Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 130 (2005) 57–77
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Towards a Rigorous Approach to UML-Based Development [1]

Zhiming Liu and He Jifeng

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

Xiaoshan Li

Faculty of Science and Technology, University of Macau, Macao SAR, China

Abstract

We discuss the promises and problems of UML-based development. We then suggest a framework in which UML can be used precisely and more disciplined so as to solve the problems and meet the promises better.

Keywords: Object-Orientation, Component-Based Development, Refinement, Specification, Transformation

1 Introduction

When programming in the small, the *specification* of the problem is mainly concerned with the control and data structures of the program. The program development is the design and implementation of data structures and algorithms through a number of steps of *refinement*. *Verification* is needed to prove that each step preserves the specification of the control and data structures in the previous step. Various formal methods, especially those state-based

[1] Email: lzm@iist.unu.edu,hjf@iist.unu.edu,xsl@umac.mo.
J. He is on leave from East China Normal University, Shanghai, China.
J. He's 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.2005.03.005

models [9,18] such as VDM [20] and Z [8], are widely found helpful in correct and reliable construction of such a program.

For programming in the large, problems become more complicated. The specification has to be described in terms of (or decomposed into) *components*, their *interfaces* and *interactions*. Such a specification in general contains different views and aspects, e.g. the static view, the interaction view and the behavioural view. An overall structure, i.e. the system architecture, is required to *consistently* unify these views. The system construction needs a process of model transformations. These transforms have to ensure consistency of the deferent views and preserve the functional and behavioural correctness. Therefore, for programming in the large, it is ideal in general to have a multi-view modelling language that supports specifications at different levels of abstraction. In this article, we will discuss how UML can be used for this purpose and how it can be used formally for correct and reliable construction of large scale software.

1.1 The promises of UML

UML2.0 is designed as a modelling language for *component-based* and *object-oriented* development, promising to support programming in the large. It is obviously a multi-view and multi-notional language and we can count up to at least 10 kinds of diagrams including component diagrams, packages, class diagrams, object diagrams and use-case diagrams for static views; activity diagrams, interaction diagrams (sequence diagrams and collaboration diagrams) and statecharts for concurrency, interaction and behavioural aspects; and deployment diagram for deployment. A textual specification language, *Object Constraint Language* (OCL), is also part of UML for writing constraints on the diagrams and pre- and post-conditions of operations and methods.

The multi-view and multi-notational aspect of UML has an obvious good purpose to allow the split of an overall system model into several views and decompose it into chunks of manageable size. Each single view focuses on a different aspect and this will ease for analysis and understanding. This decomposability of the model enable the development team to split the work of producing models among different people. It is also important for tool support as it would be more difficult for a tool to deal with one large and complex model. UML also intends to support modelling a system at different levels of abstraction. However, without clear means for information hiding, this promise is not as obvious as the one discussed earlier and we need more effort to make UML support model transformation and refinement better.

1.2 Problems in using UML

When applying UML to real software projects, several challenging issues inevitably arise from such a multi-view and multi-notational approach [34]:

- *Consistency:* the models of various views need to be syntactically and semantically compatible with each other (i.e. horizontal consistency) [12,3],
- *Transformation and evolution:* a model must be semantically consistent with its refinements (i.e. vertical consistency) [12,3].
- *Traceability:* a change in the model of a particular view leads to corresponding consistent changes in the models of other views.
- *Integration:* models of different views need to be seamlessly integrated before software production.

Consistency checking and formal analysis of UML models have been widely studied in recent years [14,4,2,13,11,37]. The majority of them focus on the formalization of individual diagrams and only treat the consistency of the models of one or two views. Another phenomenon in research on formal use of UML is that different communities intend to emphasize different notations and use the full or even extended power of, say sequence diagrams or state machines. This would lose the advantages of the multiple-view modelling and the increase in the complexity of a certain kind of models and the reduction of the role that the other kinds of UML models can play. To our knowledge, there is little work on consistent refinement of complete UML models of systems. A complete model of a system here means a family of models for the different views of the system. A majority of methods in the literature use a *transformational approach* that translate some UML models into an existing formalism, such as B, Z, and CSP, and then employs the existing theories and tools of the corresponding formalisms for the analysis and manipulation of the translated models. A problem with such an approach is that most of the translations are not *reversible* and the manipulated model will not be able to be converted back to a UML model for comparison. This would cause difficulties for most UML users.

Some researchers, e.g. the authors of [12,3], have realized the conditions and solutions for consistency depend on the diagrams involved, the development process employed, and the current stage of the development. The difficulties in consistency checking lie in the fact that the syntax and semantics of UML are informal and imprecise compared to formal modelling notations. For example, many features including role names in class diagrams and object names in sequence diagrams are optional and may not appear in the diagrams. This causes no harm if UML is only used in its sketchy mode, but it is not

satisfactory in the modes of blueprint and programming language [15]. Also different models describe overlapping aspects of a system. A particular notation, such as statecharts that in theory can specify all semantic information of sequence diagrams (and activity diagrams) and vice versa, has the power or potential to describe nearly all aspects of a system. This complicates the problems in consistency checking. A good project should restrict the roles of the diagrams to the viewpoints that they intend to represent so as to reduce the amount of overlapping among the different models.

1.3 Towards a rigorous approach to UML-based development

In fact the problems of UML are the problems of programming in the large in general, when a multi-view and multi-notaional modelling approach is to be used. To deal with the problems of consistency and refinement in the application of UML, we provide a common semantic model that can define the UML models and integrates them consistently. We do this by developing a relational model for **R**efinement of **C**omponent and **O**bject **S**ystems (rCOS) [17,28]. To ease the difficulties mentioned in the previous section in dealing with the problems of consistency and refinement, we target at a particular *use-case driven*, *incremental* and *iterative development* process (RUP) [19,21], and restrict the syntax and semantics of a kind of UML models to the roles that they play in the development to the viewpoints that they represent. We only allow the use models that are definable in rCOS for formal analysis and transformation – at the moment they are component diagrams, class diagrams, sequence diagrams for system operations, state machines for classes and activity diagrams.

