Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 101 (2004) 95–127

www.elsevier.com/locate/entcs

Unifying Views of UML [1]

Zhiming Liu, He Jifeng and Jing Liu

International Institute for Software Technology
The United Nations University, Macao SAR China

Xiaoshan Li

Faculty of Software Engineering, University of Macao, Macao SAR China

Abstract

We present an approach to embedding a formal method into Rational Unified Process (RUP). The purposes are: (a) to unify different views of UML, (b) to enhance 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 by providing a unified conceptual framework. One of the main features of RUP is that it is use-case driven and supports iterative development more explicit than other process models, such as the waterfall model. Object-orientation also supports better and more explicitly incremental programming than the traditional imperative programming. These together will help 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

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 represented in a program-

[1] Email: `lzm@iist.unu.edu,hjf@iist.unu.edu,xsl@umac.mo`.
Z. Liu is on leave from Department of Computer Science of the University of Leicester, England;
J. He is on leave from East China Normal University, Shanghai, China.
J. Liu is on leave from Shanghai University, China.
This work is partly supported by the research grant 02104 MoE and the 973 project 2002CB312000 of MoST of P.R.China.

doi:10.1016/j.entcs.2004.02.018

ming language. As the real world keeps changing, the software system that represents 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) [32] has emerged as a popular software development process [33,38,31]. As the modelling notation, RUP uses UML [9], which 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 standing above the others is the idea of *use-case driven and iterative development*. In the use-case driven and iterative approach of RUP, a system development is organized as a series of short, fixed-length mini-projects called *iterations*, each for a small number of use cases. 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 object-oriented 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. The discussion on the relationship to existing work is given in Section 7.

2 RUP and UML

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

2.1 Requirement analysis

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

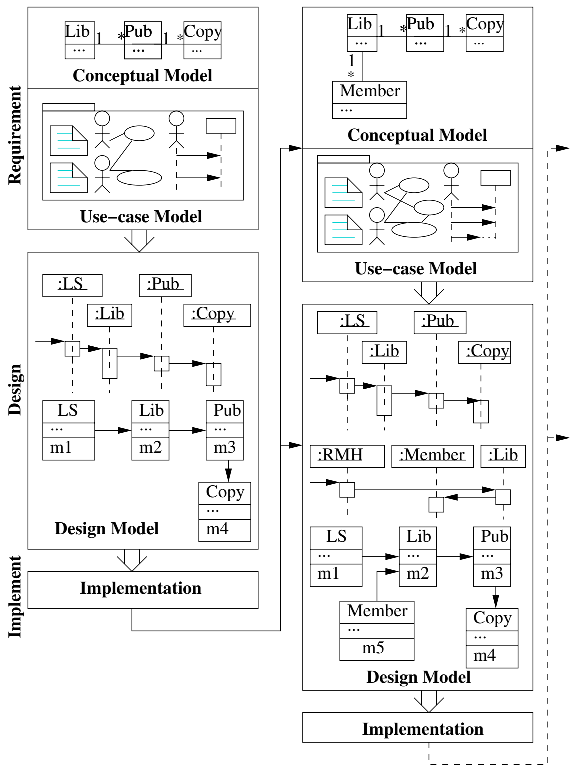


Fig. 1. Iterative software development Process

The use-case model consists of an use-case diagram and a textual description of each use case. The use-case diagram provides only static information about the use cases. The functional and behavioral properties described in a textual descriptions of the use cases are *partially* represented as sequences of interactions between *actors* of the use case and the system. A UML *system sequence diagram* is used to describe the possible 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. It is important to note that a system sequence diagram does not and should not provide information about interactions among objects inside the system [14,38,40], as the these internal interactions are the concern of the design. A formalization of an use-case model should deal with both the order of the interactions (i.e. the *interactive view*) and the state changes caused by these interactions *i.e. the functional and behavioral view.* The Object Constraint Language (OCL) [53], as a part of UML, can describe some functional aspects, such as pre- and post-conditions of operations. However, OCL does not have a formal semantics and it is not expressive enough to describe many useful aspects of object-orientation, such as recursive method call, dynamic binding. In this paper, we will define an object-oriented specification language (OOL) that can be used to specify functional behaviour at different levels of abstract.

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 and interactive 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, which 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 object-oriented 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. It is possible to develop a tool to help in transforming a design into a code of implementation.

2.4 Iterative and incremental model construction

The iterative lifecycle in RUP is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaption as core drivers to converge upon a suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as *iterative and incremental*(see Figure 1). Early iterative process ideas were known as spiral development and evolutionary development [7]. However, what is new in RUP is its nature of being use-case driven and its explicit support to object-orientation and incremental development.

Each cycle may consider one or a number of use cases to analyze, and build a conceptual model for them. Then these use cases can be designed to create the interaction diagrams and the design class diagram. The design is then verified, coded and tested. This one cycle can also be done again in an iterative way by taking the use cases or “parts” of a use case in turns. In the next cycle, we may take a refined or extended version of a use case from the previous cycle and refine or extend the system, or we take some new uses case to analyze, design and implement. Therefore, the models created in the new iteration are refinement and enlargement of those obtained in the previous iteration. This is shown in Figure 1.

In each iteration, 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* among these objects. 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. This consistency needs to be checked in each cycle. The design specification is required to be verified against the requirement specification of the same iteration. And the consistency and correctness need to be preserved in the following cycles. The main aim of the integration of the formal method into RUP is to allow these checks to be carried out precisely.

3 Summary of the Approach

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

(i) How to formalize a UML model of the requirements in an iteration and ensure the consistency between static and dynamic views of the model?

(ii) How to formalize a UML model of the design of an iteration and ensure the consistency between its static and dynamic/interaction views?

(iii) How to formally relate the UML models of the design and the requirements of an iteration?

(iv) How to preserve the established consistency and correctness in sequent iterations?

