinct:

$$DistAssoName \stackrel{def}{=} dist(\mathbf{AN})$$

4. Classes should not be related by attributes, i.e. the type of an attribute should not be a class:

$$AssoDistAttr \stackrel{def}{=} \forall \mathbf{C} \in \mathbf{CN}, a \in \mathbf{attr}(\mathbf{C}) \bullet type(\mathbf{C}.a) \in \mathbf{DT}$$

5. The attribute names of a class are distinct:

$$DistAttrName \stackrel{def}{=} \forall \mathbf{C} \in \mathbf{CN} \bullet dist(\pi_1(\mathbf{attr}(\mathbf{C})))$$

where π_1 returns the *list* of the attribute names in $\mathbf{attr}(\mathbf{C})$, and *dist* is true if the elements in the list are distinct.

Let $W(\mathcal{D}) \stackrel{def}{=} NoCirc \wedge WFAsso \wedge DistAssoName \wedge AssoDistAttr \wedge DistAttrName$ and it defines the well-formedness condition for a class diagrams \mathcal{D} .

4.3 Object diagrams as system states

An object diagram of a class diagram is a state over the $\mathbf{CVar} \cup \mathbf{AVar}$, i.e. a *well typed* mapping from $\mathbf{CVar} \cup \mathbf{AVar}$ to $\mathbb{P}(\mathbf{Object}) \cup \mathbb{P}(\mathbf{Object} \times \mathbf{Object})$.

For an association $A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2)$ and an object $o_i \in C_i$, $i = 1, 2$, let $A(o_i)$ be the set of objects in the class that is associated with C_i . Formally,

$$\begin{aligned} A(o_i) &\stackrel{def}{=} \{o_{i\oplus 1} \mid o_{i\oplus 1} \in C_{i\oplus 1} \wedge (o_1, o_2) \in A\} \\ A(C_i) &\stackrel{def}{=} \bigcup_{o \in C_i} A(o) \end{aligned}$$

where $1 \oplus 1 = 2$ and $2 \oplus 1 = 1$. The multiplicities of the roles of an association can now be defined as a state property. Let M_1 and M_2 be subsets of \mathbf{Int} . We assign M_1 and M_2 as respectively the *multiplicities* of C_1 and C_2 to the association A to enforce the following state property:

$$Multiplicity(A) \stackrel{def}{=} \bigwedge_{i=1,2} \forall o_i \in C_i \bullet (|A(o_i)| \in M_{i\oplus})$$

asserting that the number of objects in C_1 (or C_2 resp.) linked to an object in C_2 (or C_1) is bounded by the range of M_1 (or M_2). We use $A : (M_1, C_1, C_2, M_2)$ to represent an association between C_1 and C_2 with multiplicities M_1 and M_2 .

It is also required that an association A only links objects that currently exist in the state: for every association $A : \mathbb{P}(C_1 \times C_2)$ in \mathcal{D} ,

$$LinkObjects(A) \stackrel{def}{=} A(C_1) \subseteq C_1 \wedge A(C_2) \subseteq C_2$$

A state of a class diagram is *valid* if each association A of the class diagram satisfies both condition $Multiplicity(A)$ and condition $LinkObjects(A)$. In subsequent discussion, we use the term state to refer to a valid state when there is no confusion.

Conditions $Multiplicity(A)$ and $LinkObjects(A)$ define the precise meaning of an association A and the multiplicities of its roles depicted in a UML class diagram. However, only classes, associations, and their multiplicities are not enough to express all the constraints required by an application. In particular, multiplicity restrictions do not allow relationships between associations to be expressed. For example, the library application requires that a copy c that “is held” for a reservation r be a copy of the publication p reserved by the reservation r . UML uses a OCL statement to describes such as constraints. Because OCL is not expressive enough to specify the semantics of use cases, this paper uses a relational logic to specify state assertions. The above constraint in the library application can be described as the state assertion:

$$\forall c \in Copy, r \in Resv, p \in Publ \bullet IsHeldFor(c, r) \wedge IsOn(r, p) \Rightarrow Has(p, c)$$

where $R(a, b)$ iff $\langle a, b \rangle \in R$. Furthermore, this constraint can be equivalently written in terms of the algebra of relations

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

where “ \circ ” is the *composition* operation of relations, and Has^{-1} is the inverse of Has .

4.4 Conceptual model

A *conceptual model* can now be formally defined as a pair $CM = (\mathcal{D}, \Phi)$ where \mathcal{D} is a class diagram formalized above, and Φ is a state constraint on the states of \mathcal{D} . We can use the following Java-like format to specify a conceptual model as follows.

```

Conceptual Model  $CM$ 
Class  $C_{11}$  Extends  $C_{12}$   $\{T_{11} x_1; \dots; T_{1m} x_m\}$ 
    .....
Class  $C_{n1}$  Extends  $C_{n2}$   $\{T_{n1} y_1; \dots; T_{nk} y_k\}$ 
Association  $(M_1^1, C_1^1, C_1^2, M_1^2) A_1; \dots; (M_j^1, C_j^1, C_j^2, M_j^2) A_j$ 
Invariant  $\Phi$ 
End  $CM$ 

```

where C_1 **Extends** C_2 denotes that $\mathbf{super}(C_1) = C_2$. M_i^1 and M_i^2 are sets of natural numbers and represent the *multiplicities* of the roles C_i^1 and C_i^2 of association A_i , $i = 1, \dots, j$.

The following syntax can be followed when we write the specification of a conceptual class diagram *ccdec*:

$$ccdec ::= empty|cdec|adec|ccdec; ccdec$$

where *empty* denotes the empty diagram, *cdec* a class declaration, and *adec* an association declaration. From such a specification of a diagram \mathcal{D} , we can easily calculate the categories of the formal model. An alternative format would be to declare associations as classes with two attributes of the types as the associated classes.

Although we cannot present the formal UML syntax of a class diagram in this paper because of the space limit, the translation *between* a class diagram to its formalization is straightforward and can be automated. Relating the graphic presentation and the formal specification of a model in this way allows the user to obtain the later without necessarily knowing the detailed formality of the specification language.

