UNU/IIST

International Institute for
Software Technology

Unifying Views of UML

Zhiming Liu, He Jifeng, Xiaoshan Li and Jing Liu

October 2003

UNU/IIST Report No. 288

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. 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.

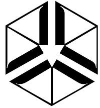
Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

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



UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Unifying Views of UML

Zhiming Liu, He Jifeng, Xiaoshan Li and Jing Liu

Abstract

We present an approach to embedding a formal method into the Rational Unified Process (RUP). The purposes are: (a) to unify different views of UML, (b) to enhance RUP and UML with the formal method to improve the quality of software systems; and (c) to support effective use of the formal method for system specification and reasoning with the iterative and incremental approach in RUP. Our overall aim is to base RUP and UML on the formal method and to scale up the use of the formal method in software system development. The model is based on Hoare and He's Unifying Theories of Programming (UTP).
Keywords: *Object-Orientation, RUP, UML, UTP*

This paper is published in the proceedings of the UML 2003 Workshop on Compositional Verification of UML, 20-24 October 2003, San Francisco, California, USA

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: hjif@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.

Jing Liu is a UNU/IIST fellow from Shanghai University of China, where she is an Associate Professor. Her research interest includes formal methods for software development. E-mail: lj@iist.unu.edu.

Copyright © 2003 by UNU/IIST, Zhiming Liu, He Jifeng, Xiaoshan Li and Jing Liu

Contents

1 Introduction **1**

2 RUP and UML **1**
2.1 Requirement analysis . . . 1
2.2 Design . . . 2
2.3 Implementation . . . 2

3 A Summary of the Approach **2**

4 The Formal Object-Oriented Specification Language **3**
4.1 Syntax . . . 3
4.1.1 Class declarations . . . 4
4.1.2 Commands . . . 4
4.1.3 Expressions . . . 5
4.2 Semantics . . . 5
4.2.1 Values and objects . . . 6
4.2.2 Variables and states . . . 7
4.2.3 Evaluation of expressions . . . 8
4.2.4 Semantics of a class declaration . . . 10
4.2.5 The well-definedness of a declaration section and semantics of a program . . . 11

5 Specification of UML Models **12**
5.1 Further development of the library system . . . 15

6 Conclusion **16**
6.1 Related Work . . . 16
6.2 Future work . . . 18

1 Introduction

Nowadays, a software system, such as one used for health care, social security, or defence, is a model (of part) of the real world. As the real world keeps changing, the software system that models it needs to be continuously maintained and evolved. To develop and maintain such an evolving software system is obviously difficult. A well disciplined process and a good modelling notation are essential to control the activities in constructing and documenting the different models obtained at different stages of the software development. The Rational Unified Process (RUP) [Kru00] has emerged as a popular software development process. As the modelling notation, RUP uses UML [BRJ99], that is the de-facto standard modelling language for the development of software in a broad range of application.

RUP promotes several best practices, but one stands above the others is the idea of *iterative development*. In the iterative approach of RUP, a system development is organized as a series of short, fixed-length mini-projects called *iterations*. Each iteration includes its own requirements analysis, design, implementation, and testing/verification activities, described in the following subsection.

Although RUP and UML are practically popular, they are not well-founded with a formal method making it hard to analyze consistency of UML specifications. This work is towards an integration of a formal method with RUP and UML.

Section 2 briefly discusses the activities and UML models in RUP. We provide a summary of the main ideas of our approach in Section 3. Section 4 introduces an OO notation that will be used in the proposed formal method. Section 5 shows the use of the specification notation in the specification of UML models. Instead of going into details of the formalization of UML, we will use a library system as an example to illustrate the treatment of models created in different cycles of the RUP. Finally, Section 6 concludes the paper with a discussion about related work.

2 RUP and UML

As said earlier, each RUP iteration includes its own requirements analysis, design, implementation, and testing and/or verification activities.

2.1 Requirement analysis

The requirements analysis of an iteration is to create a UML *model for the requirements* that contains a *use-case model* and a *conceptual class model*.

The use-case model consists of a use-case diagram and a textual description of each use case. The use-case diagram provides only static information about the use cases. The dynamic semantic aspects are described in a textual descriptions of the use cases as sequences of interactions between *actors* of the use case and the system. A UML *system sequence diagram* is used to describe the order of the interactions between the actors of 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. A formalization of a use-case model should deal with both of the order of the interactions and the state changes caused by these interactions.

The conceptual model 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 in the *application domain*. The reason why we call the class diagram conceptual at this level is that it is not concerned with what an object does, how it behaves, or how an attribute is represented. The decisions on these issues will be made during the design phase when the *responsibilities* of a use case are decomposed and assigned to appropriate objects. The UML community say that the class model represents the *static view*, whereas the use-case model represents the *dynamic view* of the requirements.

2.2 Design

The design is to transform the model of the requirements to a *model of design* that consists of a *design class diagram* and a family of *interaction diagrams*. The interaction diagrams, representing the *interactive view* of the design, show how objects of the system interact and collaborate with each other. The creation of the interaction diagrams mainly involves assignment of responsibilities to objects so that their interactions correctly realize the use cases. Use case decomposition and responsibility assignment are 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 what it can do*. What an object knows is determined by its attributes and associations with other objects.

After the responsibilities of the objects are decided, the directions of the associations (i.e. *navigation* and *visibility* from one object to another) and the methods of the classes can be determined. This will lead to the construction of the design class diagram, that shows the *static view* of the design, i.e. how the concepts and associations of the conceptual class diagram are realized by *software classes*.

2.3 Implementation

The implementation is to code the design in a programming language. In an OO programming language, this is to define the software classes from the classes in the design class diagram and their methods based on the interaction diagrams. Once the interaction diagrams and the design class diagram are obtained, code can be easily produced from it.

3 A Summary of the Approach

In this paper, we will focus on the incremental and iterative feature of RUP and address the following problems:

1. How to formalize a UML model of the requirements in an iteration and ensure the consistency between static and dynamic views of the model?
2. How to formalize a UML model of the design of an iteration and ensure the consistency between the its static and dynamic/interaction views?
3. How to formally relate the UML models of the design and the requirements of an iteration?

4. How to preserve the established consistency and correctness in sequent iterations?

We approach these problems by first developing a design calculus for OO programming. The calculus includes an OO specification language (OOL) and a refinement calculus. The calculus is based on Hoare and He's Unifying Theories of Programming (UTP) [HH98]. We will then study how UML models of requirements and designs can be formally represented and reasoned about in the design calculus.

When formalizing a UML model $RM = (cm, um)$ of requirements, we describe the static view cm as a *declaration section* $cdecls_{cm}$ and the use-case model um by a program command specification c_{um} . Therefore, RM is defined as a OO program specification $cdecls_{cm} \bullet c_{um}$. The semantics of $cdecls_{cm}$, c_{um} and their composition $\bullet$ are given in the semantics the OOL. This formalization captures both the syntax and semantics of cm and um and the consistency between them.