We approach these problems by first developing a formal framework for object-oriented programming. The framework includes an object-oriented specification language (OOL) and a calculus for refinement of object-oriented designs (COOL). The calculus is relational and predicative and it is based on Hoare and He's Unifying Theories of Programming (UTP) [29]. 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 an object-oriented program specification $cdecls_{cm} \bullet c_{um}$. The semantics of $cdecls_{cm}$, c_{um} and their composition $\bullet$ are given in the semantics of OOL. This formalization captures both the syntax and semantics of cm and um and the consistency between them. The command c_{um} specifies what operations on the system states are to be carried out and in what possible order. A system state is in fact an *object diagram* of the class diagram cm. As one cannot decide the order in which the use cases are to be performed, OOL will allow the specification of non-determinism. Therefore, OOL will not only formalize some UML models, but also complement UML to provide functional descriptions of use cases. Properties of the specification, such as class invariants and constraints on associations between classes can be reasoned in the logic. Consistency changes in the class and use case models can be ensured by preserving these properties and even by refinement.

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 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_{sd}. Therefore, a UML model of design (dc, sd) is also specified as the composition of a declaration section and a main program (which again does not have a UML counterpart): $cdecls_{dc} \bullet c_{sd}$. The consistency between the class diagram dc and the object sequence diagrams sd are captured by the semantics of $cdecls_{dc}$ and the semantics of method calls in the OOL. In fact, the specification of the methods in the design class diagram combine both of *functional view* which has not UML counterpart, *behavioural view* which is described by object UML state diagrams and *interactive view* which is captured by UML object interaction diagrams. The *correctness* of the design model (dc, sd) $w.r.t$ the requirements model (cm, um) is defined by the *refinement* relation

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

Refinement can also be used to justify steps of incremental design (e.g. those informally used in [33,38]), such as adding attributes, promoting attributes from a subclass to it superclass, encapsulating attributes, delegating functionality of an object to its associated objects, etc. 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 an object-oriented language with classes, references, visibility, dynamic binding, nested declaration, and mutual recursive method calls. Class declarations as well as commands will be defined as *designs*, the notion that is defined in [29].

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 a Java program.

The main method command corresponds to an active class. In this paper, we only deal with sequential programs thus we do not deal with active class in general. Therefore, in this case, the statechart of an object will start from a state when a method is invoked and then state changes will follow according to the method definition. In a multi-thread computation, we would have more than one “main method” or “run” method, so more than one active classes.

4.1.1 Class declarations

We assume a set $CName$ of *class names*. A *declaration section* $cdecls$ is of the form $cdecls ::= cdecl \mid cdecls; cdecl$, where $cdecl$ is a *class declaration* of the following form

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

where

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

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

  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 u of N, we use $\mathbf{dcltype}(N.u)$ to denote the type of a and $\mathbf{Init}(N.u)$ the initial value of u declared in N.
- The **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 object-oriented programming constructs:

$$\begin{array}{lll} c ::= & skip \mid chaos \mid c; c & \text{termination, abort, sequence}\\ & \mid \textbf{var}\ T\ \text{x=e} \mid \textbf{end}\ x & \text{local variable declaration and undeclaration}\\ & \mid c \lhd b \rhd c \mid c \sqcap c & \text{conditional and nondeterministic choice}\\ & \mid b * c \mid read(T\ x) & \text{iteration and read in a value}\\ & \mid C.new(x) & \text{object creation}\\ & \mid le := e \mid le.m(\underline{e}, \underline{v}, \underline{u}) & \text{assignment and method call}\end{array}$$

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.

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 $Null$. Notice that expressions can appear as arguments of method calls, but we do not allow method call to be an expression as we explicitly use result and val-result parameters in methods. We can include more expression such as type casting $(C)e$ and type test $(e \textbf{ is } C)$, but they are not needed in this paper.

4.2 Semantics

In UTP [29], 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

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

  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.

A design $D_2 \overset{def}{=} (\alpha, P_2)$ is called a *refinement* of $D_1 \overset{def}{=} (\alpha, P_1)$, denoted $D_1 \sqsubseteq D_2$, if P_2 entails P_1, i.e.

$$\forall x, x', \ldots, z, z' \cdot (P_2 \Rightarrow P_1)$$

where we assume that α contains $x, \ldots, z$.

Let ρ be a mapping from α_2 to α_1 that can be defined as a design too, then a design $D_2 \overset{def}{=} (\alpha_2, P_2)$ is a *refinement under the mapping* of $D_1 = (\alpha_1, P_1)$ under mapping ρ if

$$\forall x \in \alpha_2, \forall y' \in \alpha_1' \cdot ((\rho; P_1) \Leftarrow (P_2; \rho))$$

where α_1' is the set of primed versions of those variables in α_1; and the semantics of sequential composition is formally defined later.

We will follow this classical way of defining a state-based model for a programming language and define our OOL in terms of variables, states, expressions, commands, declarations and programs. However, for simplicity, the above model adopts a universal data type and allows neither reference types nor nested declarations. To formalize the behaviour of an object-oriented 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 REF of object identities that can be referred to, and $null \in REF$. An *object* o is an entity defined by the following structure

$$o ::= null | \; < ref, \mathbf{type}, \mathbf{state} >$$

where $ref \in REF$, and **type** is a class name, and **state** is a mapping from **attr**(**type**) to objects. For an object $o =< ref, C, \sigma >$, we use $identity(o)$ to denote the identity ref of o, $\mathbf{type}(o)$ to denote the type name C of the object o, $\mathbf{state}(o)(a)$ to denote the value $\sigma(a)$ of an attribute a of class C. When there is no confusion, we also use C to denote the set of objects o such that $\mathbf{type}(o) = C$, and in this case C also denotes the *semantic class/type* and we can say $o \in C$.

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 $o =< ref, C, \sigma >$ means one in $\mathcal{O}$ if there is no confusion, and will use $ref.a$ to denote $\mathbf{state}(o)(a)$, and $\mathbf{type}(ref)$ 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 $N = Null$ or $N = M$, 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 $i \in \{1, \ldots, k\}$.
- Let S and S_1 be sets. For better reading in the context of semantic definition, we define $S_1 \trianglelefteq S$ to be 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 $o =< ref, C, \sigma >$, an attribute a of C and an entity d which is either a member of a primitive type or an object in $\mathcal{O}$,
$$ref \oplus \{a \mapsto d\} \overset{def}{=} < ref, C, \sigma \oplus \{a \mapsto d\} >$$
- For a set $S \subseteq \mathcal{O}$ of objects,
$$S \uplus \{< ref, C, \sigma >\} \overset{def}{=} \{o \mid identity(o) = ref\} \trianglelefteq S \cup \{< ref, C, \sigma >\}$$
$$Ref(S) \overset{def}{=} \{ref \mid ref \text{ is the identity of an object in } S\}$$

4.2.2 Variables and states

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

(i) **cn**: its value is the set of class names which are declared so far, and it is modified by a class declaration.