After this introduction, we give an introduction in Section 2 to the refinement calculus for object systems. We then outline the ideas in Section 3 about how the model for object systems is to be extended to deal with component systems. In Section 4, we briefly discuss the relation between object systems and component systems.

2 A Relational Model for Object Systems

In this section we give an introduction to the syntax and semantics of rCOS . We will omit most of the formal details about the semantics and refer the reader to [17].

2.1 Syntax

In rCOS , a system (or program) S is of the form $cdecls \bullet P$, consisting of *class declaration section* $cdecls$ and a *main method* P. The main method P is a pair $(\texttt{glb}, c)$ of a finite set `glb` of *global variables declarations* and a command c. P can also be understood as the *main method class* in Java. The class declaration section $cdecls$ is a finite sequence of class declarations $cdecl_1; \ldots; cdecl_k$, where each class declaration $cdecl_i$ is of the following form

$$
\begin{array}{l}
[\texttt{private}]\ \texttt{class}\ N\ \texttt{extends}\ M\ \{ \\
\quad \texttt{private}\ \ U_1\ u_1 = a_1, \ldots, U_m\ u_m = a_m; \\
\quad \texttt{protected}\ V_1\ v_1 = b_1, \ldots, V_n\ v_n = b_n; \\
\quad \texttt{public}\ \ W_1\ w_1 = d_1; \ldots W_k\ w_k = d_k; \\
\quad \texttt{method}\ \ m_1(\underline{T}_{11}\ \underline{x}_1, \underline{T}_{12}\ \underline{y}_1, \underline{T}_{13}\ \underline{z}_1)\{c_1\}; \\
\qquad\qquad \cdots; \\
\qquad\qquad m_\ell(\underline{T}_{\ell 1}\ \underline{x}_\ell, \underline{T}_{\ell 2}\ \underline{y}_\ell, \underline{T}_{\ell 3}\ \underline{z}_\ell)\{c_\ell\} \\
\quad \}
\end{array}
$$

Note that

- A class can be declared as `private` or `public`, but by default it is assumed as `public`. Only a public class or a primitive type can be used in the global variable declarations `glb`. In [17], *structural refinement* laws allow us to add, delete, change (e.g. adding, deleting or changing attributes or methods), decomposing or composing private classes and associations among them without changing the behavioural of a system. Refinement laws are also available for consistent change in public classes and the main method.
- N and M are distinct names of classes, and M is called the direct superclass of N.
- Attributes annotated with `private` are private attributes of the class, and similarly, the `protected` and `public` declarations for the protected and public attributes. Types and initial values of attributes are also given in the declaration.
- 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 $(\underline{T}_{i3}\ z_i)$ and bodies (c_i). We sometimes denote a method by $m(\underline{paras})\{c\}$, where $\underline{paras}$ is the list of parameters of m and c is the body command of m.

  The body of a method c_i is a command that will be defined later.

We will use Java convention to write a class specification, and assume an attribute `protected` when it is not tagged with `private` or `public`. We have these different kinds of attributes to show how visibility issues can be dealt with. We can have different kind of methods too for a class. However, we omit the

declaration of private or public methods for the simplicity of the theory. Instead, we assume all methods in public classes are public and can be inherited by a subclass and accessed by the main method, and all methods in private classes are protected.

2.1.1 Commands

rCOS supports typical object-oriented programming constructs, but it also allows some commands for the purse of specification and refinement:

$$c ::= \mathit{skip} \mid \mathit{chaos} \mid \mathbf{var}\ T\ \text{x=e} \mid \mathbf{end}\ x \mid c; c \mid c \lhd b \rhd c \mid c \sqcap c \\ \mid b * c \mid le.m(\underline{e}, \underline{v}, \underline{u}) \mid le := e \mid C.new(x)[\underline{e}]$$

where b is a Boolean expression e is an expression, and le is an expression which may appear on the left hand side of an assignment and is of the form $le ::= x \mid le.a$ where x is a simple variable and a an attribute of an object. Unlike [35] that introduces "statement expressions", we use $le.m(\underline{e}, \underline{v}, \underline{u})$ to denote a call of method m of the object denoted by the left-expression le with actual value parameters $\underline{e}$ for input to the method, actual result parameters $\underline{v}$ for the return values, and value-result parameters $\underline{u}$ that can be changed during the execution of the method call and with their final values as return values too; and use the command $C.new(x)[\underline{e}]$ to create a new object of class C with the initial values of its attributes assigned to the values of the expressions in $\underline{e}$ and assign it to variable x. Thus, $C.new(x)[\underline{e}]$ uses x with type C to store the newly created object.

2.1.2 Expressions

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

$$e ::= x \mid \mathit{null} \mid \mathit{self} \mid e.a \mid e\ \texttt{is}\ C \mid (C)e \mid f(e)$$

where *null* represents the special object of the special class *NULL* that is a subclass of all classes and has *null* as its unique object, *self* will be used to denote the active object in the current scope (some object-oriented languages uses `this`), $e.a$ is the a-attribute of e, $(C)e$ is the type casting, e **is** C is the type test.

2.2 Semantics

rCOS adopts an observation-oriented and relational semantics. It supports refinement of object-oriented designs at different levels of abstraction during a system development. Here we will give a brief overview of the semantics, as for more details, please refer [17].

To formalize the behavior 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, Boolean values, 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.

Our model describes the behavior of an object-oriented program by a design containing the following logical variables as its *free variables* that form the *alphabet* in [18] of the program.