Similarly, for a UML model of design $DM = (dc, sd)$ consisting of a design class diagram dc and a family sd of sequence diagrams, we formalize the design class diagram dc with a declaration section $cdecls_{dc}$ in the OOL. Classes in this declaration section now have methods and a method of a class may call methods of other classes. Therefore, the specification of these methods describes the object interactions in the sequence diagrams. However, methods are still to be activated by commands in the main program c_d. Therefore, a UML model of design (dc, sd) is also specified as the composition of a declaration section and a main program: $cdecls_{dc} \bullet c_d$. The consistency between the class diagram dc and the object sequences diagrams sd are captured by the semantics of $cdecls_{dc}$ and the semantics of method calls in the OOL. The *correctness* of the design model (dc, sd) $w.r.t$ the requirements model (cm, um) is defined by the *refinement* relation

$$cdecls_{cm} \bullet c_{um} \sqsubseteq cdecls_{dc} \bullet c_d$$

Such an integration of the refinement calculus with RUP makes the use of the design calculus more effective in an iterative and incremental manner so that only a small model will be treated at each stage of an iteration.

4 The Formal Object-Oriented Specification Language

We develop OO language with classes, references, visibility, dynamic binding, nested declaration, and mutual recursive method calls. Both class declarations and commands as notion of *designs* in [HH98].

4.1 Syntax

A program is of the form $cdecls \bullet P$, where $cdecls$ is the *declaration section*, and P is a command, that can be understood as the main method of an Java program.

4.1.1 Class declarations

A *declaration section* $cdecls$ is of the form $cdecls := cdecl \mid cdecls; cdecl$, where $cdecl$ is a *class declaration* of the following form

$$\begin{array}{l} \textbf{Class}\ N\ \textbf{extends}\ M\ \{ \\ \quad \textbf{private}\ \underline{U}\ \underline{a} = \underline{u},\ \textbf{protected}\ \underline{V}\ \underline{b} = \underline{v},\ \textbf{public}\ \underline{W}\ \underline{c} = \underline{w}; \\ \quad \textbf{method}\ m_1(\underline{T}_{11}\ \underline{x}_1, \underline{T}_{12}\ \underline{y}_1, \underline{T}_{13}\ \underline{z}_1)\{c_1\}; \cdots; m_\ell(\underline{T}_{\ell 1}\ \underline{x}_\ell, \underline{T}_{\ell 2}\ \underline{y}_\ell, \underline{T}_{\ell 3}\ \underline{z}_\ell)\{c_\ell\} \\ \quad \} \end{array}$$

where

- N and M are names of classes, and M is called the direct superclass of N.
- The $\mathbf{private}$ declaration declares the private attributes $\underline{a}$ of the class, their types $\underline{U}$ and initial values $\underline{u}$, and similarly, the $\mathbf{protected}$ and $\mathbf{public}$ declarations for the protected and public attributes. We define

  $$\mathbf{pri}(N) \stackrel{def}{=} \{U\ a = u \mid U\ a = u \in \underline{U}\ \underline{a} = \underline{u}\}$$

  and similarly $\mathbf{pro}(N)$ and $\mathbf{pub}(N)$. We use $\mathbf{attr}(N)$ to denote the union of these three sets of attributes; and for an attribute a of N, we use $\mathbf{dtype}(N.a)$ to denote the type of a and $\mathbf{Init}(N.a)$ the initial value of a declared in N.
- The $\mathbf{method}$ declaration declares the methods, their value parameters $\underline{T}_{i1}\ \underline{x}_i$, result parameters $\underline{T}_{i2}\ \underline{y}_i$, value-result parameters $\underline{T}_{i3}\ \underline{z}_i$ and bodies c_i, denoted by $\mathbf{val}(N.m_i)$, $\mathbf{res}(N.m_i)$, $\mathbf{valres}(N.m_i)$, and $\mathbf{body}(N.m_i)$, respectively. We will also simply use $m(paras)\{c\}$ to denote a method declaration.

We will use the Java convention to write a class specification, and assume an attribute **protected** when it is not tagged with **private** or **public**.

4.1.2 Commands

Our language supports typical OO programming constructs:

$$\begin{array}{lll} c ::= & \mid skip \mid chaos \mid c; c & \text{termination, abort and sequnce} \\ & \mid \textbf{var}\ T\ x = e \mid \textbf{end}\ x & \text{variable declaration and undeclaration} \\ & \mid c \lhd b \rhd c \mid b * c & \text{conditional choice and iteration} \\ & \mid read(T\ x) & \text{read in value of type } T \\ & \mid le := e \mid le.m \mid C.New(x) & \text{assignment, method call and object creation} \end{array}$$

where b is a Boolean expression, e an expression, and le an expression which may appear on the left hand side of an assignment and is of the form $le := x|le.a|self$ where x a simple variable and a is an attribute of an object. We use

$$\mathbf{if}\{(b_i \longrightarrow P_i)|1 \leq i \leq n\}\mathbf{fi}$$

to denote the multiple choice statement.

4.1.3 Expressions

Expressions, which can appear on the right hand sides of assignments, are constructed according to the rules below.

$$e ::= x|null|self|e.a|f(e)$$

where $null$ represents the special object of the special class $Null$ that is a subclass of all class and has $null$ as its unique object. We can include more expression such as type casting $(C)e$ and type test (e **is** C), but they are not needed in this paper.

4.2 Semantics

In UTP [HH98], a program or a program command is identified as a *design*, which is represented by a pair (α, P), where

- α denotes the set of variables of the program.
- P is a predicate of the form

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

  where x and x' stand for the initial and final values of program variables $x \subseteq \alpha$, the predicate p, called the *precondition* of the program, characterizes the initial states in which the activation of the program will lead its execution to termination, and the predicate R, called the *post-condition* of the program, relates the initial states of the program to its final states. We describe the termination behaviour of a program by the Boolean variables *ok* and ok', where the former is true if the program is properly activated and the later becomes true if the execution of the program terminates successfully.

A program command usually modifies a subset of the program variables α. Let V be a subset of α , the notation $V : (p \vdash R)$ denotes the (*framed*) design

$$p \vdash (R \wedge \underline{w}' = \underline{w})$$

where $\underline{w}$ contains all the variables in α but those in V. V is called the frame of the design $p \vdash R$. In the examples, we often omit the frame of a design by assuming that a design only changes the value of a variable x if its primed version x' occurs in the design.

For simplicity, the above model adopts a universal data type and allows neither references types nor nested declarations. To formalize the behaviour of an OO program, we have to take into account the following features:

- A program operates not only on variables of primitive types, such as integers, but also objects of reference types.
- To protect attributes from illegal accesses, the model has to address the problem of visibility of attributes to the environment.
- An object can be associated with any subclass of its originally declared one. To validate expressions and commands in a dynamic binding environment, the model must keep track of the current type of each object.

