rCOS: Refinement of Component and Object Systems*

Zhiming Liu[1], He Jifeng[1,**], and Xiaoshan Li[2]

[1] International Institute for Software Technology,
United Nations University, Macao SAR, China
{lzm, hjf}@iist.unu.edu
[2] Faculty of Science and Technology,
University of Macau, Macao SAR, China
xsl@umac.mo

Abstract. We present a model of object-oriented and component-based refinement. For object-orientation, the model is *class-based* and refinement is about *correct* changes in the structure, methods of classes and the main program, rather than changes in the behaviour of individual objects. This allows us to prove refinement laws for both high level design patterns and low level refactoring. For component-based development, we focus on the separation of concerns of *interface* and *functional contracts*, leaving refinement of *interaction protocols* in future work. The model supports the specification of these aspects at different levels of abstractions and their consistency.

Based on the semantics, we also provide a general definitional approach to defining different relational semantic models with different features and constraints.

Keywords: Object-Orientation, Component-Based Development, Refinement, Specification, Consistency.

1 Introduction

Today's software engineering is mainly concerned with systematic development of large and complex systems. To cope with the scale of the problem traditional software engineers divide the problem along three axes:

1. Along the temporal axis the development activities are divided into three stages of requirements specification, design and implementation.
2. Different activities in each stage deal with different aspects of the system. Requirement analysis is split into specification of aspects of static data structure, control flows or processes and operations or services. Similarly, design may be split into design strategies for concurrency, design strategies for efficiency and design strategies for security. These strategies are commonly expressed as design patterns [13]. Finally implementation may be split into databases, user interfaces and libraries for security.

* This is a revised and extended version of the combination of the papers [17,31]. This work is partly supported by e-Macao project funded by the Government of Macao, and the research grant 02104 MoE and the 973 project 2002CB312000 of MoST of P.R. China.

** On leave from East China Normal University, Shanghai, China.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 183–221, 2005.

3. The third axis is that of system evolution and maintenance [20,24] where each evolutionary or maintenance step enhances the system by iterating through the requirements to implementation cycle.

Unfortunately in practical software engineering, all aspects are specified using informal techniques and therefore this approach does not give the desired assurances and productivity. The main problems are, among others, the following:

- Since the requirements specification is informal there is no way to ascertain its completeness resulting in a lot of gaps.
- The gaps in requirements are filled by ad-hoc decisions taken by programmers who are not qualified for the job of requirement analysis. This results in code of poor quality.
- There is no traceability between requirements and implementation making it very expensive to accommodate changes and maintain the system.
- Most of the tools are for project management and system testing. Although these are useful, they are not enough for ensuring the semantic correctness of the implementation for a requirements specification and semantic consistency of changes made in the system.

Formal methods, on the other hand, attempt to complement the informal engineering methods by techniques for formal modelling, specification, verification and refinement [46,15]. Formal methods were born and has grown up in those years of structured analysis and design. So we have theories of formal specification, verification, refinement, decomposition and composition. In principle, a formal system development starts with an abstract specification and transforms it into a program through a number of refinement steps. A formal method is supported by a sound logical framework. These have helped in improving the quality of software systems so that they are more correct and safer to use. However, they do not yet support the three dimensional development process well. It is still a great challenge to scale up formal methods to industry scale because of the problems listed below.

- Formal methods have inherited the same disadvantages from the one-dimensional "water-fall model" of development activities. They suffer even more seriously from those disadvantages as a specification of the whole system at any level, e.g. the requirement level, in a formal notation is not understandable to most system engineers, not to mention about formal verification.
- Because of the theoretical goal of completeness and independence, refinement calculi, including those for object-oriented programming [22,23,1,8], provide only refinement rules for a small change in each step. Refinement calculi can therefore be difficult to be used in practice. Data refinement always requires definition of a semantic relation between the programs (their state space) and is hard to be applied systematically and to deal with "big-step refinement".
- There is no clear separation of concerns making it difficult for domain experts, architects and programmers to collaborate towards a single solution. The existing object-oriented models, e.g. [22,23,1,8], focus on much programming aspects and it is not clear what kind of properties of an object-oriented program can be described

and proven in such a model. Therefore, these models cannot be used for software development as we do not know from what a specification that the system is to be developed or refined.
- It is not easy for software engineers to build correct and proper models, from low level designs or implementations, that can be verified by model checking tools.
- There is no explicit support for productivity enhancing techniques such as component based development.

So far, both the formal methods and the methods adopted by practical software engineers are far from meeting the quality and productivity needs of the industry. The industry continues to be plagued by high development and maintenance costs and poor quality. However, recently there have been encouraging developments in both approaches. The software engineering community has started using precise models for early requirements analysis and design [37,12]. Theories and methods for object-oriented, component-based and aspect-oriented modelling and development are gaining attention of the formal methods community[1]. There are various attempts at investigating formal aspects of object-oriented refinement, design patterns, refactoring and coordination [5,17,31].