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

(ii) Each class $C \in \texttt{cname}$ is associated with

(a) $\texttt{attr}(C)$: the set of (declared or inherited) attributes of class C

$$\{\langle a_1 : T_1, d_1\rangle, \cdots, \langle a_m : T_m, d_m\rangle\}$$

where T_i and d_i are the type and initial value of attribute a_i, and will be referred by $\mathbf{decltype}(C.a_i)$ and $\mathbf{init}(C.a_i)$ respectively. We also abuse the notation $a \in \texttt{attr}(C)$ and use it to denote

$$\exists T, d \cdot (\langle a : T, d\rangle \in \texttt{attr}(C))$$

Again, we do not allow attribute hiding (or redefinition) in a subclass. We also use an attribute name to represent its value and a type name to denote the set of its legal values.

(b) $\texttt{op}(C)$: the set of (declared and inherited) methods of class C. We allow method overriding, but not signature redefinition in a subclass.

$$\{m_1 \mapsto (\underline{x}_1 : \underline{T}_{11}, \underline{y}_1 : \underline{T}_{12}, \underline{z}_1 : \underline{T}_{13}, \Psi(C.m_1)), \cdots, m_k \mapsto (\underline{x}_k : \underline{T}_{k1}, \underline{y}_k : \underline{T}_{k2}, \underline{z}_k : \underline{T}_{k3}, \Psi(C.m_k))\}$$

which states that each method m_i has $\underline{x}_i$, $\underline{y}_i$ and $\underline{z}_i$ as its value, result and value-result parameters respectively, that are denoted by $\mathbf{val}(C.m_i)$, $\mathbf{res}(C.m_i)$, and $\mathbf{valres}(C.m_i)$, and the behavior of m_i is defined by the design $\Psi(C.m_i)$ that will be defined later in this section. When we are not interested in distinguishing the value, result and value-result parameters, we simply denote each element in $\texttt{op}(C)$ as $m_i \mapsto (\underline{paras_i}, \Psi(C.m_i))$. We also sometimes abuse the notation $m \in \texttt{op}(C)$ and use it to denote

$$\exists \underline{paras}, D \cdot (m \mapsto (\underline{paras}, D) \in \texttt{op}(C))$$

The variables in (a) and (b) are modified by class declarations.

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

(iv) $\Sigma(C)$: the set of objects of class C that currently exist in the execution of the program, and it will be updated through object creation or destruction. We let

$$\Sigma \stackrel{def}{=} \bigcup_{C \in \mathtt{cname}} \Sigma(C)$$

These variables, $\{\Sigma(C) \mid C \in \mathtt{cname}\}$ are changed by the creation of a new object (and the destruction of an existing object or garbage collection that we do not consider in this article).

We call Σ the *system state* of the program. It in fact corresponds to the *current configuration* of the program in [35]. It is also the UML object diagram of the system representing current system's state [29].

(v) **glb**: the set of global variables with their types known to the program $\{x_1 : T_1, \ldots, x_k : T_k\}$. The type T_i of x_i, denoted by $\mathbf{decltype}(x_i)$, can be either a primitive type or a class name. We assume they are declared at the beginning of the main program and will not be changed afterwards.

(vi) **locvar**: the set of the current declared local variables with their types. This set is to be modified by local variable declaration and undeclaration statements.

Commands and class declarations, as well as an object system as a whole, are semantically defined as a *framed design* $\{V\} : pre(x) \vdash Post(x, x')$ in [18] over the above alphabet.

For $P = (\mathtt{glb}, c)$, the semantics of $cdecls \bullet P$ is defined to be the sequential (relational) composition

$$[\![cdecls \bullet P]\!] \stackrel{def}{=} [\![cdecls]\!]; init;\ [\![c]\!]$$

where

- the semantics of the declaration section *cdecls* calculates the set `canme` of the declared classes, their attributes, methods, and the superclass relation.
- design *init* checks the well-definedness of the declaration section and global variable declarations, initialises the visible attributes, the set of attributes of each declared class, the semantics of methods of each declared class.

When combine them together and get

$$\begin{aligned}([\![cdecls]\!]; init) \stackrel{def}{=}\ & \{\mathtt{cname}, \mathtt{supperclass}, \mathtt{visattr}, \mathtt{attr}, \mathtt{op}\} : \\ & \mathcal{D}(cdecls) \wedge \mathcal{D}(\mathtt{glb}) \vdash \\ & initCname \wedge initSupperclass \wedge initVis \wedge initAttr \wedge initOP\end{aligned}$$

where

(i) *initCname* collects all the declared public class names in *cdecls* into `cname`

(ii) *initSupperclass* sets the *direct superclass* relation `superclass` to contain $N \mapsto M$ if N **extends** M appears in *cdecls*. Let $\preceq$ be the general subclass

relation induced from `superclass`. Also $N \prec M$ means that $N \preceq M$ but N and M are different.

(iii) *cdecls* is well-defined if the attributes names, method names of each class and parameter names of each method are distinct; all types used in *cdecls* are either classes names declared in *cdecls* or primitive types; all superclasses are also declared in *cdecls*; and protected and public attributes of a class are not redeclared in its. subclasses

(iv) The global declaration is well-defined if all types used in it are either primitive types or class names declared in *cdecls*.

(v) *initVis* sets the execution environment `visattr` as the set of all public attributes of the public classes $\{N.a \mid a \in \mathtt{pub}(N), N \in \mathtt{cname}\}$.