4.2.1 Values and objects

A value is either a member of a primitive type or an *object identity*. We assume an infinite set ID of object identities, and $null \in ID$. An *object* o is an entity defined by the following structure

$$o ::= null \mid\ < id, \mathbf{type}, \mathbf{state} >$$

where $id \in ID$, and $\mathbf{type}$ is a class, and $\mathbf{state}$ is a mapping from $\mathbf{attr}(\mathbf{type})$ to objects. For an object $o =< id, C, \sigma >$, we use $identity(o)$ to denote the identity id of o, $\mathbf{type}(o)$ to denote the type C of the object o, $\mathbf{state}(o)(a)$ to denote the value $\sigma(a)$ of an attribute a of class C.

Let $\mathcal{O}$ be the set of all objects, including $null$, such that for any o_1 and o_2 in $\mathcal{O}$, $identity(o_1) = identity(o_2)$ implies $\mathbf{type}(o_1) = \mathbf{type}(o_2)$ and $\mathbf{state}(o_1) = \mathbf{state}(o_2)$. We therefore can use identity of an object in $\mathcal{O}$ to refer to the object. In the rest of the paper, an object $o =< id, C, \sigma >$ means one in $\mathcal{O}$ if there is no confusion, and will use $id.a$ to denote $\mathbf{state}(o)(a)$, and $\mathbf{type}(id)$ to denote $\mathbf{type}(o)$.

Notations: We introduce the following notations:

- A class N is said to be a subclass of M, denoted by $N \preceq M$, if either $N = M$, or there exists a finite set $\{C_i \mid 0 \leq i \leq n\}$ of classes such that

  $$N = C_0,\ \ M = C_n\ \ and\ \ \mathbf{super}(C_i) = C_{i+1}, \ \ \text{for } 0 \leq i < n$$

- Let $\underline{s} =< s_1, \ldots, s_k >$ be a non-empty sequence. We use $head(\underline{s})$ to denote the first element $s_1$of s; $tail(\underline{s}) \overset{def}{=} < s_2, \ldots, s_k >$ that is the sequence obtained from s by removing its first element (it can be the empty list $< >$); $|\underline{s}|$ the length k of s; and $\pi_i(\underline{s})$ the ith element s_i, for $i : 1, \ldots, k$.
- Let S and S_1 be sets. $S_1 \trianglelefteq S$ is the set with the elements of S_1 being removed from S.
- For a mapping $F : D \longmapsto E$, $d \in D$ and $r \in E$

  $$F \oplus \{d \mapsto r\} \overset{def}{=} F',\ where\ \ F'(d) = r \wedge \forall b \in \{d\} \trianglelefteq D \bullet F'(b) = F(b)$$

- For an object $o =< id, C, \sigma >$, an attribute a of C and an entity d which is either a member of a primitive type or an object in $\mathcal{O}$,

  $$id \oplus \{a \mapsto d\} \overset{def}{=} < id, C, \sigma \oplus \{a \mapsto d\} >$$

- For a set $S \subseteq \mathcal{O}$ of objects,

$$S \uplus \{< id, C, \sigma >\} \stackrel{def}{=} \{o \mid identity(o) = id\} \trianglelefteq S \cup \{< id, C, \sigma >\}$$
$$Id(S) \stackrel{def}{=} \{id \mid id \text{ is the identity of an object in } S\}$$

4.2.2 Variables and states

Our model describes the behaviour of an OO program by a design containing the following logical variables as its *free variables*.

1. $\mathbf{cn}$: its value is the set of class names which are declared so far, and it is modified by a class declaration.
2. Each class $N \in \mathbf{cn}$ is associated with
   (a) $\mathbf{attr}(N)$: the set of class N's (declared or inherited) attributes. We also use $a \in \mathbf{attr}(N)$ to denote that a is an attribute name of class N.
   (b) $\mathbf{op}(N)$: the set of class N's (declared and inherited) methods.

   $$\{m_1 \mapsto (paras_1, D_1), \cdots, m_k \mapsto (paras_k, D_k)\}$$

   which states that each method m_i has $paras_i$ as its formal parameters, and that the behaviour of m_i is defined by the design D_i referred by $\mathbf{Def}(N.m_i)$. These variables are modified by class declarations.
3. For each $N \in \mathbf{cn}$, $\Sigma(N)$ is the set of objects of class N current existing in the execution of the program, and it will be changed by creating a new object (and destroying an existing object that we do not deal with in this paper). Let

   $$\Sigma \stackrel{def}{=} \bigcup_{N \in \mathbf{cn}} \Sigma(N)$$

4. **super**: the partial function mapping a class to its $direct$ superclass. This variable is also modified by a class declaration.
5. **var:** its value is the set of variables which are known to the program. Since our language allows nested declaration, **var** associates each variable with a sequence of types

   $$\{(x_1, < T_{11}, \ldots, T_{1m} >), \cdots, (x_n, < T_{n1}, \ldots, T_{nk} >)\}$$

   where T_{i1}, for $i = 1, \ldots, n$ is the most recently declared type of x_i and denoted by $\mathbf{dtype}(x_i)$. We will also use $\mathbf{var}(x)$ to denote the sequence of types associated with x.
6. **visattr**: its value is the set of attributes which are visible from inside the current class, i.e. all its declared attributes plus the protected attributes of its superclasses and all public attributes. This value will be modified by the whole declaration of the program and by variable redeclarations.
7. $\overline{x}$: this logical variable represents the state of variable x. Since a variable can be redeclared, its state is usually a nonempty finite sequence of values, whose first (head) element represents the current value of variable x. A variable of a primitive data type can take any member of that type as its value. However, an object variable can store an object *name* or *identity* as its value.

4.2.3 Evaluation of expressions

The evaluation of an expression e determines its type $\mathbf{type}(e)$ and its value. The evaluation makes use of the state of $\Sigma(C)$ for each class $C \in \mathbf{cn}$.

- A variable x is well-defined if it is declared in **var**, its type is either primitive and then its current value is a member of this type, or a class in **cn** and in this case its current value is an identity of an object.

$$
\begin{array}{lcl}
\mathcal{D}(x) & \stackrel{def}{=} & x \in \mathbf{var} \wedge (\mathbf{dtype}(x) \text{ is primitive } \vee \mathbf{dtype}(x) \in \mathbf{cname}) \\
 & \wedge & \mathbf{dtype}(x) \text{ is primitive} \Rightarrow head(\overline{x}) \in \mathbf{dtype}(x) \\
 & \wedge & \mathbf{dtype}(x) \in \mathbf{cn} \Rightarrow head(\overline{x}) \in Id(\Sigma(\mathbf{dtype}(x))) \\
\mathbf{type}(x) & \stackrel{def}{=} & \begin{cases} \mathbf{dtype}(x) & \text{if } \mathbf{dtype}(x) \text{ is primitive} \\ \mathbf{type}(head(\overline{x})) & \text{otherwise} \end{cases} \\
\mathbf{value}(x) & \stackrel{def}{=} & head(\overline{x})
\end{array}
$$

- The $null$ object expression,

$$
\mathcal{D}(null) \stackrel{def}{=} true, \quad \mathbf{type}(null) \stackrel{def}{=} NULL, \quad \mathbf{value}(null) \stackrel{def}{=} null
$$