(ii) 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.

(iii) 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

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

(iv) **super**: the partial function mapping a class to its *direct* superclass. This variable is also modified by a class declaration.

(v) **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 $i \in \{1, \ldots, n\}$, is the most recently declared type of x_i and denoted by $\mathbf{dcltype}(x_i)$. We will also use $\mathbf{var}(x)$ to denote the sequence of types associated with x.

(vi) **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.

(vii) $\overline{x}$: 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.

$$\begin{array}{lll}
\mathcal{D}(x) & \overset{def}{=} & x \in \mathbf{var} \wedge (\mathbf{dcltype}(x) \text{ is primitive } \vee \mathbf{dcltype}(x) \in \mathbf{cn}) \\
 & \wedge & \mathbf{dcltype}(x) \text{ is primitive} \Rightarrow \mathit{head}(\overline{x}) \in \mathbf{dcltype}(x) \\
 & \wedge & \mathbf{dcltype}(x) \in \mathbf{cn} \Rightarrow \mathit{head}(\overline{x}) \in \mathit{Ref}(\Sigma(\mathbf{dcltype}(x)) \\
\mathbf{type}(x) & \overset{def}{=} & \begin{cases} \mathbf{dcltype}(x) & \text{if } \mathbf{dcltype}(x) \text{ is primitive} \\ \mathbf{type}(\mathit{head}(\overline{x})) & \text{otherwise} \end{cases} \\
\mathbf{value}(x) & \overset{def}{=} & \mathit{head}(\overline{x})
\end{array}$$

- The *null* object expression,

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

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

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

- 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:

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

$$\begin{array}{lcl}\mathbf{D}(le.a) & \overset{def}{=} & \mathbf{D}(le) \wedge \mathbf{type}(le).a \in \mathbf{visattr} \\ \mathbf{value}(le.a) & \overset{def}{=} & \mathbf{vaule}(le).a \\ \mathbf{type}(le.a) & \overset{def}{=} & \mathbf{type}(\mathbf{value}(le).a)\end{array}$$

- The following exemplifies the well-definedness and evaluation of built-in expressions

$$\begin{array}{lcl}\mathcal{D}(e/f) & \overset{def}{=} & \mathcal{D}(e) \wedge \mathcal{D}(f) \wedge \mathbf{type}(e) = Real \\ & & \wedge\ \mathbf{type}(f) = Real \wedge \mathbf{value}(f) \neq 0 \\ \mathbf{value}(e/f) & \overset{def}{=} & \mathbf{value}(e)/\mathbf{value}(f)\end{array}$$

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

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

4.2.4 Semantics of commands

A typical aspect of an execution of an object-oriented program is about how objects are to be attached to program variables (or entities [43]). 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.

Command *skip* terminates and does not change the state

$$skip \overset{def}{=} \emptyset : (true \vdash true)$$

Command *chaos* has the weakest specification

$$chaos \overset{def}{=} \emptyset : (false \vdash true)$$

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.

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

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.

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

Condition choice: is defined in the traditional way:

$$\begin{array}{rl} P \lhd b \rhd Q \overset{def}{=} & (\mathcal{D}(b) \wedge \mathbf{type}(b) = Bool) \\ \Rightarrow & (P \wedge \mathbf{value}(b) \vee Q \wedge \neg\mathbf{value}(b)) \end{array}$$

We use

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

to denote the multiple choice statement. Its semantics is defined to be the design

$$\bigwedge_{i=1}^{n} (\mathcal{D}(b_i) \wedge \mathbf{type}(b_i) = Bool) \Rightarrow \bigvee_{i=1}^{n} (\mathbf{value}(b_i) \wedge c_i)$$

Non-deterministic choice is defined as

$$P \sqcap Q \overset{def}{=} P \vee Q$$

Sequential composition corresponds to relational composition:

$$P(s, s'); Q(s, s') \overset{def}{=} \exists m \cdot P(s, m) \wedge Q(m, s')$$

Loop statement is defined in terms of the weakest fixed point:

$$b * P \overset{def}{=} \mu X.(P; X) \lhd b \rhd skip$$

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.

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

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:

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

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

$$\mathbf{end}\ x \overset{def}{=} \{\mathbf{var}, x\} : (\mathbf{x} \in \mathbf{var}) \vdash \\ \begin{pmatrix} (|\mathbf{var}(x)| = 1 \wedge \mathbf{var}' = \{x\} \unlhd \mathbf{var}) \vee \\ (|\mathbf{var}(x)| > 1 \wedge \overline{x}' = tail(\overline{x}) \wedge \mathbf{var}' = \{x\} \unlhd \cup \{(x, tail(\mathbf{var}(x)))\}) \end{pmatrix}$$

Read in Value: is defined as

$$read(T\ x) \overset{def}{=} \exists c_0 \cdot (\mathbf{var}\ T\ x = c_0)$$

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 vr.

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

where x, y, z are the value, result and value-result parameters of the method of class **type**(le), and $\Psi(N.m)$ stands for the design associated with method m of class N, that will be defined in Section 4.2.6.

4.2.5 Semantics of a class declaration

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

(i) N has not been declared before: $N \notin \mathbf{cn}$, N and M are distinct: $N \neq M$, and the attribute names of N are distinct

$$distinct(\underline{u} \cdot \underline{v} \cdot \underline{w})$$

where *distinct* can be obviously defined.