(vi) *initAttr* sets $\mathtt{attr}(N)$ for each $N \in \mathtt{canme}$ as the union of the private attributes of N, the protected and inherited protected and public attributes of N's superclasses:

$$\mathtt{attr}(N) \stackrel{def}{=} \mathtt{pri}(N) \cup \bigcup\{\mathtt{prot}(M) \mid N \preceq M\}$$

(vii) *initOp* sets $\mathtt{op}(N)$ for each $N \in \mathtt{cname}$ the set of its own declared methods and inherited from its superclasses:

$$\mathtt{op}(N) \stackrel{def}{=} \{m \mapsto (\underline{paras}, \Psi(N.m)) \mid (m \mapsto (\underline{paras}, c)) \in \mathtt{op}(M) \wedge N \preceq M\}$$

where $\Psi(N.m)$ gives the behaviour of $N.m$ that we define below.

A method of an object of class N is dynamically bound to its current type, and thus we *set*(N) to set the execution to the environment of class N and *Reset* to recover the global execution environment. The behaviour of a method m of a class is defined to be the design $\Psi(N.m)$

$$\Psi(N.m) \stackrel{def}{=} \begin{cases} Set(N); \phi_N(body(N.m)); Reset & m \in N \\ Set(N); \phi_N(body(M.m)); Reset & m \in M \succ N \end{cases}$$

and

$$Set(N) \stackrel{def}{=} \{\mathtt{visattr}\} : true \vdash \mathtt{visattr}' \left(\{N.a \mid a \in \mathtt{attr}(N)\} \cup \bigcup_{M \in \mathtt{cname}} \{N.a \mid a \in \mathtt{pub}(M)\} \right)$$

$$Reset \stackrel{def}{=} \{\mathtt{visattr}\} : true \vdash \mathtt{visattr}' \bigcup_{M \in \mathtt{cname}} \{M.a \mid a \in \mathtt{pub}(M)\}$$

where $\phi_N(body(N.m))$ is recursively defined to prefix each attribute and a

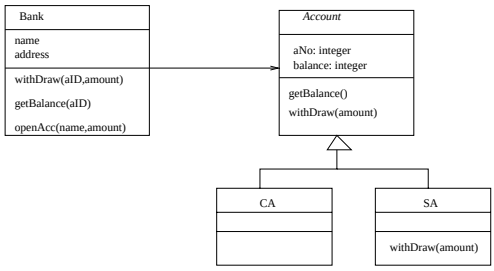

Fig. 1. A bank system

method of class N appearing in the body of m with *self*:

$$\phi_N(skip) \stackrel{def}{=} skip, \quad \phi_N(chaos) \stackrel{def}{=} chaos$$
$$\phi_N(P_1; P_2) \stackrel{def}{=} \phi_N(P_1); Set(N); \phi(P_2)$$
$$\phi_N(P_1 \lhd b \rhd P_2) \stackrel{def}{=} \phi_N(P_1) \lhd \phi_N(b) \rhd \phi_N(P_2)$$
$$\phi_N(P_1 \sqcap P_2) \stackrel{def}{=} \phi_N(P_1) \sqcap \phi_N(P_2)$$
$$\phi_N(b * P) \stackrel{def}{=} \phi_N(b) * (\phi_N(P); Set(N))$$
$$\phi_N(\mathbf{var}\ T\ x = e) \stackrel{def}{=} \mathbf{var}\ T\ x = \phi_N(e), \quad \phi_N(\mathbf{end}\ x) \stackrel{def}{=} \mathtt{end}\ x$$
$$\phi_N(C.new(x)) \stackrel{def}{=} C.new(\phi_N(x)), \quad \phi_N(le := e) \stackrel{def}{=} \phi_N(le) := \phi_N(e)$$
$$\phi_N(le.m(v, r, vr)) \stackrel{def}{=} \phi_N(le).m(\phi_N(v, r, vr))$$
$$\phi_N(v, r, vr) \stackrel{def}{=} (\phi_N(v), \phi_N(r), \phi_N(vr))$$
$$\phi_N(m(v, r, vr)) \stackrel{def}{=} self.m(\phi_N(v), \phi_N(r), \phi_N(vr))$$
$$\phi_N(x) \stackrel{def}{=} \begin{cases} self.x & x \in \bigcup_{N \preceq M} \mathtt{attr}(M) \\ x & otherwise \end{cases}$$
$$\phi_N(self) \stackrel{def}{=} self, \quad \phi_N(le.a) \stackrel{def}{=} \phi_N(le).a$$

Then the semantics of $\phi_N(body(M.m))$ can be defined as a design of the form $pre \vdash Post$ following the Unifying Theories of Programming in [18].

In rCOS , an object program is specified as a sequence class declarations $class_1; \cdots; class_n$, where each class is declared with its fields and methods. With rCOS , a class diagram can be easily formalised as a sequence of class declarations in which associations can be either specified as a class or an attribute of a class [29]. Sequence diagrams and statecharts are formalized and integrated into the body of the methods of the relevant classes [30,25,38,24,26]. Therefore, within the proposed framework, we can provide a consistent integration of UML models in an object-oriented development.

Example

Consider the UML class diagram for a simple bank system in Figure 1. This class diagram can be specified in rCOS as a class declaration section,