- $self$ is a special variable whose type has to be a class in **cn**, and it is evaluated in the same way as other variables,

$$
\begin{array}{lcl}
\mathcal{D}(self) & \stackrel{def}{=} & self \in \mathbf{var} \wedge \mathbf{dtype}(self) \in \mathbf{cn} \\
 & \wedge & head(\overline{self}) \in Id(\Sigma(\mathbf{dtype}(self))) \\
\mathbf{type}(self) & \stackrel{def}{=} & \mathbf{type}(head(\overline{self})) \\
\mathbf{value}(self) & \stackrel{def}{=} & head(\overline{self})
\end{array}
$$

- An attribute $le.a$ is defined only when le is of type of a class and attached to an non-null object, and a is an attribute name. An attribute is thus defined inductively as follows: of that class:

$$
\begin{array}{lcl}
\mathcal{D}(x.a) & \stackrel{def}{=} & \mathcal{D}(x) \wedge \mathbf{dtype}(x) \in \mathbf{cn} \wedge head(\overline{x}) \neq null \\
 & \wedge & \mathbf{type}(x).a \in \mathbf{visattr} \\
\mathbf{type}(x.a) & \stackrel{def}{=} & \mathbf{type}(head(\overline{x}).a) \\
\mathbf{value}(x.a) & \stackrel{def}{=} & head(\overline{x}).a \\
\mathbf{D}(le.b.a) & \stackrel{def}{=} & \mathbf{D}(le.b) \wedge \mathbf{type}(le.b).a \in \mathbf{visattr} \\
\mathbf{value}(le.b.a) & \stackrel{def}{=} & \mathbf{vaule}(le.b).a \\
\mathbf{type}(le.b.a) & \stackrel{def}{=} & \mathbf{type}(\mathbf{value}(le.b).a)
\end{array}
$$

The semantics of the equality $e_1 = e_2$ is the *reference equality*:

$$
\mathcal{D}(e_1) \wedge \mathcal{D}(e_2) \wedge (\mathbf{value}(e_1) = \mathbf{value}(e_2)) \wedge (\mathbf{type}(e_1) = \mathbf{type}(e_2))
$$

Semantics of commands A typical aspect of an execution of an OO program is about how objects are to be attached to program variables (or entities [Mey89]). An attachment is made by an assignment, the creation of an object or passing a parameter in a method invocation.

When we define the semantics $[\![\mathcal{E}]\!]$ of an element $\mathcal{E}$ of the language, we will use $\mathcal{E}$ itself to denote its semantics in a semantic defining equation.

Assignments: There are two cases of assignments. The first is to (re-)attach a value to a variable. This can be done only when the type of the object is consistent with the declared type of the variable. The attachment of values to other variables are not changed.

$$\begin{array}{l} x := e \overset{def}{=} \\ \{x\} : \mathcal{D}(x) \wedge \mathcal{D}(e) \wedge (\mathbf{type}(e) \preceq \mathbf{dtype}(x)) \vdash (\overline{x}' =< \mathbf{value}(e) > \cdot tail(\overline{x})) \end{array}$$

The second case is to modify the value of an attribute of an object attached to a variable. This is done by finding the attached object in the system state Σ and modify its state accordingly. Thus, all variables that points to the identity of this object will be changed.

$$\begin{array}{l} le.a := e \overset{def}{=} \{\Sigma(\mathbf{type}(le))\} : (\mathcal{D}(le.a) \wedge \mathcal{D}(e) \wedge (\mathbf{type}(e) \preceq \mathbf{dtype}(le.a))) \vdash \\ (\Sigma(\mathbf{type}(le))' = \Sigma(\mathbf{type}(le)) \uplus \{\mathbf{value}(le) \oplus \{a \mapsto \mathbf{value}(e)\}) \end{array}$$

Object creation: The execution of $C.New(x)$ (re-)declares variable x, creates a new object, attaches the object to x and attaches the initial values of the attributes to the attributes of x too.

$$\begin{array}{l} C.New(x) \overset{def}{=} \{\mathbf{var}, x, \Sigma(C)\} : C \in \mathbf{cn} \vdash \exists id \notin Id(\Sigma) \bullet \\ \left(\begin{array}{l} \Sigma(C)' = \Sigma(C) \cup \{< id, C, \{a \mapsto \mathbf{Init}(C.a) > \mid a \in \mathbf{attr}(C)\}\} \wedge \\ (x \in \mathbf{var} \wedge (\overline{x}' =< id > \cdot \overline{x})) \wedge \\ (\mathbf{var}' = \{x\} \trianglelefteq \mathbf{var} \cup \{(x, < C > \cdot \mathbf{var}(x))\}) \vee \\ (x \notin \mathbf{var} \wedge (\overline{x}' =< id >) \wedge (\mathbf{var}' = \mathbf{var} \cup \{(x, < C >)\})) \end{array} \right) \end{array}$$

We use $C.New(x)[\underline{c}]$ to denote the command $C.New(x); x.\underline{a} := \underline{c}$, where $\underline{a}$ is the lists of attributes of C, and c is a list of expressions of the same length.

Variable declaration: declares a variable and initializes it:

$$\begin{array}{l} \mathbf{var}\, T\, x = e \overset{def}{=} \{x, \mathbf{var}\} : \mathcal{D}(e) \wedge (\mathbf{type}(e) \preceq T) \vdash \\ \left(\begin{array}{l} (x \in \mathbf{var} \wedge \overline{x}' =< \mathbf{value}(e) > \cdot \overline{x} \wedge \\ \mathbf{var}' = \{x\} \trianglelefteq \mathbf{var} \cup \{< T > \cdot \mathbf{var}(x)\} \vee \\ x \notin \mathbf{var} \wedge (\overline{x}' =< \mathbf{value}(e) >) \wedge (\mathbf{var}' = \mathbf{var} \cup \{(x, < T >)\}) \end{array} \right) \end{array}$$

Variable undeclaration: terminates the block of the permitted use of a variable:

$$\begin{array}{l} \mathbf{end}\, x \overset{def}{=} \{\mathbf{var}, x\} : (\mathrm{x} \in \mathbf{var}) \vdash \\ \left(\begin{array}{l} (|\mathbf{var}(x)| = 1 \wedge \mathbf{var}' = \{x\} \trianglelefteq \mathbf{var}) \vee \\ (|\mathbf{var}(x)| > 1 \wedge \overline{x}' = tail(\overline{x}) \wedge \mathbf{var}' = \{x\} \trianglelefteq \cup \{(x, tail(\mathbf{var}(x))))\} \end{array} \right) \end{array}$$

Method Call: Let v, r and vr be lists of expressions. The command $le.m(v, r, vr)$ assigns the values of the actual parameters v and vr to the formal value and value-result parameters of the method m of the object o that le refers to, and then executes the body of m. After it terminates, the value of the result and value-result parameters of m are passed back to the actual parameters r and vs.

$$\begin{array}{l} le.m(v, r, vr) \stackrel{def}{=} \mathbf{D}(le) \wedge \mathbf{type}(le) \in \mathbf{cn} \wedge m \in \mathbf{op}(\mathbf{type}(le)) \Rightarrow \\ \exists N \bullet (\mathbf{type}(le) = N) \wedge \left(\begin{array}{l} \mathbf{var}\ N\ self = le, T_1\ x = v, T_2\ y = r, T_3\ z = vr; \\ N.m;\ r, vr := y, z; \mathbf{end}\ self, x, y, z \end{array} \right) \end{array}$$

where x, y, z are the value, result and value-result parameters of the method of class $\mathbf{type}(le)$, and $N.m$ stands for the design associated with method m of class N.