For a state constraint Ψ , we write $CM \vdash \Psi$ iff $\Phi \Rightarrow \Psi$, meaning that Ψ can be proven from Φ in the relational calculus. This allows us to reason about properties of a conceptual model and relationships between two conceptual models. For example, we can define transformations between conceptual diagrams that have to preserve a state constraint.

4.5 Special classes and associations

We also use *state constraints* or *invariants* to specify some special classes and associations.

Abstract classes In a conceptual class diagram D , a class C is called an *abstract class*, if $C = C_1 \cup C_2 \cup \dots \cup C_k$ is an invariant of the system, where C_1, \dots, C_k are all the direct subclasses of C and $k \geq 2$. This means an object in the abstract class can only be created as an instance of one of its direct subclasses.

Association classes UML allows *association classes* to represent associations that have data properties too. An example is shown in Diagram (a) of Figure 2. **JobContract** is about the association *Employs*. This fact can be modelled by decomposing the association into two associations as shown in Diagram (b) of Figure 2. Notice the multiplicities of **Company** in the association *Has* and **People** in the association *IsFor* are both 1 to ensure that an instance of **JobContract** only relates one company and one person. However, we relate the association *Employs* with the two newly introduced associations by the constraint

$$Has \circ IsFor = Employs$$

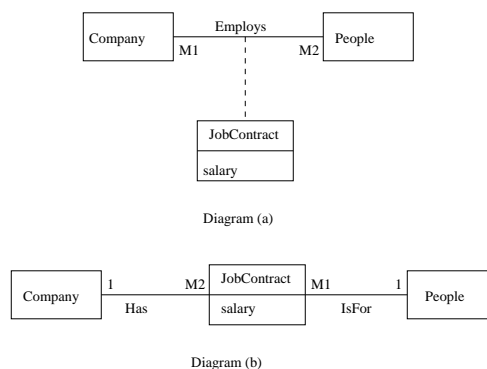


Figure 2: Example of an association class

In general, we model an association class **Aclass** for an association $A : (M_1, C_1, C_2, M_2)$ by introducing two fresh associations

$$A_1 : (\{1\}, C_1, \mathbf{Aclass}, M_2) \text{ and } A_2 : (M_1, \mathbf{Aclass}, C_2, \{1\})$$

such that $A_1 \circ A_2 = A$. The decomposition also changes the many-to-many association into one-to-many associations that are much easier to realize in the later design. This treatment of association classes can be also used in applications where an association class relates a number of classes.

Aggregations An *aggregation* can be safely treated as a general association. Its special properties, such as the visibility of the *whole* class to its *part* classes, are more relevant in the design and implementation and thus should be deferred till the time when we deal with the design. However, the property that a *part* of a *composite*

whole exists if and only when the whole itself exists, i.e. parts are created or destroyed when the whole is created or destroyed, can be specified as a state invariant. Assume **Composite** and **Part₁, ..., Part_n** are classes, and $IsPart_i : (\{m_i\}, \mathbf{Part}_i, \mathbf{Composite}, \{1\})$ are associations in a conceptual class diagram. We say that **Composite** is *composite aggregation* of **Part_i**, $i = 1, \dots, n$, if the following state invariants are true

$$\begin{aligned} HasParts &\stackrel{def}{=} \\ \forall c \in Composite &\Rightarrow \quad \exists o_{11} \dots o_{1m_1} \in Part_1, \dots o_{n1}, \dots o_{nm_n} \in Part_n \bullet \\ &IsPart_1(o_{11}, c) \wedge \dots \wedge IsPart_1(o_{1m_1}, c) \\ &\wedge \dots \\ &\wedge IsPart_n(o_{n1}, c) \wedge \dots \wedge IsPart_n(o_{nm_n}, c) \end{aligned}$$

$$HasWhole \stackrel{def}{=} \bigvee_{i=1}^n \exists o \in Part_i \Rightarrow \exists c \in Composite \bullet IsPart_i(o, c)$$

$$\begin{aligned} NoShare &\stackrel{def}{=} \\ &\bigwedge_{i=1}^n (\forall c_1, c_2 \in Composite, o \in Part_i \bullet IsPart_i(o, c_1) \wedge Part_i(o, c_2) \Rightarrow c_1 = c_2) \end{aligned}$$

And these invariants indicates a decision later in design and implementation that the whole should the visibility to the parts and parts are realized as *attributes* of the whole class.

Please note that there are many discussions about the meaning of an aggregation, particular on the *pUML mailing list* (see www.puml.org). Here we have provided a way of formally defining the semantics rather than fixing it by the above three invariants. One can of course have different constraints if they fit in the application better. If one wants to avoid any confusion, we suggest the use of a general association and specify the constraints as required.

5 Use-Case Model

A use case model consists of a use-case diagram and a textual description of each use case in the use case diagram. As said earlier, a use-case diagram provides only static information about the use cases. The dynamic semantic aspects are described in the textual descriptions of the use cases as sequences of interactions between actors and the system. Therefore, for a formal design it is more important to formalize the textual description.

An actor of a use case can be any entity external to the system. It interacts with the system by calling a *system operation* to request a service of the system. The system may also require services from actors to carry out a requested service. A UML *system sequence diagram* is used to describe the order of the interaction between the actors of in a use case and the system treated as a black box, but it does not describe the change of the system state caused by such an interaction. It is important to note that a system sequence diagram does not and should not provide information about interaction among objects inside the system [DW98, Lar01]. The main task in the system design is in fact to *realize* the use cases by interactions among objects inside the system. This is done by decomposing the responsibilities of the use cases and assigning them to objects as methods.