denoted by *BankDecls*. The semantics of the class diagram is formalized as

$$
\begin{array}{l}
BankDecls; init = true \vdash \\
\left(\begin{array}{l}
\mathtt{cnname}' = \{Acc,\ CA, Bank\} \wedge \mathtt{sup}' = \{CA \mapsto Acc, SA \mapsto Acc\} \wedge \\
\mathtt{visibattr}' = \emptyset \wedge \\
\mathtt{attr}'(Acc) = \{aNo : Int, balance : Int\} \wedge \\
\mathtt{attr}'(CA) = \{aNo : Int, balance : Int\} \wedge \\
\mathtt{attr}'(SA) = \{aNo : Int, balance : Int\} \wedge \\
\mathtt{attr}'(Bank) = \{name : String, address : String, A : Set[Acc]\} \wedge \\
\mathtt{op}'(Acc) = \{gB \mapsto (\underline{pas}, , \Psi(Acc.gB)), wD \mapsto (\underline{pas}, \Psi(Acc.wD))\} \wedge \\
\mathtt{op}'(CA) = \{gB \mapsto (\underline{paras}, \Psi(CA.gB)), wD \mapsto (\underline{pas}, \Psi(CA.wD))\} \wedge \\
\mathtt{op}'(SA) = \{gB \mapsto (\underline{paras}, \Psi(SA.gB)), wD \mapsto (\underline{paras}, \Psi(SA.wD))\} \wedge \\
\mathtt{op}'(Bank) = \{gB \mapsto (\underline{pas}, \Psi(Bank.gB)), \\
openA \mapsto (\underline{paras}, \Psi(Bank.openA)), wD \mapsto (\underline{paras}, \Psi(Bank.wD))\}
\end{array}\right)
\end{array}
$$

Notice here that we have used some shorthands, such as gB for *getBalance* and wD for *withDraw*.

Not assume that the body of *withDraw*(x) is $balance := balance - x$. Then the design of the *withDraw* method of CA that is inherited from the definition of *Account* is

$$
\begin{array}{rcl}
\Psi(CA.wD) & = & Set(CA); \phi_N(balance := blance - x); Reset \\
& = & \mathtt{visattr} := \{CA.blance,\ CA.aNo\}; \\
& & self.balance := self.balance - x; \\
& & \mathtt{visattr} := \emptyset
\end{array}
$$

However, assume for a saving account of calls SA, *withDraw* is allowed only if the current balance is no less than the amount to withdraw. The method *withDraw*(x) is SA is rewritten from that of *Account* to

$$(balance{:=}blance\text{-}x) \lhd (x \geq balance) \rhd (\text{output "no enough money"})$$

Then the design of the *withDraw* method of SA that overwrites that of *Account* is

$$
\begin{array}{rcl}
\Psi(SA.wD) & = & Set(SA); \phi_N(balance := blance - x); Reset \\
& = & \mathtt{visattr} := \{SA.blance,\ SA.aNo\}; \\
& & self.balance := self.blance - x) \lhd \\
& & (x \geq self.balance) \rhd (\text{out put "no enough money"}); \\
& & \mathtt{visattr} := \emptyset
\end{array}
$$

The above examples show how dynamic binding in our model works.

2.3 Refinement of Object Systems

rCOS includes refinement of class declarations (*structural refinement*, refinement of commands and refinement of a whole system.

Definition 2.1 (**Design refinement**) Design $D_2 = (\alpha, P_2)$ is a *refinement*

of design $D_1(\alpha, P_1)$, denoted by $D_1 \sqsubseteq D_2$, if P_2 entails P_1, i.e.

$$\forall \underline{x}, \underline{x}', ok, ok' \cdot (P_2 \Rightarrow P_1)$$

where $\underline{x}$ are variables contained in α. $D_1 \equiv D_2$ if and only if $D_1 \sqsubseteq D_2$ and $D_2 \sqsubseteq D_1$.

Definition 2.2 (**Data refinement**) Let ρ be a mapping (that can also be specified as a design) from α_2 to α_1. Design $D_2 = (\alpha_2, P_2)$ is a *refinement* of design $D_1 = (\alpha_1, P_1)$ under ρ, denoted by $D_1 \sqsubseteq_\rho D_2$, if $(\rho; P_1) \sqsubseteq (P_2; \rho)$. In this case, ρ is called a refinement mapping.

Definition 2.3 (**System refinement**) Let S_1 and S_2 are object programs which have the same set global variables **glb**. S_1 is a *refinement* S_2, denoted by $S_1 \sqsupseteq_{sys} S_2$, if its behavior is more controllable and predictable than that of S_2:

$$\forall \underline{x}, \underline{x}', ok, ok' \cdot (S_1 \Rightarrow S_2)$$

where $\underline{x}$ are those variables in **glb**.

This indicates the external behavior of S_1, that is, the pairs of pre- and post global states, is a subset of that of S_2. To prove one program S_1 refines another S_2, we require that they have the same set of global variables and the existence of a *refinement mapping* between the variables of S_1 to those of S_2 that is identical on global variables.

Let *stateV* be the set of variables listed above, except for those in **glb**. The semantics of a program is defined by

$$cdecls \bullet P \stackrel{def}{=} \exists stateV, \exists stateV' \cdot (cdecls; init; c)$$

where $\exists setV$ for a set *setV*of variables is a short hand for applying $\exists$ to each of the variables in the set, and the design *init* performs the tasks of computing the initial state. It is explained in details in [17].

Definition 2.4 (**Class refinement**) Let $cdecls_1$ and $cdecls_2$ be two declaration sections. $cdecls_1$ is a *refinement* of $cdecls_2$, denoted by $cdecls_1 \sqsupseteq_{class} cdecls_2$, if the former can replace the later in any object system:

$$cdecls_1 \sqsupseteq_{class} cdecls_2 \stackrel{def}{=} \forall P \cdot (cdecls_1 \bullet P \sqsupseteq_{sys} cdecls_2 \bullet P)$$

where P stands for a main method $(\mathbf{glb}, c)$.

Intuitively, it states that $cdecls_1$ supports at least the same set of services as $cdecls_2$.

The following refinement laws capture the basic principles in object-oriented design and decomposition, and can be used to prove general object-oriented design patterns within the UML framework:

(i) Adding a class declaration: this allows us to add a class into the class

diagram, sequence diagrams and state machines of the methods of the new class.

(ii) Introducing a *fresh* private attribute to a class: this corresponds to adding a fresh attribute of a primitive type to the class or adding a directed association from the class to another in the class diagram.