(ii) The initial values of the attributes matches their corresponding types.

$$\forall i : 1..m \bullet \mathbf{type}(u_i) = U_i \wedge \forall i : 1..n \bullet \mathbf{type}(v_i) = V_i \\ \wedge\ \forall i : 1..k : \bullet \mathbf{type}(w_i) = W_i$$

(iii) The method names are distinct

$$distinct(< m_1, \ldots, m_\ell >)$$

(iv) The parameters of every method are distinct.

$$\forall i : 1..\ell \bullet \begin{pmatrix} \mathit{distinct}(\underline{x}_i \cdot \underline{y}_i \cdot \underline{z}_i) \wedge \\ |\underline{x}_i| = |\underline{T}_{i1}| \wedge |\underline{y}_i| = |\underline{T}_{i2}| \wedge |\underline{z}_i| = |\underline{T}_{i3}| \end{pmatrix}$$

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.

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

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.

4.2.6 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 4.2.5), the design *init*, and the meaning of command P:

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

where the design *init* performs the following tasks

(i) to check the well-definedness of the declaration section,

(ii) to decide the values of **attr** and **visattr** from those of **pri**, **pro** and **pub**.

(iii) to define the meaning of every method body c.

The design *init* is formalized as:

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

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 M such that $N \preceq M$, and every method $m \in \mathbf{op}(M)$ an equation

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

where F is constructed according to the following cases.

Case (1) m is not defined in N, but in a superclass of N, i.e. $m \notin \mathbf{op}(N) \wedge m \in \cup\{\mathbf{op}(M) \mid N \preceq M\}$. Then a call to the method m in the environment of N will be executed according to the definition of m in the *lowest* superclass of N that declares the method. Let M be that lowest superclass of N, i.e. m is declared in M but no subclass of M declares m. The defining equation for $F_{N.m}(\Psi)$ in this case is

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

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

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

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

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

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

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

$$\phi_N(skip) \overset{def}{=} skip$$

$$\phi_N(chaos) \overset{def}{=} chaos$$

$$\phi_N(p_1; p_2) \overset{def}{=} \phi_N(p_1); Set(N); \phi_N(p_2)$$

$$\phi_N(P_1 \triangleleft b \triangleright P_2) \overset{def}{=} \phi_N(P_1) \triangleleft \phi_N(b) \triangleright \phi_N(P_2)$$

$$\phi_N(P_1 \sqcap P_2) \overset{def}{=} \phi_N(P_1) \sqcap \phi_N(P_2)$$

$$\phi_N(b * P) \overset{def}{=} \phi_N(b) * (\phi_N(P); Set(N))$$

$$\phi_N(\mathbf{var}\ T\ x = e) \overset{def}{=} \mathbf{var}\ x : T = \phi_N(e)$$
$$\phi_N(\mathbf{end}\ x) \overset{def}{=} \mathbf{end}\ x$$
$$\phi_N(C.New(x)) \overset{def}{=} C.New(\phi_N(x))$$
$$\phi_N(le := e) \overset{def}{=} \phi_N(le) := \phi_N(e)$$
$$\phi_N(le.m(v, r, vr)) \overset{def}{=} \phi_N(le).m(\phi_N(v), \phi_N(r), \phi_N(vr))$$
$$\phi_N(m(v, r, vr)) \overset{def}{=} self.m(\phi_N(v), \phi_N(r), \phi_N(vr))$$
$$\phi_N(x) \overset{def}{=} \begin{cases} self.x & \exists M \cdot N \preceq M \wedge x \in \mathbf{attr}(M) \\ x & otherwise \end{cases}$$
$$\phi_N(self) \overset{def}{=} self$$
$$\phi_N(le.a) \overset{def}{=} \phi_N(le).a$$
$$\phi_N(null) \overset{def}{=} null$$
$$\phi_N(f(e)) \overset{def}{=} f(\phi_N(e))$$

Notice that we did not introduce the syntax *super.m* to explicitly indicate the call to a method according to its definition in the superclass. Instead the a method call will be executed according to the definition of method at the lowest position in the inheritance hierarchy. There is no difficulty to introduce *super.m* and define its semantics accordingly.

5 Specifying UML Models

This section uses an iterative development of a library system as an example to show how to specify and reason about UML models of requirements analysis and designs. We refer the reader to our paper [40] for details of formal specification of UML models of requirements, and to [37,36] for a formal semantics of UML sequence diagrams.

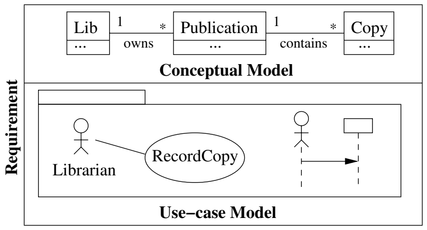
5.1 Conceptual class diagram

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 2. A library *Lib Owns* a number of *Publications* and each publication *Contains* some *Copy*(ies).

We specify this diagram as CM_1 below:

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



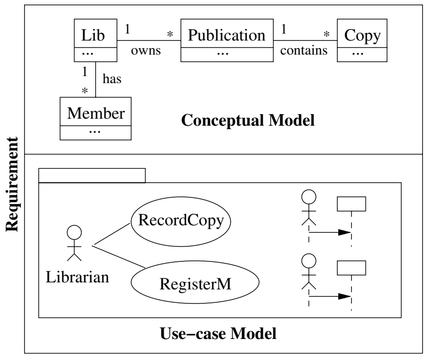


Fig. 2. Models of the Requirements for Cycle 1 (Left) and Cycle 2 (Right)

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. For the above declared conceptual class diagram, we have

$$\mathbb{P}\mathbf{Lib}\ \mathit{Lib} = \varnothing; \mathbb{P}\mathbf{Publication}\ \mathit{Publication} = \varnothing; \mathbb{P}\mathbf{Copy}\ \mathit{Copy} = \varnothing;$$
$$\mathbb{P}\mathbf{Owns}\ \mathit{Owns} = \varnothing; \mathbb{P}\mathbf{Contains}\ \mathit{Contains} = \varnothing$$

5.2 *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[2]. 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 LS (denoting the logic library system):

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

In the specification, we have used 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

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

and c_2 are executed.

Then we define the system specification by defining the **main** method in which
$RCopy \overset{def}{=} read(String\ cid, String\ pid); hrc.RecordCopy(cid, pid)$ in the following statement:

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

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.

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

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

A call to a system operation $m(x, y, z)$ can also be written as a CSP-like process, where x, y and z are the value, result and value-result parameters:

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

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 $(U_{H_1} [] \cdots [] U_{H_\ell})^*$.