5.1 System operations

When an actor calls a system operation to carry out a step of a use case, the execution of the called operation changes the system state, by creating new objects, deleting existing objects; forming or breaking links between objects; modifying attributes of objects. We therefore treat the system under consideration as a *component* and a system operation a *provided method* of the component [CD01]. We model this component as a *use-case handler class* [Lar01] that encapsulates the classes in the conceptual model:

```

Class Use-Case-Name-Handler {
  Attr :  $\underline{x} : \mathbf{T}$ ;
  Method :  $op_1(\mathbf{val} x_1 : \mathbf{T}_{11}, \mathbf{res} y_1 : \mathbf{T}_{21})\{c_1\}$ ;
           ...;
  Method :  $op_n(\mathbf{val} x_n : \mathbf{T}_{1n}, \mathbf{res} y_n : \mathbf{T}_{2n})\{c_n\}$ 
}

```

where the attributes \underline{x} may include state control variables so that the use case can be defined by a state machine; and for each method $op_i(\mathbf{val} x_i : \mathbf{T}_{1i}, \mathbf{res} y_i : \mathbf{T}_{2i})\{c_i\}$, $\mathbf{val} x_i$ is a list of value parameters and $\mathbf{res} y_i$ a list of result parameters. The command c_i in an operation allows us to specify the effect of the operation at different levels of abstractions and is in one of the following forms:

$$\begin{array}{l}
c ::= d \mid x := e \mid c; c \\
\mid \mathbf{var} x : \mathbf{T} \mid \mathbf{end} x \quad \text{variable declaration and undeclaration} \\
\mid c \triangleleft b \triangleright c \quad \text{conditional} \\
\mid b * c \quad \text{iteration} \\
\mid c \sqcap c \quad \text{non-deterministic choice} \\
\mid Actor.m \quad \text{a call to a required method of an actor Actor} \\
\mid o.\bar{a}(y) \mid o.a(e) \quad \text{reading and resetting attribute}
\end{array}$$

where d is a framed design, b is a Boolean expression and e is an expression. In general, an expression can be in one of the following forms:

$$e ::= x \mid \mathbf{null} \mid \mathbf{new} C \mid \mathbf{self} \mid e.a \mid f(e)$$

We use a command $\mathbf{var} x : \mathbf{T}$ to introduce local variables in a block, and a command $\mathbf{end} x$ to end the variable block. A set of program variables, denoted by \mathbf{locvar} , is needed to record the set of local variables in scope in a state. The value of \mathbf{locvar} is of the form $\{v_1 : \mathbf{T}_1, \dots, v_m : \mathbf{T}_m\}$.

5.2 Actors

An actor of a use case calls system operations in the use case controller. However, some actors provide services to the system too. We thus can treat an actor as a component which may provide services as well

as to request services. As we are only required to design the system under consideration, we only specify an actor's services required by the system in terms of methods in the actor class.

```

Class Actor {
  Attr :  $x : \underline{\mathbf{T}}$ ;
  Method  $m_1(\text{val } x_1 : \mathbf{T}_{11}, \text{res } y : \mathbf{T}_{12})\{c_1\}$ ;
  ...;
  Method  $m_k(\text{val } x_k : \mathbf{T}_{k1}, \text{res } y_k : \mathbf{T}_{k2})\{c_k\}$ ;
}

```

where attributes $x : \underline{\mathbf{T}}$ may include control state variables; m_i is a method that can be called by the system, but the command c_i in such a method is only a framed design to avoid infinite recursive method calls.

5.3 System specification

To specify a use case H , we specify its handler **H-Handler** and actors $\text{Actor}_1, \dots, \text{Actor}_m$. Putting these together we have a specification $\text{Spec}(H)$. Suppose that for the design of a system S we have identified the use cases H_1, \dots, H_ℓ and constructed a conceptual model CM . We combine the specification of the conceptual model CM , the use-case handlers and the actors $CM; \text{Spec}(H_1); \dots; \text{Spec}(H_\ell)$ to form the *structural specification* of the system, i.e. the conceptual model, the use-handlers, and its environment that consists of the actors classes. This specification can be illustrated graphically in the diagram in Figure 3.

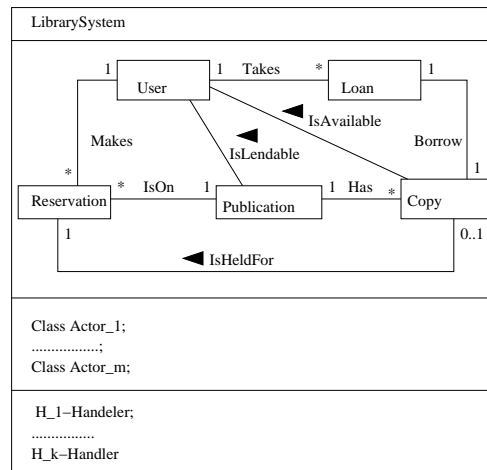


Figure 3: A system specification

Each *scenario* or *instance* of a use case is an execution sequence of calls (by actors) to methods in the use-case handler with given input values to the **val** parameters. To describe all possible scenarios of all use cases, we introduce a set **IN** of variables for the input values and a set **OUT** of variables for the output values. Let **L** be a set of *control variables* that are used to control the order of the execution of the use case actions (that will be formally defined soon). In general, scenarios of a use case can be executed concurrently and this needs multiple instances of the use-case handler class.

Let the system state variables be those declared in the conceptual model **CVar**, that now includes the use-case

handler and the actor classes, **AVar** that now includes an association between a use-case handler and an actor if the use-case handler requires services from the actor.

Now define $\alpha \stackrel{def}{=} \mathbf{AVar} \cup \mathbf{CVar} \cup \mathbf{IN} \cup \mathbf{OUT} \cup \mathbf{L}$. Let Φ be the state constraint of the conceptual model conjoining with the condition that each actor class has a single instance. We use a predicate Θ to specify the initial states for starting up the system. Θ has to imply some instances of each use-case handler class **H-Handler**, and the instance of each actor class **Actor**, denoted by *Actor*, have already been created.

A *use case action* is a guarded command $g \longrightarrow c$, where g is a boolean expression over $\mathbf{IN} \cup \mathbf{OUT} \cup \mathbf{L}$ and c is a command on $\mathbf{IN} \cup \mathbf{OUT} \cup \mathbf{L}$ to process the input and output values, or a call $h.op$ to a method op of an instance of a use-case handler **H-Handler**. A *use case* is a set U_h of use case actions that contains calls to the same use-case handler instance h . Let P be a set of use cases, a *system requirement specification* S is then of the *canonical form*

$$Spec(S) \stackrel{def}{=} CM; Spec(H_1); \dots; Spec(H_\ell) \bullet P$$

We thus have obtained the transition system model $(\alpha, \Phi, \Theta, P)$ for the system S .