(iii) Promoting a private attribute of a class to a protected attribute, and a protected attribute to a public attribute: the same refinements can be applied to a class diagram.

(iv) Adding a *fresh* method into a class: this allows us to add a method signature into the class in the class diagram, and add a sequence diagram, modify the state machine to incorporate this method. The newly added methods must not violate any state constraint required by the model.

(v) Refining the body command of a method $m()\{c\}$ in a class: this leads to the replacement of the subsequence diagrams corresponding to the occurrences of $m()$, and refine the actions of transitions with $m()$ as the triggering event in the state machine of the class.

(vi) Introducing inheritance: If none of the attribute of class N is defined in class M or any superclass of M, we can make M a direct superclass of N.

(vii) Moving some attributes from a class to its direct superclass.

(viii) Introducing a fresh superclass to a class: If M is not in the class declaration, we can introduce M and make it a superclass of an existing class N.

(ix) Moving common attributes of classes which are direct subclasses of a class to the superclass.

(x) Moving a method from a class to its direct superclass.

(xi) Copying (**not** removing) a method of a class to its direct subclass.

(xii) Removing unused attributes: for a private attribute, it can be removed if it does not appear in any method of the class; for a protected attribute, it can be removed if it does not appear in any method of the class or any of its subclasses; for a public attribute, it can be removed if it does not appear in any method. This is because the main method does not access attributes directly.

We can also refine a class diagram by flattening it into a diagram without inheritance relations between classes. Refinement rules are also available for the object-oriented design patterns. General Responsibility Assignment Software Patterns (GRASP) [22] is a frequently used object-oriented design technique. We have used the facade controller in a requirement specification.

One of the most important design patterns is called the *expert pattern*, which shows how part of a functionality of a class can be delegated to another class:

Law 1 (**Expert**) *If a method of a class contains a subcommand that can be realized by a method of another class, we can replace that subcommand with a method invocation to the of the latter class (see Figure 2).*

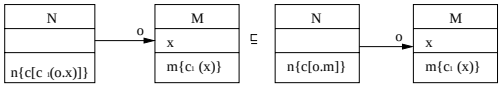

Fig. 2. Expert Pattern

Note that the sequence diagrams and state machines involving $N :: m()$ are refined accordingly. They are not shown here due to the length limit of this paper.

The *Low-Coupling Pattern* of GRASP, on the other hand, can help us remove unnecessary associations to reduce the coupling between classes and simplify reuse and maintenance:

Law 2 (**Low Coupling**) *A call from one class to a method of another can be realized via a third class that is associated with these two classes. This is shown in Figure 3.*

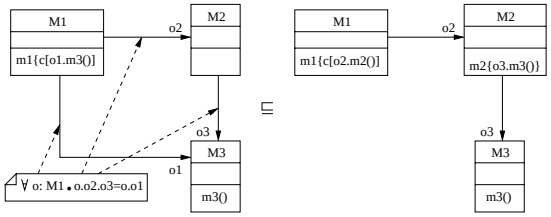

Fig. 3. Low Coupling Pattern

The *High-Cohesion Pattern* corresponds to the principle to decompose a complex class into several related classes. A highly cohesive design makes reuse and maintenance more flexible.

Law 3 (**High Cohesion**) *Assume that there are two methods $m_1()$ and $m_2()$n a class M and m_1 does not depend on m_2 (though $m_2()$ may call $m_1()$), we can decompose the class into three associated classes so that the original class M only delegates the functionalities to the newly introduced classes. There are two ways of doing this, as shown in Figure 4.*

The case (a) in Pattern 3 requires M to be coupled with both M_1 and M_2; and in case (b) M is only coupled with M_2, but more interactions are needed between M_2 and M_1.

The other design patterns in [16], such as *Adaptor Pattern*, *Observer Pattern*, *Strategy Pattern* and *Abstract Factory Pattern* can also be formalized.

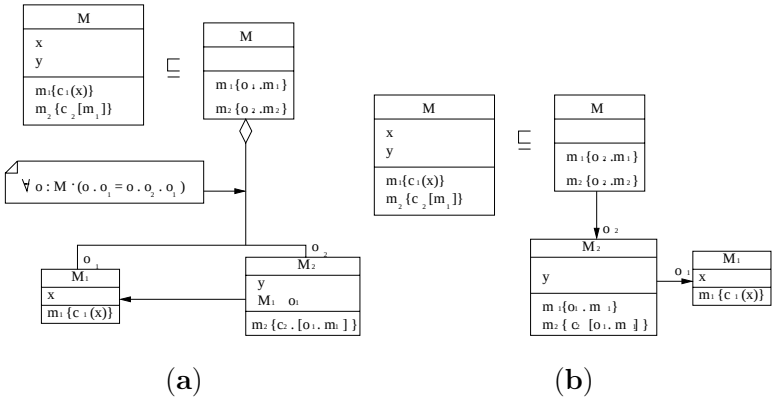

Fig. 4. High Cohesion pattern

In fact the laws above are also reversible and thus can be used for re-engineering. This also implies the result in [5] that every object-oriented program can be converted back to a normal form specification corresponding to an imperative program. Moreover, such a normal form in our framework corresponds to the requirement specification in terms of use cases [23,29]. In [22,27], the five GRASP patterns are systematically used for the development of a case stsudy.