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. Although it will not affect the overall functionality of the system, it is recommendable to have a handler class for each use case as in the normal form of the system specification. This makes the method more programmatic and fit in an iterative development process better. In a later stage of the design, we can refine the specification to combine some classes.
- 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 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.

We have a third suggestion in our component calculus [26] to deal with concurrency at the level of component compositions via the interfaces and protocols of components.

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

In general, a use case is more complicated than the one we discussed above and may have alternatives and *exceptional courses of events*. Due to the space limit, we cannot treat such a use case in this paper. However, use cases [33,38] show that the method for informal description of such a general use case, the drawing of its system sequence diagram, and the specification of the design of the system operation identified in the sequence diagram is quite programmatic and can scale up very well. The formalization in OOL can be worked out systematically for the informal description.

By calculating the semantics of $(CM_1; HandleRcopyDecl) \bullet \mathbf{main}()$ *within our model, we can check that the conceptual class diagram is consistent with the use case specified specification.* The semantics supports to check the well-definedness of the declaration section $(CM_1; HandleRcopyDecl)$, the well-definedness of the commands in $\mathbf{main}()$. The well-definedness of $(CM_1; Handle RcopyDecl)$ implies the well-formedness of the corresponding UML class diagram, and the *consistency* between the class diagram and the use case model. The sufficient condition to ensure the overall consistency is that semantics of the composition

$$(CM_1; HandleRcopyDecl) \bullet \mathbf{main}()$$

does not equal to *chaos*. An intuitive understanding about the consistency problem is to think of a static inconsistency, such as missing class or association and conflict names, as an "compiling error" of the program $(CM_1; HandleRcopy Decl) \bullet \mathbf{main}()$, and a dynamic inconsistency, such as violation of pre-condition or invariants and dynamic binding error, as an "execution error" of the program. The advantage of the formalization is that it enables us to reason about inconsistency without compilation and execution and the reasoning can be done at the requirement level. Notice that within our semantic framework, a correction of any inconsistency is then formally treated as a *refinement* of the program specification.

5.3 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

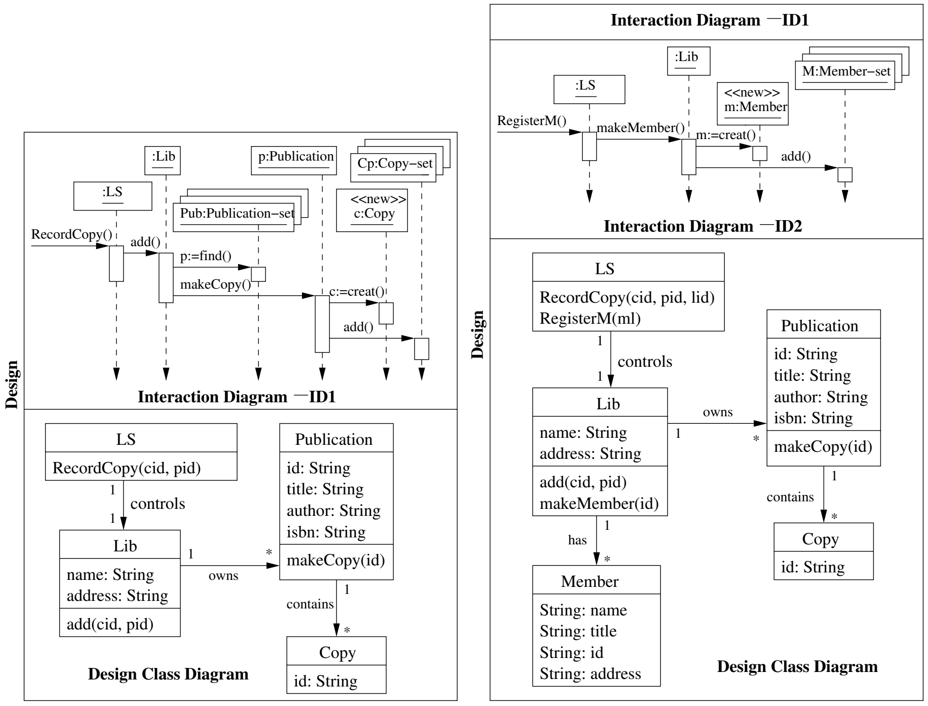


Fig. 3. Models of the Designs for Cycle 1 (Left) and Cycle 2 (Right)

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. Formal definitions of sequences diagrams are given in our related work in [37,36]. For example, the object sequence diagram and its design class diagram in Figure 3 (on the left) are specified by the following class declarations:

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

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

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

Define the data refinement mapping:

$$\Sigma(Pub) \overset{def}{=} ls.lib.Pub, \qquad \Sigma(Contains) \overset{def}{=} \bigcup_{p \in lib.Pub} \{p\} \times p.Cp$$
$$\Sigma(Copy) \overset{def}{=} \bigcup_{p \in ls.lib.Pub} p.Cp, \qquad \Sigma(Owns) \overset{def}{=} \{ls.lib\} \times ls.lib.Pub$$

With our model and refinement calculus, we can check whether a design class diagram is consistent with a family of sequence diagrams, that this design refines *the program that represents the use-case specification.* Details about the consistency of a model of design and the link between a UML model of design to a UML of requirements are formally addressed in [37,36]. The refinement involves in adding or removing attributes, methods, classes and associations; delegating methods; and refining commands. A refinement calculus for object-oriented systems is given in [25] in which all these "refectoring" operations on the structure of an object-oriented program are proven to be valid refinement.

An alternative but still correct design of *RecordCopy* is to redefine the *add*() method in *Lib*,

$Lib :: add()\{\mathbf{var}\ Publication\ , p, Copy\ c;$
$p := Pub.find(pid); Copy.New(c); p.addCopy(c); \mathbf{end}\ p, c$
$\}$

and to replace *Publication* :: *makeCopy*() by

$Publication :: addC(\mathbf{val}\ Copy\ c)\{Cp.add(c)\}$

However, this design adds an extra *dependency* between *Lib* and *Copy*.

5.4 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 2 to the one on the right, that is denoted by CM_2, by adding the following two classes.