In the case for sequential programs, only one instance h of a use-case handler **H-Handler** is needed and $h.op$ is simply written as op . Then each use case U_H is a piece of sequential program, and whole program P is an iterative deterministic choice among the use cases:

$$\neg stop * (read(service); \mathbf{if} \{service = H \longrightarrow U_H\} \mathbf{fi}; read(stop))$$

where $read(x) \stackrel{def}{=} true \vdash x' \in type(x)$.

A call to a system operation $m(\mathbf{val} x, \mathbf{res} y)$ can also be written as a CSP-like process

$$m?x \longrightarrow m(x, y); m!y$$

Then use case H as a whole can be written as a CSP process U_H and the program P in the canonical system specification can be specified as an iterative process $(U_{H_1} \square \dots \square U_{H_\ell})^*$.

Therefore our methodology is:

- For a sequential software development, after the system operations are identified and specified in the use-case handler classes, writing the formal specification of the use cases becomes writing a specification of the *main method* P of the object-oriented software.
- For a concurrent system, writing the formal specification of the use cases is to write the specification of the *run methods* of the concurrent actors that requires services from the system.
- However, as suggested in RUP and UML, the development takes a sequential view first and treats concurrency in the implementation stage by using *activity diagrams*. Then the design and implementation of the system is mainly to design and implement the system operations by decomposing them into interactions between objects of the system.

This is a typical top-down development, but the use cases and system operations can be taken in turns in an iterative process.

Example For the library application, use case *BorrowCopy* is about how an actor *Borrower* can borrow a copy of a publication. Obviously, its pre-conditions are: an user and the copy are currently known to the system, the publication of the copy is lendable to the user, and the copy is currently available. The effect of the use case is to create a loan to record the fact that the user has taken this loan on the copy, and the copy not available anymore. The system has to record the date of the loan. For this, the system needs to interact with the system clock to get the date. Therefore this use case has another actor, and we call it *Clock* that is *partially* specified here as follows

```

Class Clock {
  Attr : Date date, Time time;
  Method : getDate(res out : Date){true ⊢ out' = date}
}

```

The use-case handler can be given below

```

Class BorrowCopy-Handler {
  Method BorrowCopy(val String Cid, String Uid){
    Pre  $\exists c \in Copy, u \in User \bullet c.Id = Cid \wedge u.Id = Uid \wedge$ 
       $IsLendableTo(Has(c), u) \wedge IsAvailable(c, u)$ ;
    var Loan  $\ell$ ;  $\ell := New Loan()$ ;
    var Date date;  $Clock.getDate(date)$ ;  $\ell.(date)$ ; end date;
    Loan := Loan  $\cup \{\ell\}$ ;
    Borrow := Borrow  $\cup \{< \ell, c >\}$ ; Takes := takes  $\cup \{< u, \ell >\}$ ;
    IsAvailable := IsAvailable /  $\bigcup_{u \in User} \{< c, u >\}$ 
  end  $\ell$ 
}
}

```

Strictly speaking, use case *BorrowCopy* is not well-formed regarding to the given conceptual model in Section 4.1, as class **Copy** has no attribute *Id* declared. However, if we refine (following a refinement rule in [HLL02]) the conceptual model by adding this attribute to class **Copy**. The specification becomes well-formed.

Introducing input variables *Uid* and *Cid* and assuming *BorrowCopy-Handler* is the instance of the given use-case handler class already created, use case *BorrowCopy* is programmed by the following statement:

```
read(Uid, Cid); BorrowCopy(Cid, Uid)
```

This corresponds to a system sequence diagram in which actor *Borrower* calls method *BorrowCopy* of the system and the system calls *getDate()* method of the clock.

Suppose we declare in the use-case handler class three operations *FindU()* that finds the user for a given user identifier, *FindC()* that finds the copy for an input copy identifier, and *RecordL()* that checks whether

a loan can be made and records it if so. The *BorrowCopy* use case can be refined to or *zoomed in* [DW98] the following use cases

$$\begin{aligned} & read(Uid, Cid); Find(Uid, u); FindC(Cid, cp); \\ & RecordL(u, cp) \triangleleft (u \neq null \wedge cp \neq null) \triangleright \perp \end{aligned}$$

The system calls the *getDate()* method of the clock in *RecordL()*. Thus the above program corresponds to the system sequence diagram in which actor *Borrower* calls method *FindUser()*, then *FindC()* and then *RecordL()* of the system, after receiving the call of *RecordL()*, the system calls *getDate()* of actor *Clock*. This sequence diagram is a refinement of the earlier one. This refinement is carried out without changing the underlying conceptual model.

Another dimension of refinement is to refine the conceptual model together with the use cases. For example, the association *Isavailable* can be realized by a boolean attribute *IsA* of class **Copy** such that for every copy *c* and every user *u*, *Isavailable(c, u)* iff *IsA = true*. Then checking the availability off a copy becomes simply checking this variable. This kind of refinement corresponds the traditional data refinement. Both refinements we have just discussed are not involved with allocation of parts of the computation of a system operation to internal objects of the system [HLL02]. We have a *delegation rule* in [HLL02] to deal with delegating part of a method of an object to an object in another class. Formal transformation of a use-case model into a UML object sequence diagrams is out of the scope of this paper.

6 Semantics

This section defines a semantics of a system specification. We start with showing how to validate an expression and determine its value.

6.1 Expressions

To validate an expression *e*, we introduce a predicate $W(e)$ which is true just in those circumstances in which *e* can be successfully evaluated.

A state binds variables in α to their values. A variables of type **DT** simply takes a value of that type. However, a variable of a class **C** takes an object of **C** as it value. An object is defined in terms of its *identity*, values of its attributes, and its current type:

$$\{identity \mapsto id\} \cup \{class \mapsto \mathbf{C}\} \cup \{a \mapsto value \mid a \in \mathbf{attr}(\mathbf{C})\}$$

For the identities of objects, we require that $o_1(identity) = o_2(identity)$ iff $o_1 = o_2$. $W(e)$, $type(e)$, and the value of *e* are defined as follows:

- *x* is well-formed if it is known in the environment of the use case: $W(x) \stackrel{def}{=} x \in \alpha$.