All other programming constructs will be defined in exactly the same ways as their counter-parts in a procedural language. For example, *the sequential composition* corresponds to relational composition:

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

4.2.4 Semantics of a class declaration

A class declaration $cdecl$ given in Section 4.1.1 is well-defined if the following conditions hold.

1. N has not been declared before, N and M are distinct, and the attribute names of N are distinct.
2. The initial values of the attributes matches their corresponding types.
3. The method names are distinct.
4. The parameters of every method are distinct.

Let $\mathcal{D}(cdecl)$ denote the conjunction of the above conditions . The class declaration $cdecl$ adds the structural information of class N to the state of the following up program, and this role is characterized by the following design.

$$\begin{array}{l} cdecl \stackrel{def}{=} \{\mathbf{cn}, \mathbf{super}, \mathbf{pri}, \mathbf{protattr}, \mathbf{pub}\} : \mathcal{D}(cdecl) \vdash \\ \left(\begin{array}{l} \mathbf{cn}' = \mathbf{cn} \cup \{N\} \wedge \mathbf{super}' = \mathbf{super} \oplus \{N \mapsto M\} \wedge \\ \mathbf{pri}' = \mathbf{pri} \oplus \{N \mapsto \{< \underline{u} : \underline{U}, \underline{a} >\}\} \wedge \\ \mathbf{pro}' = \mathbf{pro} \oplus \{N \mapsto \{< \underline{v} : \underline{V}, \underline{b} >\}\} \wedge \\ \mathbf{pub}' = \mathbf{pub} \oplus \{N \mapsto \{< \underline{w} : \underline{W}, \underline{c} >\}\} \wedge \\ \mathbf{op}' = \mathbf{op} \oplus \{N \mapsto \{(m_1 \mapsto (paras_1, c_1)), \ldots, (m_\ell \mapsto (paras_\ell, c_\ell))\}\} \end{array} \right) \end{array}$$

where the dynamic behaviour of the methods cannot be defined before the dependency relation among classes is specified. At the moment, the logical variable $\mathbf{op}(N)$ binds each method m_i to code c_i rather than its definition which will be calculated in the end of the declaration section.

4.2.5 The well-definedness of a declaration section and semantics of a program

A class declaration section $cdelcs$ comprises a sequence of class declarations. Its semantics is defined from the semantics of a single class declaration given in the previous subsection, and the semantics of sequential composition. However, the following well-definedness conditions need to be enforced:

D_1**:** All class names used in the variable, attribute and parameter declarations are defined in the section.

D_2**:** The function **super** does not induce circularity.

D_3**:** No attributes of a class can be redefined in its subclasses.

D_4**:** No method of a class is allowed to redefine its signature in its subclass.

Let $cdecls$ be a class declration section and P a command, the meaning of a program $(cdecls \bullet P)$ is defined as the composition of the meaning of class declarations $cdecls$ (defined in Section 4.2.4), the design $init$, and the meaning of command P:

$$cdecls \bullet P \stackrel{def}{=} (cdecls; init; P)$$

where the design $init$ performs the following tasks

1. to check the well-definedness of the declaration section,
2. to decide the values of $\mathbf{attr}$ and $\mathbf{visattr}$ from those of $\mathbf{pri}$, $\mathbf{pro}$ and $\mathbf{pub}$.
3. to define the meaning of every method body c.

The design $init$ is formalized as:

$$init \stackrel{def}{=} \{\mathbf{visattr}, \mathbf{attr}, \mathbf{op}\} : \mathcal{D}_1 \wedge \mathcal{D}_2 \wedge \mathcal{D}_3 \wedge \mathcal{D}_4 \vdash \left(\begin{array}{l} \mathbf{visattr}' = \bigcup_{N \in \mathbf{cn}} \{N.a \mid \exists T, c \bullet < a : T, c > \in \mathbf{pub}(N)\} \wedge \\ \forall N \in \mathbf{cn} \bullet \mathbf{attr}'(N) = \cup \{\mathbf{pri}(M) \cup \mathbf{pro}(M) \cup \mathbf{pub}(M) \mid N \preceq M\} \wedge \\ \mathbf{op}'(N) = \{m \mapsto (paras, \Psi(N.m)) \mid m \in \mathbf{op}(M) \wedge N \preceq M\} \end{array}\right)$$

where the family of designs $\Psi(N.m)$ defined by a set of recursive equations. It contains for each class $N \in \mathbf{cn}$, each class such that $N \preceq M$, and every method $m \in \mathbf{op}(M)$ and equation

$$\Psi(N.m) = F_{N.m}(\Psi) \quad \text{where } \mathbf{supercalss}(N) = M$$

where F is constructed according to the following rules:

Case (1) m is not defined in N, but in a superclass of N, i.e. $m \notin \mathbf{op}(N) \wedge m \in \cup\{\mathbf{op}(M) \mid N \preceq M\}$. The defining equation for this case is simply

$$F_{N.m}(\Psi) \stackrel{def}{=} \Psi(M.m)$$

Case (2) m is a method defined in class N. In this case, the behaviour of the method $N.m$ is captured by its body and the environment in which it is executed

$$F_{N.m}(\Psi) \stackrel{def}{=} Set(N); \phi_N(\mathbf{body}(N.m)); Reset$$

where the design $Set(N)$ finds out all attributes visible to class N, whereas $Reset$ does it for the main program:

$$Set(N) \stackrel{def}{=} \{\mathbf{visattr}\} : true \vdash \mathbf{visattr}' = \left(\begin{array}{l} \{N.a \mid a \in \mathbf{pri}(N)\} \cup \{M.a \mid N \preceq M, a \in \mathbf{pro}(M)\} \cup \\ \{M.a \mid M \in \mathbf{cn}, a \in \mathbf{pub}(M)\} \end{array} \right)$$

$$Reset \stackrel{def}{=} \{\mathbf{visattr}\} : true \vdash \mathbf{visattr}' = \{M.a \mid M \in \mathbf{cn}, a \in \mathbf{pub}(M)\}$$