Class *Member* {*String name*, *String title*, *String id*, *String address*};
Class *Has* {*Lib lib*, *Member m*}

Let *SList* be the type $String \times String \times String \times String$ and $details \overset{def}{=} (name, title, id, address)$ denote the tuple of the attributes of *Member*. We can 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 and a method *RegisterM*. However, we can also use the same use case handler class *LS* to handle this new use case too:

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

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

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

guarded by $s =$ "$RegisterM$"

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

Following the design patterns in [33], we can work out the interaction diagram and the design class diagram on the right Figure 3.

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

The **main** method is nearly the same as in the requirement specification.

We prove the design of $RegisterM$ is correct by defining the data refinement mapping

$$\Sigma(Member) \overset{def}{=} ls.lib.M \quad \Sigma(Has) \overset{def}{=} \{< ls.lib, m > \ \mid m \in ls.lib.M\}$$

With CM_2, we can specify and design use cases $SearchMember$, $SearchPub$ and $SearchCopy$.

Borrow a copy

Now let us consider the use case $BorrowCopy$ that records the fact that a registered *member* has borrowed a copy from the library. We need to introduce a new concept *Loan* that records the information of a loan of a copy by a member. The *Loan* class is associated with classes *Member* and *Copy* so that

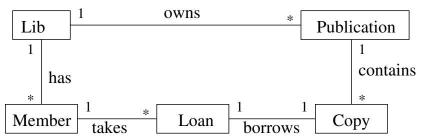


Fig. 4. Conceptual model for *BorrowCopy*

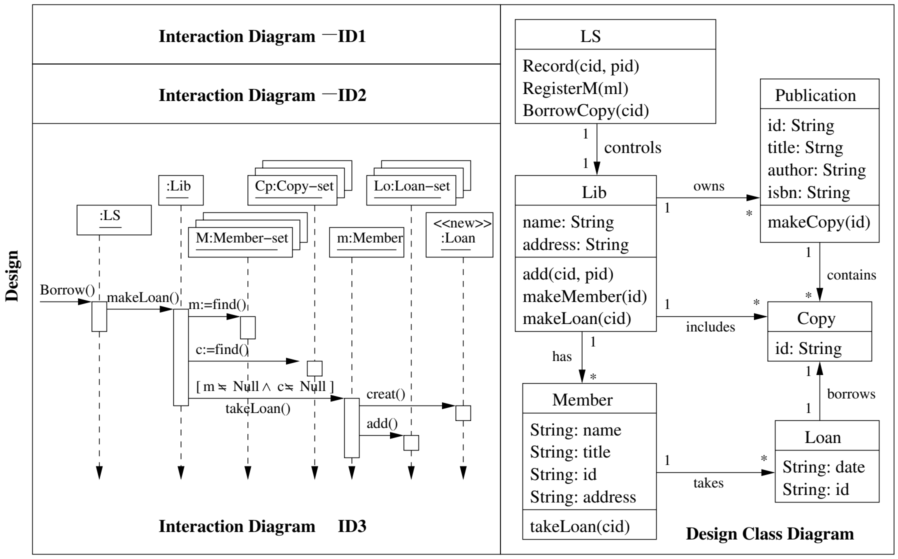


Fig. 5. Design model for cycle 3

a member *Takes* the loan which *Borrows* a copy. The conceptual model CM_2 is extended CM_3 in Figure 4, and specified as follows:

$CM_3 \overset{def}{=} CM_2$; //**Reuse conceptual model
Class *Loan* $\{String\ date,\ String\ id\}$;
Class *Takes* $\{Member\ m,\ Loan\ loan\}$ **Association Class**
Class *Borrows* $\{Copy\ c,\ Loan\ loan\}$ **Association Class**

We specify the use case as a method of the same handler class *LS*, though we can use a new handler.

```
LS:: $BorrowCopy($**val** $String\ cid, String\ mid)\{$
    $\exists m \in Member \bullet member.id = mid$
    $\wedge \exists c \in Copy \bullet c.id = cid \wedge \neg \exists loan \in Loan \bullet <loan, c> \in Borrows \;\Rightarrow$
    **var** $Loan\ loan, Member\ m$;
    $Loan.New(loan) \wedge m := Members.find(mid)$;
    $Loan' = Loan \cup \{loan\} \wedge$
    $Borrows' = Borrows \cup \{<loan, c>\} \wedge Takes' = Takes \cup \{<m, loan>\}$;
    **end** $loan, m$
    $\}$
```

As the second iteration, add "*BorrowCopy*" to *Service*, and define

$BCopy \overset{def}{=} read(String\ cid, String\ mid); LS.BorrowCopy(cid, mid)$

The **main** program is specified as

```
main(){ var $Bool\ stop, Service\ s$;
$stop := true$; while $\neg stop$ do{ $read(s); read(stop)$
if {$s = "RecordCopy" \longrightarrow RCopy, s = "RegisterM" \longrightarrow RegisterM,$
$s = "BorrowCopy" \longrightarrow BCopy$} fi;
end $s, stop$}
}
```

The **main** program can be extended as before by adding the new *BorrowCopy* service.

The same conceptual model CM_3 supports the specification of the use cases for finding the number of *loans* for a member *MGetLoanSize* and for a copy *CGetLoanSize*.

Design use case *BorrowCopy*

We give a design by adding an attribute $\mathbb{P}Copy\ Copy$ and a method *makeLoan*() to class *Copy*, an attribute $\mathbb{P}Loan\ Loan$ and a method *takeLoan*() to class *Member*, redefine class *Loan* so that it has an attribute *Copy c*, and redefine the method *BorrowCopy*() in *LS*. We give a UML design model in Figure 5, and its specification below:

```
$Lib :: \mathbb{P}Copy\ Copy = \varnothing$;
$Lib :: makeLoan($val $String\ mid, String\ cid)${
    var $Member\ m, Copy\ c$;
    $m := Member.find(mid) \wedge c := Copy.find(cid)$;
    $m.takeLoan(c) \lhd (c \neq null \wedge m \neq null) \rhd chaos$;
    end $m, c$ };

$Member :: \mathbb{P}Loan\ Loan$;
Member :: $takeLoan($val $Copy\ c)${
    var $Loan\ loan; loan := Loan.New(loan)[c]$;
    $Loan.add(loan)$; end $loan$
    };
Class $Loan$ {$Date\ date, String\ id, Copy\ c$};
$LS :: BorrowC($val $String\ mid, String\ cid)${
    $lib.makeLoan(mid, cid)$
    }
```

More use cases, such as *Return a Copy*, can be developed in the next iteration.