- $W(\mathbf{null}) \stackrel{def}{=} true$, and $type(\mathbf{null}) \stackrel{def}{=} \mathbf{NULL}$. \mathbf{NULL} is a reserved class name and a subclass of all classes. The identity of \mathbf{null} is undefined and \mathbf{null} is the only object without an identity.
- The variable \mathbf{self} is only used as local variable when defining $o.\bar{a}(y)$ and $o.a(e)$:

$$W(\mathbf{self}) \stackrel{def}{=} \mathbf{self} \in \mathbf{locvar}$$

- $\mathbf{new C}$ is well-formed if \mathbf{N} is declared:

$$\begin{aligned} W(\mathbf{new C}) &\stackrel{def}{=} \mathbf{C} \in \mathbf{CN} & type(\mathbf{new C}) &\stackrel{def}{=} \mathbf{C} \\ \mathbf{new C}(\mathbf{class}) &\stackrel{def}{=} \mathbf{C} \\ \mathbf{new C}(\mathbf{identity}) &\notin \{id \mid \exists o \in C \bullet o(\mathbf{identity}) = id\} \end{aligned}$$

$\mathbf{newC}(\mathbf{identity})$ is *fresh* and this will be also specified by $\mathbf{newC} \notin C$.

- $e.a$ is well formed if e is an object, and a is a declared attribute of the class of e :

$$\begin{aligned} W(e.a) &\stackrel{def}{=} W(e) \wedge type(e) \in \mathbf{CN} \wedge a \in \mathbf{attr}(type(e)) \\ type(e.a) &\stackrel{def}{=} type(type(e).a) \\ e.a &\stackrel{def}{=} e(a) \end{aligned}$$

- Well-formedness of built-in expressions $f(e)$ is defined by the building rules. For example, for two association variables $A_i : (M_{i1}, \mathbf{C}_{i1}, \mathbf{C}_{i2}, M_{i2}), i = 1, 2$,

$$W(A_1 \circ A_2) \stackrel{def}{=} W(A_1) \wedge W(A_2) \wedge \mathbf{C}_{21} \leq \mathbf{C}_{12}$$

This means if we want to derive a composition association from two given associations A_1 and A_2 the target class \mathbf{C}_{12} of A_1 should be the source class \mathbf{C}_{21} or a superclass of it.

6.2 Commands

Each command c is defined as a predicate of the form $W(c) \Rightarrow D(c)$. $W(c)$ is true when the command is well-formed in the initial state and captures the consistency of the conceptual model with the command. $D(c)$ is of the form a framed design $V : p(x) \vdash R(x, x')$ and captures the dynamic behavior of the command c . This integrates syntactic consistency and semantic consistency check mechanism with the traditional specification-oriented semantics. Let $skip \stackrel{def}{=} \emptyset : (true \vdash true)$ be the command that does nothing and terminates successfully, and the command $chaos \stackrel{def}{=} \emptyset : (false \vdash true)$ that has unpredictable behaviour:

Let P and Q be designs. The notation $P \triangleleft b \triangleright Q$ describes a design which behaves like P if b is true in the initial state, and like Q otherwise.

$$P \triangleleft b \triangleright Q \stackrel{def}{=} W(b) \wedge (type(b) = \mathbf{B}) \Rightarrow (P \wedge b \vee Q \wedge \neg b)$$

We use the condition $\mathbf{Pre} b \stackrel{def}{=} (skip \triangleleft b \triangleright chaos)$ to represent a Floyd assertion, which behaves like $chaos$ if the initial value of b is false, otherwise it has no effect. This is useful when we specify the precondition of a use case.

Let $\{P_i | 1 \leq i \leq n\}$ be a family of designs. The multiple conditional choice

if $\{(b_i \rightarrow P_i) | 1 \leq i \leq n\}$ **fi**

selects P_i to execute if its guard b_i is true. When all the guards are false it behaves like *chaos*:

$$\mathbf{if} \{(b_i \rightarrow P_i) | 1 \leq i \leq n\} \mathbf{fi} \stackrel{def}{=} \bigwedge_{i:1\dots n} (W(b_i) \wedge type(b_i) = \mathbf{B}) \Rightarrow (\bigvee_{i:1\dots n} (b_i \wedge P_i) \vee \neg(\bigvee_{i:1\dots n} b_i) \wedge chaos)$$

For the non-determinism, let P and Q be designs. $P \sqcap Q \stackrel{def}{=} P \vee Q$ stating that it behaves either like P or Q .

The command $P; Q$ is executed by first executing P followed by executing Q when P terminates. The final state of P is passed on as the initial state of Q .

$$P(x, x'); Q(x, x') \stackrel{def}{=} \exists m \bullet P(x, m) \wedge Q(m, x')$$

If b is a condition, the iteration $b * P$ repeats P as long as b is true before each iteration:

$$b * P \stackrel{def}{=} \mu X \bullet (P; X) \triangleleft b \triangleright skip$$

where $\mu X \bullet F(X)$ denotes the weakest fixed point of the recursive equation $X = F(X)$.

The semantics of an assignment is defined by the following design:

$$x := e \stackrel{def}{=} W(x) \wedge W(e) \wedge (type(e) \leq type(x)) \Rightarrow \{x\} : (true \vdash (x' = e))$$

Declaration $\mathbf{var} x : \mathbf{T}$ introduces a new program variable x to allow x of type \mathbf{T} to be used in the portion of the program that follows it. The complementary command takes the form **end** x . It terminates the region of allowable use of x :

$$\begin{aligned} \mathbf{var} x : \mathbf{T} &\stackrel{def}{=} x \notin \alpha \Rightarrow \mathbf{locvar} : (true \vdash \mathbf{locvar}' = \mathbf{locvar} \cup \{x : \mathbf{T}\}) \\ \mathbf{end} x &\stackrel{def}{=} x \in \mathbf{locvar} \Rightarrow \mathbf{locvar} : (true \vdash \mathbf{locvar}' = \{x\} \trianglelefteq \mathbf{locvar}) \end{aligned}$$

where $\{x\} \trianglelefteq \mathbf{locvar}$ denotes the set \mathbf{locvar} after removing variable x . For convenience, we allow variables to be declared together in a list $\mathbf{var} x_1 : \mathbf{T}_1, \dots, x_k : \mathbf{T}_k$ as the short hand of $\mathbf{var} x_1 : \mathbf{T}_1; \dots; \mathbf{var} x_k : \mathbf{T}_k$.