The function ϕ_N renames the attributes and methods of class N in the code $body(N.m)$ by adding object reference $self$:

$$\begin{array}{l}
\phi_N(skip) \stackrel{def}{=} skip, \quad \phi_N(chaos) \stackrel{def}{=} chaos \\
\phi_N(p_1; p_2) \stackrel{def}{=} \phi_N(p_1); Set(N); \phi(p_2) \\
\phi_N(P_1 \triangleleft b \triangleright P_2) \stackrel{def}{=} \phi_N(P_1) \triangleleft \phi_N(b) \triangleright \phi_N(P_2) \\
\phi_N(P_1 \sqcap P_2) \stackrel{def}{=} \phi_N(P_1) \sqcap \phi_N(P_2) \\
\phi_N(b * P) \stackrel{def}{=} \phi_N(b) * (\phi_N(P); set(N)) \\
\phi_N(\mathbf{var}\ x : T = e) \stackrel{def}{=} \mathbf{var}\ x : T = \phi_N(e), \quad \phi_N(\mathbf{end}\ x) \stackrel{def}{=} \mathbf{end}\ x \\
\phi_N(C.New(x)) \stackrel{def}{=} C.New(\phi_N(x)), \quad \phi_N(le := e) \stackrel{def}{=} \phi_N(le) := \phi_N(e) \\
\phi_N(le.m(v, r, vr)) \stackrel{def}{=} \phi_N(le).m(\phi_N(v), \phi_N(r), \phi_N(vr)) \\
\phi_N(m(v, r, vr)) \stackrel{def}{=} self.m(\phi_N(v), \phi_N(r), \phi_N(vr)) \\
\phi_N(x) \stackrel{def}{=} \begin{cases} self.x & \exists M \cdot N \preceq M \wedge x \in \mathbf{attr}(M) \\ x & otherwise \end{cases} \\
\phi_N(self) \stackrel{def}{=} self, \quad \phi_N(le.a) \stackrel{def}{=} \phi_N(le).a \\
\phi_N(null) \stackrel{def}{=} null, \quad \phi_N(f(e)) \stackrel{def}{=} f(\phi_N(e))
\end{array}$$

5 Specification of UML Models

This section uses an example of a simple library system to show how to specify and reason about UML models of requirements analysis and designs.

Conceptual class diagram: A class in a class diagram is specified as class declaration. An association between two classes N and M is a type of pairs of objects of N and M, and modelled as a class that has two attributes with the association's *end roles* N and M as their types.

Assume that an iteration of a library system development considers the use case to record a copy and the conceptual model in left diagram of Figure 1. A library $Lib\ Owns$ a number of $Publication$s and each publication $Contains$ some $Copy$(ies).

We specify this diagram as CM_1 below:

$$\begin{array}{l}\mathbf{Class}\ Lib\ \{String\ name, String\ address\}; \mathbf{Class}\ Copy\ \{String\ id\};\\ \mathbf{Class}\ Pub\ \{String\ id, String\ title, String\ author, String\ isbn;\};\\ \mathbf{Class}\ Contains\ \{Pub\ p, Copy\ c;\}; \mathbf{Class}\ Owns\ \{Lib\ lib, Pub\ p\}\end{array}$$

We also assume the set of objects $\Sigma(N)$ is initially empty for each class N, and the operations $S.find()$, $S.add()$, and $S.delete()$ for a set S.

Use cases: The informal identification and description of a use case is important for the creation of the conceptual model to support it. However, the formal specification of the use cases depends on the specification of the conceptual model. We have a *canonical form* of a use case specification by introducing a *use-case handler* class[1]. At any time during the execution, this class will only have a single instance. Considering the use case $RecordCopy$ that adds a new copy of an *existing* publication to the library. We specify this use case by introducing a use case handler class $HandleRcopy$:

$$\begin{array}{l}CM_1;\ \text{/** import the conceptual model}\\ \mathbf{Class}\ HandleRcopy ::\\ \quad \mathbf{method}\ RecordCopy(< String\ cid, String\ pid >)\{\\ \quad \exists p \in \Sigma(Pub), \ell \in \Sigma(Lib) \bullet p.id = pid \wedge (\ell, p) \in \Sigma(Owns)\ \vdash\\ \quad Copy.New(c)[cid]; \mathbf{var}\ Pub\ p = \Sigma(Pub).find(pid);\\ \quad \Sigma(Copy) := \Sigma(Copy) \cup \{c\} \wedge \Sigma(Contains) := \Sigma(Contains) \cup \{< p, c >\};\\ \quad \mathbf{end}\ c, p\}\end{array}$$

We have used in the specification programming commands, programming constructs, predicate and logical connectives. This is because that programming commands and constructs have been defined as predicates and logical operations. Also $c_1 \wedge c_2$ does not specify the order in which of the commands c_1 and c_2 are executed.

Then we define the system specification by defining the $\mathbf{main}$ method in which

$$RCopy \stackrel{def}{=} read(String\ cid, String\ pid); hrc.RecordCopy(cid, pid)$$

[1]This is suggested by the *facade controller pattern*.

in the following statement:

$$
\begin{array}{l}
\mathbf{main}()\{\mathbf{var}\ Bool\ stop = false, Services\ s;\\
HandleRCopy.New(hrc); Lib.New(lib);\\
\neg stop * (read(s); \mathbf{if}\ \{s = "RecordCopy" \longrightarrow RCopy\}\ \mathbf{fi}; read(stop));\\
\mathbf{end}\ stop, s\}
\end{array}
$$

where $Services$ denotes the set of *names* of services that the library system provides. When further use cases are developed, their names are added to $Services$ and their executions are added to the command set in the multiple choice statement of the main method.

Within our model, we can check that the conceptual class diagram is consistent with the use case specified specification by calculation the semantics of $(CM_1; HandleRcopyDecl) \bullet \mathbf{main}()$, that checks the well-definedness of the declaration section $(CM_1; HandleRcopyDecl)$, the well-definedness of the commands in $\mathbf{main}()$. The well-definedness of $(CM_1; HandleRcopyDecl)$ implies the well-formedness (i,e. *static consistency*) of the corresponding UML class diagram, and the corresponding class diagram and the use case model are *consistent* if semantics of the composition $(CM_1; HandleRcopyDecl) \bullet \mathbf{main}()$ does not equal to $chaos$. A correction of any inconsistency is then formally treated as a *refinement* of the program specification.

Interaction diagrams and design class diagrams: To specify a sequence diagram and a design class diagram, we need to refine the classes in the conceptual model into *software classes* by adding methods, and realizing the associations by attributes of classes. Roughly speaking, an incoming message to a class in a sequence diagram corresponds to a method of the class, the outgoing messages of the class following that incoming message is specified as sequential composition of method calls. For example, the object sequence diagram and its design class diagram in Figure 2 (on the left) are specified by the following class declarations:

$$
\begin{array}{l}
\mathbf{Class}\ Lib\ \{String\ name; String\ address; \mathbb{P}Pub\ pub;\ \text{/** newly added}\\
\quad \mathbf{method}\ add(< String\ cid, String\ pid >)\{\\
\quad\quad \mathbf{var}\ Pub\ p = Pub.find(pid); p.makeCopy(cid); \mathbf{end}\ p\}\};\\
\mathbf{Class}\ Pub\ \{String\ id; String\ title; String\ author; String\ isbn; \mathbb{P}Copy\ Cp;\\
\quad \mathbf{method}\ makeCopy(< String\ cid >)\ \{Copy.New(c)[cid]; Cp.add(c)\}\};\\
\mathbf{Class}\ Copy\ \{String\ id\};\\
\mathbf{Class}\ HandleRcopy\ \{Lib\ lib;\\
\quad \mathbf{method}\ RecordCopy(< String\ cid, String\ pid >)\{lib.add(cid, pid)\}\}
\end{array}
$$