6 Conclusion

Based on UTP [29], we have presented a model for object-oriented 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 object-oriented programming, require revising only the affected parts of the model and not the model as a whole. The model allows us to compose the static UML view with the UML functional view of the requirements, and the static UML model with the UML interaction model of the design. It also supports the verification of the correctness of a design model against a requirement models.

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. For example, consider the need to impose the restriction on the use case $BorrowCopy$ that a member is only allowed to take a limited number k of loans. We only need to add a method $NBorrow()$ method in LS

$$\begin{array}{l} LS :: NBorrow(\mathbf{val}\ String\ mid, String\ cid)\{ \\ \mathbf{Int}\ n := LoanSize(mid); \\ self.Borrow(mid, cid) \lhd (n < k) \rhd (\text{"}\mathbf{error}\text{"}) \end{array}$$

Then we replace $Borrow$ use case with $NBorrow$ in the **main** program. This does not need to change the implementation of any other *core* classes of the system.

Because our approach supports refinement of models by introducing modelling elements in an incremental manner, it clearly supports the Sketch and blueprint modes of UML usage pointed out by Martin Fowler in his invited talk at <<UML>> 2003 [19]. Also, our formalization of UML model in a specification in a Java-like notation can be seen as a step toward third mode of usage of UML suggested by Martin Fowler as a programming language. Being able to use UML as a programming language is very important for the success of MDA [19,6,42].

7 Related Work

7.1 Models of object-oriented programs

There is a large number of publications on models for object-oriented programming, e.g. [1,2,8,4,10,45]. A large body of work on modelling object-oriented programming is based on type theories or operational semantics. Our approach is among those that are state-based and uses a predicate logic.

State-based formalisms have been used in conjunction with object-oriented techniques, via languages such as Object-Z [10] and VDM^{++} [15], and methods such as Syntropy [13] (which uses the Z notation) and Fusion [12] (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 object-oriented mechanisms, such as dynamic binding and polymorphism.

Cavalcanti and Naumann define an object-oriented programming language with subtype and polymorphism using predicate transformer [11,45]. Mikhajlova and Sekerinski [44] design a rich object-oriented language by using a type system and predicate transformers as well. However, neither reference types nor mutual dependency between classes is allowed in those approaches.

There are a number of recent articles on Hoare Logics for object-oriented programming (see, e.g. [46,52,30,47]). The normal form of a program in our paper is similarly to those in [11,46]. However, a class declaration (section) and a program, as well as a command, are represented as predicates called *designs* in UTP [29]. This provides us with a formal characterization of the contextual/structural feature of the object-oriented programs and a *structural refinement relation* between object-oriented programs [25]. Also the refinement relation between programs is defined as implication between their designs. The proof of a refinement will be carried out in the predicate logic rather than in a Hoare logic proof system. The notion of structural refinement supports a formal treatment of refectorings [20] and object-oriented designs to support iterative and incremental development. This has turned out to be essential when we use this model to formalize and compose different UML models. 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 [29,54,50], makes our approach ready for extension to cope with these different aspects of object-oriented programs. Alternatively, one can also use temporal logic, such as [3], for the specification and verification of multithreading Java-like programs. However, we would like to deal with concurrency at a higher level when we extend this model for component-based software development [27,26].

7.2 Formalizations of UML

The research on formal support for UML modelling (e.g. [18,5,17,16,22,48]) is currently very active . 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 directly towards improved support for requirement analysis and transition from requirements to design models in RUP. The notion of structural refinement enables us to transform UML models consistently. This is very important for UML-based development as these transformations allow developers to model a system at different level of abstraction. In fact the article [18] pointed out the need of such correctness preserving manipulation of UML models and made the initial attempt to provide some transformations. Unfortunately, it could not go far enough and only considered transformation of class diagrams to preserve state invariant. 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.

An earlier version of the semantic model for OOL was presented in [24] that does not deal with references and nested local variable declaration. Based on [24], the notions of refinements for commands, class declarations and programs are defined in [25]. Our idea about combining a conceptual class model and a use-case model was initially presented in [39], without using a specification language. It was embedded into a use-case driven approach for requirement analysis in the paper [34]. These two papers evolved into a comprehensive method for UML-based requirement analysis in [40] with the ideas of normal form of specification. The completeness of the model for requirement analysis can be justified by the computation model presented in [41]. Then we established the semantic model for OOL with references and nested local variable declaration in the report [28]. We have recently used OOL to give a formal semantics of UML interaction diagrams in [37,36]. This paper presents the formal semantics of OOL given in [28] and embeds it in the context of RUP and UML based software development.

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 necessary automated tools [35]. 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 as *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 [40].

7.3 Future work

We are currently working on a comprehensive set of refinement laws to support both behavioural and structural refinement of object-oriented designs. Future work also includes the extension of this method to component-based software development (e.g. [14,51]) so that components can be developed in parallel, and the application of this framework to formal treatment of *patterns* [21]. Defining an executable semantics for the modelling language in this paper and thus to provide an executable semantics for (a subset of) UML will be attractive to the MDA [19,6,42] community[3].

In addition, tool support, e.g. in the direction of [23], for formal object-oriented methods is an area of considerable significance for further industrial take-up of these methods. We are also interested in studying the difference and relationship between our model and Separation Logic [49].

Acknowledgments

First of all, we are very much grateful to the two referees for their careful reading, insight comments and for the technical errors they pointed, that have helped a lot to bring the paper into this revised form. We thank Dines Bjorner at Technical University of Denmark, Anders Ravn from Aalborg University of Denmark and Uday Reddy from Birmingham University of the UK for their helpful comments and discussions at and after the seminars that the second author gave on parts of the works when he visited them. Our UNU-IIST fellows Quan Long, Bhim Upadhyaya and Jing Yang also read and gave useful comments on earlier versions of the article. The second author would also like to thank the students at the University of Leicester and those participants of the UNU-IIST training schools and courses who took his course on Software Engineering and System Development for their feedback on the understanding of the use-case driven, incremental and iterative OO development and the design patterns.