To avoid direct access to object attributes in order for further implementation, we use an attribute reading command to get the value of an object's attribute, and an attribute resetting command to update an object's

attribute. This is important for further refinement. The behavior of these two commands are defined as follows.

$$\begin{aligned}
o.\bar{a}(y) &\stackrel{def}{=} (W(o) \wedge W(y) \wedge type(o) \in \mathbf{CN} \wedge a \in \mathbf{attr}(o(class))) \Rightarrow \\
\text{if } \{ (o(class) = \mathbf{C}) \rightarrow &\left(\begin{array}{l} \mathbf{var } z : \mathbf{T}, \mathbf{self} : \mathbf{C}; \mathbf{self} := o; \\ \{z\} : (\mathbf{T} = type(\mathbf{C}.a) \vdash z' = \mathbf{self}.a); \\ y := z; \mathbf{end } z, \mathbf{self} \end{array} \right) \\
| \mathbf{C} \in \mathbf{CN} \} \mathbf{fi} \\
o.a(e) &\stackrel{def}{=} (W(o) \wedge W(e) \wedge type(o) \in \mathbf{CN} \wedge a \in \mathbf{attr}(o(class))) \Rightarrow \\
\text{if } \{ (o(class) = \mathbf{C}) \rightarrow &\left(\begin{array}{l} \mathbf{var } z : \mathbf{T}, \mathbf{self} : \mathbf{C}; z := e; \mathbf{self} := o; \\ \{\mathbf{self}.a\} : (\mathbf{T} = type(\mathbf{C}.a) \vdash \mathbf{self}'.a = z); \\ o := \mathbf{self}; \mathbf{end } z, \mathbf{self} \end{array} \right) \\
| \mathbf{C} \in \mathbf{CN} \} \mathbf{fi}
\end{aligned}$$

6.3 Declarations

In Section 4.4, a syntax of the specification of a conceptual model was given, which is composed from class declarations and association declarations. We treat the semantics of a declaration in the same way as treating a command, i.e. we define its semantics as a design.

A class declaration **Calass** \mathbf{C}_1 **Extends** $\mathbf{C}_2 \{ \mathbf{T}_1 a_1; \dots; \mathbf{T}_m a_m \}$ modifies the logic variables **Cvar**, **super** and **Attr** according to the following design

$$\left(\begin{array}{l} \mathbf{C}_1, \mathbf{C}_2 \in \mathbf{CN} \\ \wedge \mathbf{C}_2 \in \mathbf{CVar} \\ \wedge \bigwedge_{i=1}^m a_i \notin \mathbf{Attr}(\mathbf{C}_2) \end{array} \right) \vdash \left(\begin{array}{l} \mathbf{CVar}' = \mathbf{CVar} \cup \{ \mathbf{C}_1 : \mathbb{P}\mathbf{C}_1 \} \\ \wedge \mathbf{super}' = \mathbf{super} \cup \{ \mathbf{C}_1 \mapsto \mathbf{C}_2 \} \\ \wedge \mathbf{Attr}' = \mathbf{Attr} \cup \\ \{ \mathbf{C}_1 \mapsto \{ a_i : \mathbf{T}_i | 1 \leq i \leq m \} \cup \mathbf{Attr}(\mathbf{C}_2) \} \end{array} \right)$$

This design says that the declaration declares a new class \mathbf{C}_1 as a subclass of \mathbf{C}_2 and the newly added attributes of \mathbf{C}_1 .

Similarly, an association declaration **Association** $(M_1, \mathbf{C}_1, \mathbf{C}_2, M_2)$ A adds a new association to **AVar** and the state constraints on this association into the set **Invariant**:

$$\mathbf{C}_1, \mathbf{C}_2 \in \mathbf{CVar} \wedge A \notin \mathbf{AVar} \vdash \left(\begin{array}{l} \mathbf{AVar}' = \mathbf{AVar} \cup \{ A : \mathbb{P}(\mathbf{C}_1 \times \mathbf{C}_2) \} \\ \wedge \mathbf{Invariant}' = \mathbf{Invariant} \\ \wedge Multiplicity(A) \wedge LinkObjects(A) \end{array} \right)$$

A declaration of an actor or a use-case handler introduces methods as well as class variables and attributes. For this, we introduce a logic variable **Meth** that is a function from **CN** to the set of method definitions of the form

$$op(\mathbf{var } x : \mathbf{T}_1, \mathbf{res } y : \mathbf{T}_2) \{c\}$$

Now, given a declaration of an actor or a use-case handler class

```

Class A {
  Attr :  $\underline{x} : \underline{\mathbf{T}}$ ;
  Method  $m_1(\mathbf{val} x_1 : \mathbf{T}_{11}, \mathbf{res} y_1 : \mathbf{T}_{12})\{c_1\}$ ;
  ...;
  Method  $m_k(\mathbf{val} x_k : \mathbf{T}_{k1}, \mathbf{res} y_k : \mathbf{T}_{k2})\{c_k\}$ ;
}

```

Its semantics is defined by the design

$$\mathbf{A} \in \mathbf{CN} \vdash \left(\begin{array}{l} \mathbf{CVar}' = \mathbf{CVar} \cup \{\mathbf{A}\} \\ \wedge \mathbf{Attr}' = \mathbf{Attr} \cup \{\mathbf{A} \mapsto \underline{x} : \underline{\mathbf{T}}\} \\ \wedge \mathbf{Meth}' = \mathbf{Meth} \cup \{\mathbf{A} \mapsto \{m_i def \mid 1 \leq i \leq k\}\} \end{array} \right)$$

where $m_i def$ is $m_i(\mathbf{var} x_i : \mathbf{T}_{1i}, \mathbf{res} y_i : \mathbf{T}_{2i})\{c_i\}$.