3 Modelling Component Systems

Within rCOS , a component system S is modelled as family $\mathcal{C}$ of *components*. Each component C is specified as a tuple $\langle St, PI, Comp, Prot, RI, RProt \rangle$ where

- St is a set typed *state variables* and a type can be a class. St contains a control variable *wait* for synchronization.
- PI is a set of methods called the *provided interface* of the component, each method is of the form $m(in : U, out : V)$ with an input parameter in and an output parameter out.
- $Comp$ is a mapping that associate each method m in PI with a specification of the required functionality of m, and when m is called by its environment it is executed according to $Comp(m)$ to change the state St and to provide output via out. Each specification $Comp(m)$ is written in a well defined s language that allows specifications at different level of abstraction. At the highest level of abstraction, each $Comp(m)$ is a *design* in UTP [18] of the

form $(\alpha,\ P)$ where,

$$\begin{aligned}
\alpha &\stackrel{def}{=} in\alpha \cup out\alpha \\
in\alpha &\stackrel{def}{=} \{in : U\} \cup St \\
out\alpha &\stackrel{def}{=} \{out' : V\} \cup St' \\
p \vdash R &\stackrel{def}{=} (ok \wedge p(x)) \Rightarrow (ok' \wedge R)
\end{aligned}$$

Also a design may be *guarded* in the form $g\&(p \vdash R)$, where is a Boolean expression, that is defined as

$$g \wedge (p \vdash R) \vee \neg g \wedge (true \vdash wait')$$

- *Prot* is a set of sequences of method invocations of the form

  $$?m_1(in_1); \cdots ; ?m_k(in_k)$$

  where $m_i \in PI$, to specify the protocol between the component with its environment.
- RI is a set of methods that are required by the component. When executing methods in PI. When $Comp(m)$ is a design for each $m \in PI$. The component is independent of RI. Therefore, only when $Comp$ assigns some methods in PI to specifications that contain calls to methods that are not in PI, RI is the set of these *called methods*. Then for each given specification mapping $SpecRI$ gives each method in RI a design, we can recursively calculate $Comp(m)$ as a design.
- *RProt* is a set of method calls of the form

  $$!m_1(out_1); \cdots ; !m_k(out_k)$$

  where $m_i \in RI$, to specify the possible order in which the required methods may be called by the component.

Note that

- *ok* are *wait* are local to the component being concerned. So when the guard of a method does not hold when it is called, the calling component is suspended, but the owning component of the operation can still accept calls to other operations.
- *ok* and *wait* are not programming variables and will not be used in any code of an operation.
- At the level of functional requirement specification, $Comp(m)$ is a design and thus does not depend on any other services.
- At lower levels of abstraction, $Comp(m)$ contains programming statements that may even include calls to methods, called *required services*, that are not declared in PI.

All these *required methods* of a component form the *required interface*. For

a given specification of the required operations, we can calculate $Comp(m)$ as a design. Therefore, a component is a higher order logical formula at low level specification.

We can see that the interfaces of the components describe the UML component diagram of the system, $Comp$ provides a specification of the statechart of the component, and specification of $Comp$ at low level design or implementation can formalize the interaction diagrams among the components, and protocols corresponds to high level interaction diagrams (at the system level).

When $Comp(m)$ are designs for all $m \in PI$, they allow us to calculate the sequences of interactions of the component with its environment that lead the component to a *divergent state* (or livelock state). From them, we can also calculate the set of refusals, i.e. the set of pairs (s, X) where s is a sequence of interactions between the component with its environment and X is a set of methods which the component my refuse to respond after it has engaged in the sequence s of events. With the set of refusals of a component, we can define the condition of deadlock for the component.

The different parts of a component specification are *consistent* if it will never enter a deadlock state if its environment follows $Prot$ when it calls the provided services and will react to any call to the required methods in RI. The components of a component system are *consistent* if the composed system does not deadlock, i.e. the protocols agree with each other, and the specification of the provided methods meets those required.

A component $C_1 = \langle St, PI_1, Comp_1, Prot_1, RI, RProt_1 \rangle$ is a refinement of component $C = \langle St, PI, Comp, Prot, RI, RProt \rangle$ if

(i) C_1 provides at least as many as services as C, $PI \subseteq PI_1$,

(ii) any sequence of call acceptable C is acceptable by C_1, $Prot \subseteq Prot_1$,

(iii) any environment that can provide services to C well should also provide services to C_1 well, $RProt_1 \subseteq RProc$.

We have refinement rules of five categories:

(i) Refine private or provided operations of a component

(ii) Add public or private methods. This is useful for incremental and iterative development

(iii) Delegation of tasks methods to other methods. This formalises the expert pattern in [22], but in a component-based style.

(iv) Refine the provided protocol by providing more choices of sequences

(v) Decompose complex components into composition of simpler and more cohesive components.

This are essential for scaling up the method, software to be high cohesive, easy to maintain and reuse

4 Object-Orientation and Component-Based Development

In most books on component-based design in the UML framework, e.g. [10,36], components are taken as a family of collaborating objects (or class at the level of templates or styles) without being formally defined. Some papers, e.g. [6,33,1], are critical to object-orientation and think that objects or classes are not composable and thus cannot be treated as objects. To some extend, this true as objects or classes do not specify their required interfaces. On the other hand, all the existing component technologies, such as JavaBeans, EJB, and .COM, are based on object-oriented methods.

In our framework, we can take a class and translate it to primitive objects easily by calculating the required methods from the code of the class methods. However, in general, a component in our proposed model can be realized by a family of collaborating classes. Therefore, for a component C, we treat the interface methods of C and the protocol as the specification of a the use cases of the component and the components in environment of C as the actors of these use cases. The type of state variables of C may be a class and all the classes and their relationships form a class diagram. The design and implementation of this component can then be carried in a UML-based object-oriented framework.

5 Conclusion

We have proposed a classical relational model (rCOS) for component-based and objectoriented development. This model provides a smooth link between component-based design and object-oriented development. This model supports rigorous application of UML in an iterative and incremental development process (RUP). The formalism is based on the design calculus in Hoare and He's Unifying Theories of Programming [18] In a top-down process, model provides the fundamental basis for Model Driven Development. If we take a bottom-up approach, it supports re-engineering. Our message is: to support programming in the large,

- we need a multi-view modelling approach,
- a multi-notational modelling language is of a great advantage (though not everyone has to use UML),

- consistent refinement of different views is important
- different verification techniques may be applied to refinement of different views.