[3] A number of panellists of the Panel of "What should a good modelling language look like" at UML'03 pointed out that a modelling language should be executable in terms of having a compiler so the testing and simulation can be carried out effectively.

References

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

[2] 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.

[3] E. Abraham-Mumm, F.S. de Boer, W.P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *Foundations of Software Science and Computation Structures, Lecture Notes in Computer Science 2303*, pages 5–20. Springer, 2002.

[4] 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*, Lecture Notes in Computer Science 489, pages 60–90. 1991.

[5] 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.

[6] J. Benivin. MDA: from hype to hope and reality. In P. Srevens, J. Whittle, and G. Booch, editors, *<<UML>> 2003 -The Unified Modeling Language, 6th International Conference, Lecture Notes in Computer Science 2863*. Springer, 2003.

[7] B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, May 1988.

[8] 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,* Lecture Notes in Computer Science 1422, pages 68–95. Springer, 1998.

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

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

[11] A. Cavalcanti and D. Naumann. A weakest precondition semantics for an object-oriented language of refinement. In *Lecture Notes in Computer Science 1709*, pages 1439–1460. Springer, 1999.

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

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

[14] D. D'Souza and A.C. Wills. *Objects, Components and Framework with UML: The Catalysis Approach.* Addison-Wesley, 1998.

[15] E. Dürr and E.M. Dusink. The role of VDM^{++} in the development of a real-time tracking and tracing system. In J. Woodcock and P. Larsen, editors, *Proc. of FME'93*, Lecture Notes in Computer Science 670. Springer-Verlag, 1993.

[16] A. Egyed. Scalable consistency checking between diagrams: The Viewintegra approach. In *Proc. 16th IEEE ASE*, San Diego, USA, 2001.

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

[18] A. Evans, *et al.* Developing the UML as a formal modelling notation. In *Proc. UML'98, Lecture Notes in Computer Science 1618.* Springer-Verlag, 1998.

[19] M. Fowler. What is the point of UML. In P. Srevens, J. Whittle, and G. Booch, editors, *<<UML>> 2003 -The Unified Modeling Language, 6th International Conference, Lecture Notes in Computer Science 2863*. Springer, 2003.

[20] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

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

[22] 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.

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

[24] 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.

[25] J. He, Z. Liu, and X. Li. Towards a refinement calculus for object-oriented systems. In *Proc. ICCI02, Alberta, Canada.* IEEE Computer Society, 2002.

[26] J. He, Z. Liu, and X. Li. A component calculus. In H.D. Van and Z. Liu, editors, *Proc. Of FME03 Workshop on Formal Aspects of Component Software (FACS03), UNU/IIST Technical Report 284, UNU/IIST, P.O. Box 3058, Macao*, Pisa, Italy, 2003.

[27] J. He, Z Liu, and X. Li. Contract-oriented component software development. Technical Report 276, UNU/IIST, P.O. Box 3058, Macao SAR China, 2003.

[28] J. He, Z. Liu, and X. Li. Modelling object-oriented programming with reference type and dynamic binding. Technical Report UNU/IIST Report No 280, UNU/IIST, P.O. Box 3058, Macau, May 2003.

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

[30] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *FASE 2000, Lecture Notes in Computer Science 1783*, pages 284–303.

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

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

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

[34] X. Li, Z. Liu, and J. He. Formal and use use-case driven requirement analysis in UML. Technical Report UNU/IIST Report No 230, UNU/IIST, P.O. Box 3058, Macau, March 2001. A short version is accepted for presentation at COMPSAC2001, October, USA.

[35] X. Li, Z. Liu, and J. He. Generating a prototype from a UML model of system requirements. Submitted for publication, 2003.

[36] X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagrams. In *Pro. of Australian Software Engineering Conference (ASWEC'2004).* IEEE Computer Sciety, 2004.

[37] J. Liu, Z. Liu, J. He, and X. Li. Linking UML models of design and requirement. In *Pro. of Australian Software Engineering Conference (ASWEC'2004).* IEEE Computer Sciety, 2004.

[38] Z. Liu. Object-oriented software development in UML. Technical Report UNU/IIST Report No. 259, UNU/IIST, P.O. Box 3058, Macau, SAR, P.R. China, July 2002.

[39] Z. Liu, J. He, and X. Li. Formalizing the use of UML in requirement analysis. Technical Report UNU/IIST Report No 228, UNU/IIST, P.O. Box 3058, Macau, March 2001. A short version "Towards a formal use of UML for software requirement analysis is accepted for presentation at PDPTA2001, June, 2001, Las Vegas, USA.

[40] Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, Lecture Notes in Computer Science 2885*, pages 641–664. 2003.

[41] Z. Liu, X. Li, and J. He. Using transition systems to unify *uml* models. Technical report, Dept. of Maths and Computer Science, the University of Leicester, England., May 2002.

[42] S.J. Mellor and M.J. Balcer. *Executable UML: a foundation for model-driven architecture.* Addison-Wesley, 2002.

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

[44] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-orient programs. In *Proc of FME'97, Lecture Notes in Computer Science*. Springer, 1997.

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

[46] C. Pierik and F.S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute pf Information and Computing Science, Utrecht University, 2003.

[47] A. Poetzsch-Heffter and P. Muller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Proc. Programming Languages and Systems (ESOP'99), Lecture Notes in Computer Science 1576*, pages 162–176. Springer, 1999.

[48] G. Reggio, *et al.* Towards a rigorous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *Proc. FASE 2001, Lecture Notes in Computer Science 2029*. Springer, 2001.

[49] J. Reynolds. Separation logic: a logic for a shared mutable data structure (invited talk). In *Proceedings of IEEE Symposium Logic in Computer Science (LICS'02)*. IEEE Computer Sciety, 2002.

[50] A. Sherif and J. He. Towards a time model for Circus. In *ICFEM02, Lecture Notes in Computer Science 2495*. Springer, 2002.

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

[52] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

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

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