According to the semantics of sequential definition given in the previous subsection, the semantics of the structural specification of a system $CM; Spec(H_1); \dots; Spec(H_\ell)$ given in Section 5.3 is now well defined and calculates the alphabet α and the state invariant Φ of the transition system $S = (\alpha, \Phi, \Theta, P)$.

6.4 Call to a required method

Let $vale$ and $rese$ be lists of expressions. A call $Actor.m(vale, rese)$ to a method of an actor assigns the values of the actual parameters $vale$ to the formal value parameters $\mathbf{val} x$ of the declared method m . After it terminates, the values of the result parameters $\mathbf{res} y$ of op are passed back to the actual value parameters $rese$.

$$Actor.m(vale, rese) \stackrel{def}{=} m \in \mathbf{Meth}(type(Actor)) \Rightarrow \left(\begin{array}{l} \mathbf{var} x : \mathbf{T}_1, y : \mathbf{T}_2; \\ x := vale; y := rese; \\ type(Actor).m; \\ rese := y; \mathbf{end} x, y \end{array} \right)$$

where

- $\mathbf{Meth}(type(Actor))$ is the set of methods declared in the class of $Actor$.
- x, y are the value and result parameters of the method m .
- $type(Actor).m$ stands for the design associated with the command of m defined in the actor class $type(Actor)$.

Notice that $type(Actor) = \mathbf{Actor}$.

6.5 System specification

An action of a use case that is a call to a method of a use-case handler $h.op(val, res)$ is defined similar to a call to a method of an actor by a use-case handler:

$$op \in \mathbf{Meth}(type(h)) \Rightarrow \left(\begin{array}{l} \mathbf{var} \ x : \mathbf{T}_1, y : \mathbf{T}_2; \\ x := val; y := res; type(type(h)).op; \\ rese := y; \mathbf{end} \ x, y \end{array} \right)$$

The semantics of a guarded action $g \longrightarrow c$ is $W(g) \Rightarrow g \wedge c$, where $W(g)$ is true g is a Boolean expression over $\mathbf{IN} \cup \mathbf{OUT} \cup \mathbf{L}$.

We define the semantics of a system $S = (\alpha, \Psi, \Theta, P)$ be the set of infinite state sequences

$$\llbracket \mathbf{S} \rrbracket \stackrel{def}{=} \{\sigma_0, \sigma_1 \dots, \mid \sigma_i \in \Sigma_\alpha\}$$

such that

- Σ_α is the set of states over α .
- σ_0 satisfies the initial condition Θ and each state transition (σ_{i-1}, σ_i) is carried out by an enabled call to a method in the use-case handler, i.e. there an action $U \in P$ such that $(\sigma_{i-1}, \sigma_i) \models U$.
- Ψ is a proof obligation that each action $U \in P$ preserves this invariant in the sense that $(g \wedge Pre(U) \wedge \Psi) \Rightarrow (Post(U) \Rightarrow \Psi')$, where g is the guard of action U , $Pre(U)$ is the precondition and $Post(U)$ the post-condition of design U , and Ψ' is the predicate obtained from Ψ by replacing each free variable x with it primed version x' .

A *refinement* of a system can be carried out by refining a method in a use-case handler.

7 Conclusion & Related Work

7.1 Conclusion

We have given a relational model for UML conceptual diagrams and use cases. The conceptual model of a system declares the system variables, and its object diagrams form the system state space. We formally define a use-case model of a system to describe the interactions between the system and its external environment that consists of the actors of the use cases. Both the actors and the system are treated as components that have *required services* as well as *provided services*. The execution of a call to a system service by an actor will cause a change of the system state with some new objects created, some existing objects deleted, some new links between object established, some existing links between objects removed, and values of some object attributes modified. It enhances RUP for OO software development with a formal method roughly as follows:

we first write a use case informally; then construct a conceptual model based on the use case; then draw a system sequence diagram and identify the methods of the use-case handler class; transform the conceptual model into a formal specification and formally specify these methods are in the notation provided in this paper; then check the consistency of these methods with the conceptual model with the method provided in this paper; refine the conceptual model and the use-case specification if they are not consistency; for an executable specification¹, test it by running it for some input values; finally we can take this specification into the design, implementation, and testing. This completes a cycle. Then new use cases can then be specified, analyzed, and the existing conceptual model is refined to support the newly added use cases. During the design of a new use case, one can reuse the methods of classes that have already been designed. For more details about the integration of the formal method in this paper with RUP can be found in [LLH03].

The formalism is based on the design calculus in Hoare and He's Unifying Theories of Programming [HH98]. Some of the mathematics may seem rather theoretical at first, but the approach is quite practical. For example, the choice of a java-like syntax for the specification language is a pragmatic solution to the problems of representing name spaces and (the consequences of) inheritance in a notation such as CSP.

7.2 Related work

Instead of taking a process view, such as that in [FOW01, DC02], we keep an object-oriented and state-based approach and the specification in a java-like style. We specify consistency between the models in the preconditions in terms of well-formedness conditions of use cases.

Our work [LLH02] establishes the soundness and (implicit) the completeness of the action systems for both conceptual and use-case modelling. This paper, extend that work with a formal notation for the specification. Our related work [LLH01] demonstrates that our method supports stepwise refinement and reflects the informal way of using UML for requirement analysis. Use-case refinement can be carried out using the traditional refinement calculus of imperative programs, and can be done by refining the methods provided in the use case-case handlers one by one [HLL02].