In the design, $\mathbf{main}()$ method is *almost* the same as that in the use-case model.

$$
\begin{array}{l}
\mathbf{main}()\{\mathbf{var}\ Bool\ stop = false, Services\ s;\\
Lib.New(lib); HandleRCopy.New(hrc)[lib];\\
\neg stop * (read(s); \mathbf{if}\ \{s = "RecordCopy" \longrightarrow RCopy\}\ \mathbf{fi}; read(stop));\\
\mathbf{end}\ stop, s\}
\end{array}
$$

With our model and refinement calculus, we can prove that this design refines *the program that represents the use-case specification.*

5.1 Further development of the library system

Now consider the use case to register a member that creates a member and logs it to the library. We thus have to *extend* the class diagram on the left of Figure 1 to the one on the right, that is denoted by CM_2, by adding the following two classes.

$$\textbf{Class } Member \ \{String\ name; String\ title; String\ id; String\ address; \};$$
$$\textbf{Class } Has \ \{Lib\ lib; Member\ m\}$$

Let $SList$ be the type $String \times String \times String \times String$ and $details \stackrel{def}{=} (name, title, id, address)$ denote the tuple of the attributes of $Member$. We then specify the use case to register a member, denoted by $RegisterM$, by a use-case handler $HandleRM$ which has $Lib\ lib$ as an attribute amd a method $RegisterM$:

$$\begin{array}{l} HandleRM :: RegisterM(< SList\ ml >)\{ \\ \neg\exists m \in \Sigma(Member) \bullet m.details = ml \wedge \exists lib \in \Sigma(Lib) \vdash \\ Member.New(m)[ml]; \Sigma(Member) := \Sigma(Member) \cup \{m\}; \\ (\ell \in \Sigma(Lib) \Rightarrow \Sigma(Has) := \Sigma(Has) \cup \{< \ell, m >\})\} \end{array}$$

We can prove that this use case is consistent with the extended conceptual model.

We can consider $RegisterM$ independently from $RecordCopy$ with its own conceptual model of classes Lib, $Member$ and the association Has. We then obtain CM_2 by merging this model with CM_1. If different names are used for the same concept, rename one to another.

Then the **main** method will be enlarged by adding the service name "$RegisterM$" in $Service$ and adding the following command to the multiple choice statement

$$RegM \stackrel{def}{=} read(SList\ ml); hrm.RegisterM(ml)$$

guarded by $s =$ "$RegisterM$"

$$\begin{array}{l} \textbf{main}()\{\textbf{var}\ Bool\ stop = false, Services\ s; \\ Lib.New(lib); HandleRCopy.New(hrc)[lib]; HandleRM.New(hrm)[lib]; \\ \neg stop * (read(s); \textbf{if} \left\{ \begin{array}{l} s = \text{"}RecordCopy\text{"} \longrightarrow RCopy, \\ s = \text{"}RisterM\text{"} \longrightarrow RegM \end{array} \right\} \textbf{fi}; read(stop)); \\ \textbf{end}\ stop, s\} \end{array}$$

Following the design patterns in [Lar01], we can work out the interaction diagram and the design class diagram on the right Figure 2.

$$\begin{array}{l} Lib :: \mathbb{P}Member\ M;\ /**\text{add a new attribute to } Lib \\ Lib :: makeMember(<\ SList\ ml >)\{/**\text{add a method to } Lib \\ \quad New\ Member(m)[ml]; M.add(m);\ \textbf{end}\ m\}; \\ HandleRM ::\ RegisterM(< SList\ ml >)\{lib.makeMember(SList\ ml)\} \end{array}$$

The **main** method is nearly the same as in the requirement specification. With CM_2, we can specify and design use cases $SearchMember$, $SearchPub$ and $SearchCopy$. Further cycles can be carried in the same way for more use cases, such as $BorrowCopy$ and $ReurnCopy$.

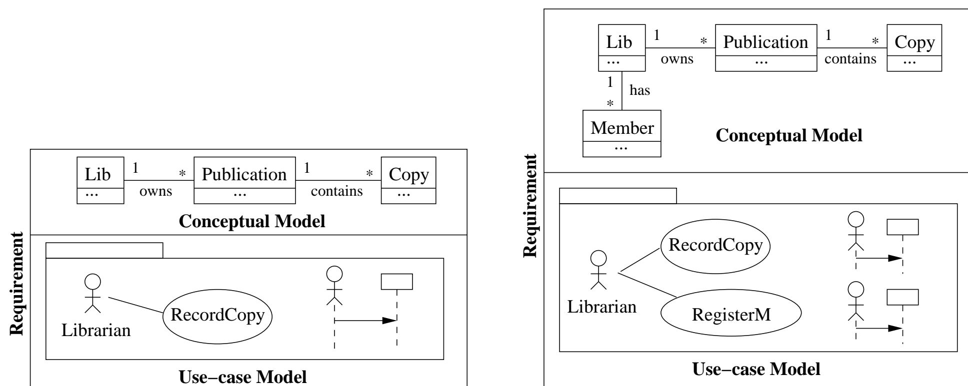


Figure 1: Models of the Requirements for Cycle 1 (Left) and Cycle 2 (Right)

6 Conclusion

Based on UTP [HH98], we have presented a model for OO programs. The model is compositional, where the well-definedness of a class is determined in dependence of its constituents. Incremental code changes, as what often happen in OO programming, require revising only the affected parts of the model and not the model as a whole. The important nature of the integrated method is that each iteration is only concerned with a small part of the system functionality and a small model at a time. Instead of using a traditional compositional approach, we decompose the system informally according to use cases. We obtain formal models in each iteration and compose them to form a larger system step by step. We believe that this is important for scaling up the use of a formal method. A system developed this way is easy to maintain when the business rules change.

6.1 Related Work

Models of object-oriented programs: There is a large number of publications on models for OO programming, e.g. [AC96, AL97, BKS98, Ame, Car89, Nau94]. It is difficult to give a fair account of the relation of this paper with them. However, a large body of work on modelling OO programming is based on type theories or operational semantics. Our approach is among those that are state-based and use a simply predicate logic.

Sate-based formalisms have been used in conjunction with OO techniques, via languages such as Object-Z [Car89] and VDM^{++} [DD], and methods such as Syntropy [CD94] (which uses the Z notation) and Fusion [Col94] (which is related to VDM). Whilst these formalisms are effective in modelling data structures as sets and relations, they are not ideal for capturing more sophisticated OO mechanisms, such as dynamic binding and polymorphism.