In the framework of ROOL [7], Borba, *et al*, also investigate refinement of object systems in [5]. Although, ROOL and rCOS share a number of common refinement laws, rCOS supports more features, such as references, and enjoys more refinement laws than ROOL. In our related works, general transition systems are introduced to provide an integrated model for conceptual class diagrams and use cases (without the treatment of sequence diagrams, state machines and use-case diagrams) [31]. rCOS is used in [29] for the specification UML models of requirements, but a requirement model there only consists of a conceptual class diagram and a use-case model directly specified by rCOS . Article [25] uses rCOS for the specification of design class diagrams and sequence diagrams, but without rules for model transformation. A tool for requirement analysis has been developed using this framework [26]. Algorithms are also designed for consistency checking and executable code generation from a system model [32].

Future work includes the completion of the calculus rCOS for component systems, and test it with some case studies. We are also interested in a theory of tool integration within this framework.

References

[1] N. Aguirre and T. Maibaum. A temporal logic approach to component-based system specification and verification. In *Proc. ICSE'02*, 2002.

[2] P. Andre, A. Romanczuk, J.-C. Royer, and A. Vasconcelos. Checking the consistency of UML class diagrams using Larch Prover. In *Proc. ROOM'2000, York, UK*, 2000.

[3] E. Astesiano and G. Reggio. An attempt at analysing the consistency problems in the UML from a classical algebraic viewpoint. In *Proc. WADT 2002, LNCS 2755*. Springer Verlag, 2003.

[4] 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.

[5] P. Borba, A. Sampaio, and M. Cornélio. A refinment algebra for object-oriented programming. In L. Cardelli, editor, *Proc. ECOOP03, LNCS2743*, pages 457–482. Springer, 2003.

[6] M. Broy. Object-oriented programming and software development - a critical assessment. In A. McIver and C. Morgan, editors, *Programming Methodology*. Springer, 2003.

[7] A. Cavalcanti and D.A. Naumann. A weakest precondition semantics for an object-oriented language of refinement. Technical Report CS Report 9903, Stevens Institute of Technology, Hoboken, NJ 07030, February 2000.

[8] J. Davis and J.P. Woodcock. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.

[9] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program semantics*. Springer, 1989.

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

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

[12] G. Engels, J.M. Kuester, and L. Groenewegen. Consistent interaction of software components. In *Pro. of IDPT2002*, 2002.

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

[14] A. Evans, *et al.* Developing the UML as a formal modelling notation. In *Proc. UML'98, LNCS 1618*. Springer-Verlag, 1998.

[15] 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, LNCS 2863*, San Fancisco, CA, USA, 2003. Springer.

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

[17] J. He, Z. Liu, X. Li, and S. Qin. A relational model of object oriented programs. In *Proceedings of the Second ASIAN Symposium on Programming Languages and Systems (APLAS04), Lecture Notes in Computer Science 3302*, pages 415–436, Taiwan, March 2004. Springer.

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

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

[20] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

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

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

[23] X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *COMPSAC01*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.

[24] X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagrams. In *Proceedings of ASWEC2004*, pages 168–177, Melbourne, Australia, 2004. IEEE Computer Society.

[25] X. Li, Z. Liu, J. He, and Q. Long. Generating prototypes from a UML model of requirements. In *International Conference on Distributed Computing and Internet Technology (ICDIT2004), Lecture Notes in Computer Science 3347*, pages 215–225, Bhubaneswar, India, 2004. Springer.

[26] J. Liu, Z. Liu, J. He, and X. Li. Linking UML models of design and requirement. In *Proceedings of ASWEC2004*, pages 329–338, Melbourne, Australia, 2004. IEEE Computer Society.

[27] Z. Liu. Object-oriented software development in UML. Technical Report UNU/IIST Report No. 228, UNU/IIST, P.O. Box 3058, Macau, SAR, P.R. China, March 2001.

[28] Z. Liu, J. He, and X. Li. Contract-oriented development of component systems. In *Proceedings of IFIP WCC-TCS2004*, pages 349–366, Toulouse, France, 2004. Kulwer Academic Publishers.

[29] 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, LNCS 2885*, pages 641–664. Springer, 2003.

[30] Z. Liu, J. He, X. Li, and J. Liu. Unifying views of UML. *Electronic Notes of Theoretical Computer Science (ENTCS)*, 101C:95–127, 2004.

[31] Z. Liu, X. Li, and J. He. Using transition systems to unify UML models. In C. george and H. Miao, editors, *Proc. of International Conference on Formal Engineering Methods (ICFEM 2002), Lecture Notes in Computer Science 2495*, pages 535–547, 2002.

[32] Q. Long, Z. Liu, X. Li, and J. He. Consistent code generation from UML models. In *Pro. of Australian Software Engineering Conference (ASWEC'2005)*, Brisbane, Australia, 2005. IEEE Computer Sciety.

[33] F. Luders and K.K. Lau. Specification of software components. In I. Crnkovik and M. Larsson, editors, *Building Reliable Component-Based Software Systems*, pages 23–38. Artech House, 2002.

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

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

[36] R. Pooley and P. Steven. *Using UML: Software Engineering with Objects and Component*. Addison-Wesley, 1999.

[37] 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.

[38] J. Yang, Q. Long, Z. Liu, and X. Li. A predicative semantic model for integrating UML models. In *Proceedings of 1st International Colloquium on Theoretical Aspects of Computing (ICTAC), Lecture Notes in Computer Science 3407*, Guiyang, China, 2004. Springer.