We take a similar schema of the semantics of UML proposed in [RCA01], where different kinds of diagrams of a UML model are given individual semantics and then such semantics are composed to get the semantics of the overall model. However, unlike [RCA01] that uses an universal algebraic approach, we use a simple predicate theory to define a specification language with a syntax similar to Java programming language. We believe that this feature makes the method more accessible to people who are familiar to the general theory of programming languages. The main difference between our work and that in [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, but it does not consider use cases. This model of associations can also be used in the specification of *connectors* in architecture models [FM96, Sel, AM02]. Our work also shares some common ideas with [BPP99] in the treatment of use cases. However, our treatment of use cases is at the system interface level without referring to the design details about how internal objects of the system behave, or what methods that a class of the system provides. We believe our model is simpler and addresses the tight relationships among different models more clearly. Also in our model, actors are not only users of the system but also service providers. We will carry out the design of the system by decomposing the methods in the use-case handlers, in turns one by one, and assign the decomposed responsibilities to classes of the conceptual model. This is the main task in the creation of UML interaction diagrams, i.e. *object sequence*

¹If the program specification is not executable but realizable, refine it into an executable specification.

diagrams or *collaboration diagrams*. The formalization of such a design within our framework in [HLL02] is given in [LLH03]. The main difference between our work and most of the work of the precise UML consortium (see www.puml.org) is that, rather than trying to formalize individual views of UML, we aim to tightly combine different views in a unified formal language, and our formal language provided built-in facilities to naturally capture the object-oriented features of UML, rather than using a traditional formalism which were not designed for object-oriented systems to derive the definitions of classes, objects, inheritance, etc.

In this paper, we have focused on only conceptual aspects of object orientation. The transformation of a UML conceptual model into a formal specification can be easily automated. We can refine a use case specified in our notation to executable specification without the need the the details of inter-object interaction or the internal behavior of objects. We can thus execute the use case to generate the post state (i.e. the post object diagrams) from the pre-state (i.e. pre-object diagram) of the use case to validate the use case and to check the consistency between the use case and the conceptual model. Then based on the generated object-diagrams and our work on the refinement calculus for object systems [HLL02], we can decompose the use cases by “zooming them in” [DW98]. In [LLH03], we have developed a method for automatic generation of a prototype from a formal specification of a UML requirement model. A prototype generated that way can be used for the validation of a use-case model and a conceptual model, as well as to check the state invariants and the consistency between a use-case model and a conceptual model.

Acknowledgement We would like to thank Chris George for his useful comments on the work, and the referees for their comments that has helped us to improve the presentation.

References

- [AM02] N. Aguirre and T. Maibaum. A temporal logic approach to component-based system specification and verification. In *Proc. ICSE'02*, 2002.
- [BPP99] R.J.R. Back, L. Petre, and I.P. Paltor. Formalizing UML use cases in the refinement calculus. Technical Report 279, Turku Centre for Computer Science, Turku, Finland, May 1999.
- [CD01] J. Cheesman and J. Daniels. *UML Components. Component Software series*. Addison-Wesley, 2001.
- [DC02] J. Davies and C. Crichton. Concurrency and refinement in the unified modelling language. In *Preliminary Proceedings of REFINER'02: An FME Sponsored Refinement Workshop in Collaboration with BCS FACS*, Copenhagen, Denmark, 2002.
- [DW98] D. D'Souza and A.C. Wills. *Objects, Components and Framework with UML: The Catalysis Approach*. Addison-Wesley, 1998.
- [Egy01] A. Egyed. Scalable consistency checking between diagrams: The Viewintegra approach. In *Proc. of the 16th IEEE International Conference on Automated Software Engineering*, San Diego, USA, 2001.
- [EKHG01] G. Engels, J.M. Kuster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *The Proc. of International Conference on Foundation of Software Engineering, FSE-10*, Austria, 2001.
- [FELR98] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19:325–334, 1998.

- [FM96] J. Fiadeiro and T. Maibaum. Design structures for object-based systems. In S. Goldsack and S. Kent, editors, *Fomal Methods and Object Technology*. Springer-Verlag, 1996.
- [FOW01] C. Fischer, E-R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In Heinrich Hussmann, editor, *4th FASE, Genova, Italy, 2001, Proceedings*, volume 2029 of *LNCS*, pages 91–108. Springer, 2001.
- [Gro99] Object Modelling Group. Unified Modelling Language Specification, version 1.3. URL: uml.shl.com:80/docs/UML.1.3/99-06-08.pdf, 1999.
- [HH98] C.A.R. Hoare and J. He. *Unifying theories of programming*. Prentice-Hall International, 1998.
- [HLL01] J. He, Z. Liu, and X. Li. A relational model for object-oriented programming. Technical Report UNU/IIST Report No 231, UNU/IIST, P.O. Box 3058, Macau, March 2001.
- [HLL02] J. He, Z. Liu, and X. Li. Towards a refinement calculus for object-oriented systems. In *Proc. ICCI02, Alberta, Canada*. IEEE Computer Socioty, 2002.
- [HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Israel, September 2000.
- [JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [Jür02] J. Jürjens. Formal semantics for interacting UML subsystems. In *FMOODS 2002*, pages 29–44, 2002.
- [Ken97] S. Kent. Constraint diagrams: Visualising invariants in object-oriented models. In *OOPSLA97*. ACM Press, 1997.
- [Kru] P. Kruchten. *The Rational Unified Process – An Introduction. 2nd edition*.
- [Lar01] C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
- [LLH01] X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *COMP-SAC01*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.
- [LLH02] Z. Liu, X. Li, and J. He. Using transition systems to unify uml models. In Chris 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.
- [LLH03] X. Li, Z. Liu, and J. He. Generating a prototype from a UML model of system requirements. Submitted for publication, 2003.
- [LLLH03] J. Liu, Z. Liu, X. Li, and J. He. Towards an integrating a formal method with the Rational Unified Process. Submitted for publication, 2003.
- [MP81] Z. Mana and A. Pnueli. The temporal framework for concurrent programs. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–274. Academic Press, 1981.
- [RCA01] G. Reggio, M. Cerioli, and E. Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *Proc. FASE 2001, number 2029, Lecture Notes in Computer Science*. Springer Verlag, 2001.

-
- [Sel] B. Selic. Using UML for modelling complex real-time systems. In F. Muller and A. Bestavros, editors, *Languages, compilers, and Tools for Embedded Systems, Volume 1474 of Lecture Notes in Computer Science*, pages 250–262. Springer Verlag.
- [WK99] J. Warmer and A. Kleppe. *the Object Constraint Language: precise modeling with UML*. Addison-Wesley, 1999.