Naumann defines an OO programming language with subtype and polymorphism using predicate transformer [Nau94]. Mikhajlova and Sekerinski [MS97] design a rich OO language by using a type system and predicate transformers as well. However, neither reference types nor mutual dependency between classes is allowed in

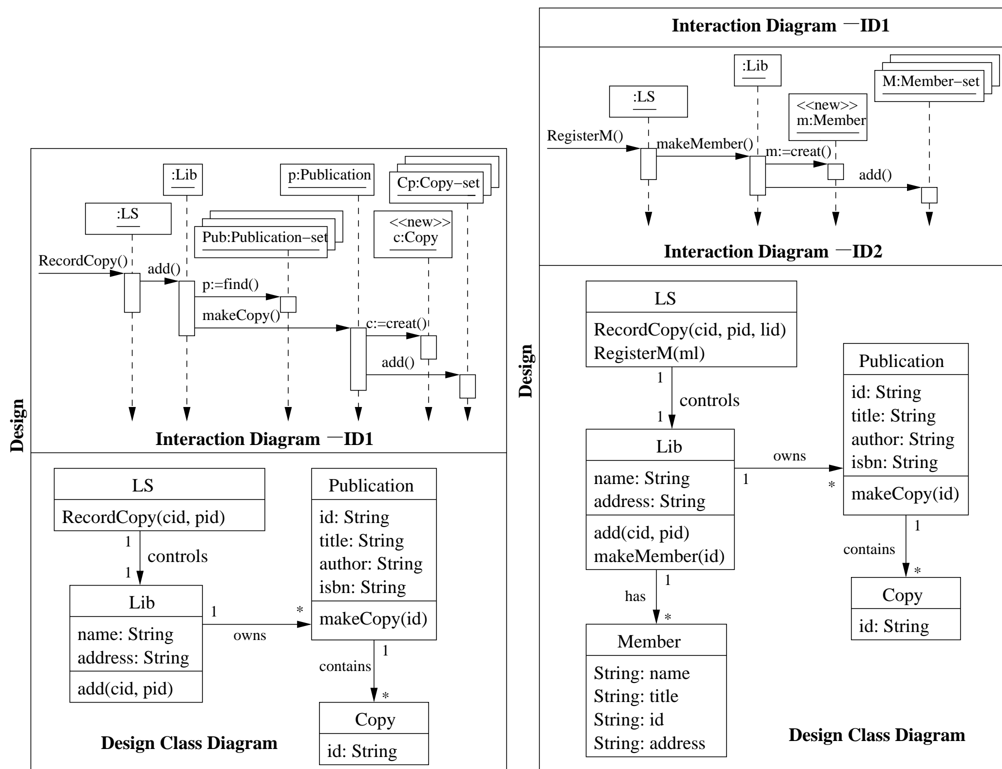


Figure 2: Models of the Designs for Cycle 1 (Left) and Cycle 2 (Right)

those approaches. Leino [Lei98], has extended an existing calculi with OO features, but restricting inheritance and not dealing with classes and visibility.

In our model, a program is represented as a predicate called a *design* in UTP [HH98]. So the refinement relation between programs is defined as implication between their designs. Another advantage of our approach is that writing a specification in the relational calculus is quite straightforward and a specification is easy to understand. Although we have not dealt with concurrency, the power of UTP for describing different features of computing, including concurrency and communication, timing, and higher-order computing [HH98, Woo02, SH02], makes our approach ready for extension to cope with these different aspects of OO programs.

Formalizations of UML: The research of formal support for UML modelling is currently very active (e.g. [Eva, BPP99, Eng01, Egy01, HR00, Reg01]). However, there is a large body of work in formalizing UML and providing tool support for UML focuses on models for a particular view (e.g. a class models, statecharts, and sequence diagrams), and the translation of them into an existing formal formalism (e.g. Z, VDM, B, and CSP). In contrast, we concentrate on use cases and combinations of different UML models. This is the most *imprecise* part of UML and the majority of existing literature on the UML formalization often avoids them. Our methodology is directed towards improved support for requirement analysis and transition from requirements to design models in RUP. Our 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.

In this paper, we focus on only conceptual aspects of object orientation. Most syntactical and semantic con-

sistency conditions defined in this paper have straightforward algorithms for checking and hence support necessary automated tools. For example, the transformation of a class diagram to a declaration section is obvious and the well-defined conditions for declaration section is clearly consistent with the well-formed conditions of UML defined in terms of OCL. Other constraints on a class model, such *multiplicities* of an association, properties of an *aggregation association*, characterization of an *abstract class* and *associative classes*,can be specified as state invariants that need to be preserved by use case commands [LHLC03].

6.2 Future work

Future work includes the extension of this method to component-based software development (e.g. [DW98, Szy98]) so that components can be developed in parallel, and the application of this framework to formal treatment of *patterns* [Gam95]. In addition, tool support, e.g. in the direction of [Har02], for formal OO methods is an area of considerable significance for further industrial take-up of these methods.

References

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

[AL97] M. Abadi and R. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference*, pages 682–696. Springer-Verlag, 1997.

[Ame] P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, illem P. de Roever, and G. Rozenberg, editors, *REX Workshop*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. 1991.

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

[BPP99] R.J.R. Back, L. Petre, and I.P. Paltor. Formalizing UML use cases in the refinement calculus. In *Proc. UML'99*. Springer-Verlag, 1999.

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

[Car89] D. Carrington, *et al*. *Object-Z: an Object-Oriented Extension to Z.* North-Halland, 1989.

[CD94] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall, 1994.

[Col94] D. Coleman, *et al*. *Object-Oriented Development: the FUSION Method.* Prentice-Hall, 1994.

[DD] E. Dürr and E.M. Dusink. The role of VDM^{++} in the development of a real-time tracking and tracing system. In *Proc. of FME'93*, volume 670 of *LNCS*.

[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. 16th IEEE ASE*, San Diego, USA, 2001.

[Eng01] G. Engels, *et al*. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *The Proc. FSE-10*, Austria, 2001.

[Eva] A. Evans, *et al*. Developing the UML as a formal modelling notation. In *Proc. UML'98, LNCS 1618*. Springer-Verlag.

[Gam95] E. Gamma, *et al*. *Design Patterns*. Addison-Wesley, 1995.

[Har02] D. Harel, *et al*. Smart play-out of behavioral requirements. In *Proc. FMCAD02*, pages 378–398, 2002.

[HH98] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[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.

[Kru00] P. Kruchten. *The Rational Unified Process – An Introduction (2nd Edition)*. Addison-Wesly, 2000.

[Lar01] C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.

[Lei98] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programming. In *LNCS 1381*. Springer, 1998.

[LHLC03] Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. To appear in Proc. of ICFEM03, Singapore, 2003.

[Mey89] B. Meyer. From structured programming to object-oriented design: the road to Eiffel. *Structured Programming*, 10(1):19–39, 1989.

[MS97] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-orient programs. In *Proc of FME'97, LNCS*. Springer, 1997.

[Nau94] Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olerog, editor, *Proc. of PROCOMET'94*. North-Holland, 1994.

[Reg01] G. Reggio, *et al*. Towards a rigorous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *Proc. FASE 2001, LNCS 2029*. Springer, 2001.

[SH02] A. Sherif and J. He. Towards a time model for Circus. In *ICFEM02, LNCS 2495*. Springer, 2002.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Woo02] J.C.P. Woodcock. Unifying theories of parallel programming. In *Logic and Algebra for Software Engineering*. IOS Press, 2002.