In this article, we present a calculus of **R**efinement of **C**omponent and **O**bject **S**ystems (rCOS)[2]. It captures the essential features of object-orientation including *reference types, inheritance, dynamic binding, and visibility*. Unlike the object logic in [1], rCOS is *class-based* and refinement is about *correct* changes in the structure, methods of classes and the main program, rather than changes in the behaviour of individual objects. For component-based development, we are concerned with the separation of concerns of *interfaces*, *functional contracts* and *interaction protocols*. In this paper, we focus on *interfaces* and *functional contracts*. rCOS reflects the basic object-oriented and component-based principles of *component and class decomposition*, *task delegation* and *data encapsulation*. It allows refinement for both high level *design patterns* and low level *refactoring* [34].

We briefly introduce in Section 2, as our semantic basis, the notion of *designs* in Unifying Theories of Programming [19]. In Section 3, we define the model for object systems. We present refinement calculus of *object-oriented designs* in Section 4.1. We then show in Section 5 how the model for object systems is extended to deal with component systems. In Section 6, we conclude the article with discussion and related work.

2 Semantic Basis

We take a classical approach to modelling the execution of a program in terms of a relation between the *states* of the program. However, the concept of state is more general than what programmers usually understand and it depends on what the modeler

[1] In addition to the well-established conferences such OOPSLA and ECOOP, many new conferences and workshops on component systems, object systems and aspect-oriented techniques.

[2] rCOS is produced by LaTeX command {\large r}\textsc{COS}. In [17], the calculus was named as OOL.

wants to observe of the execution of a program. For example, for a terminating sequential program, we are only interested in the initial inputs and final outputs. For a possible non-terminating program, we need an observable by which we can describe if the program terminates for some inputs. For concurrent and communicating program, we would like to observe the possible *traces* of interactions, *divergencies* and *refusals*, in order to verify if program is deadlock free and livelock free. If we are interested in real-time programs, we need to observe the time. Identification of what to observe in different kinds of systems is one of the core ideas of the Unifying Theories of Programming [19].

We call what to be observed of a program *P* the *observables* or *alphabet* of the program, denoted by $\alpha(P)$ and simply α when there is no confusion. An observable of *P* may take different values for different executions or runs, but from the same value space called the *type* of the observable. Therefore, an observable is also a *variable*. Though not all observables have to appear in a program text, but they are all needed to define the semantics of the program.

Given an alphabet α, a *state* of α is a (well-typed) mapping from α to the value spaces of the observables. A program *P* with an alphabet α is then defined as a pair of predicates, called a *design* and represented as $Pre \vdash Post$, with free variables in α. It means that if the value of observables satisfies the *precondition Pre* at the beginning of the execution, the execution will *generate* observables satisfying the *postcondition Post*, and thus defined as the implication

$$(Pre \vdash Post) \stackrel{def}{=} Pre \Rightarrow Post$$

2.1 Programs as Designs

This subsection briefly shows how the basic programming constructs can be defined as designs. For details, we refer the reader to [19].

For an imperative sequential program, we are interested in observing the values of the input variables $in\alpha$ and output variables $out\alpha$. Here we take the convention that for each input variable $x \in in\alpha$, its primed version x' is in an output variable in $out\alpha$, that gives the final value of x after the execution of the program. We use a Boolean variable *ok* to denote whether a program is *started properly* and its primed version ok' to represent whether the execution has terminated. The alphabet α is defined as the union $in\alpha \cup out\alpha \cup \{ok, ok'\}$, and a design is of the form

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

where

- p is a predicate over $in\alpha$ and R is a predicate over $out\alpha$,
- p is the *precondition*, defining the initial states
- R is the *postcondition*, relating the initial states to the final states.
- ok and ok': describe the termination, they do **not** appear in expressions or assignments of program texts

The design represents a *contract* between the "user" and the program such that if the program is started properly in a state satisfying the precondition it will terminate in a state satisfying the postcondition R.

A design is often *framed* in the form

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

where $\underline{w}$ contains all the variables in $in\alpha$ but those in β.

Before we define the semantics of a program, we first define some operations on designs.

- Given two designs such that the output alphabet of P is the same as primed version of the input alphabet of Q, the sequential composition

$$P(in\alpha_1, out\alpha_1); Q(in\alpha_2, out\alpha_2) \stackrel{def}{=} \exists m \cdot P(in\alpha_1, m) \wedge Q(m, out\alpha_2)$$

- Conditional choice: $(D_1 \lhd b \rhd D_2) \stackrel{def}{=} (b \wedge D_1) \vee (\neg b \wedge D_2)$
- Demonic and angelic choice operators:

$$D_1 \sqcap D_2 \stackrel{def}{=} D_1 \vee D_2 \quad D_1 \sqcup D_2 \stackrel{def}{=} D_1 \wedge D_2$$

- `while` b `do` D is defined as the weakest fixed point

$$X = ((D; X) \lhd b \rhd \mathit{skip})$$

We can now define the meaning of primitive commands program commands as framed designs in Table 1. Composite statements are then defined by the operations on designs.

Table 1. Basic commands as designs

command: c	**design: $[\![c]\!]$**	**description**
skip	$\{\} : \mathit{true} \vdash \mathit{true}$	does not change anything, but terminates
chaos	$\{\} : \mathit{false} \vdash \mathit{true}$	any thing, including non-terminating, can happen
$x := e$	$\{x\} : true \vdash x' = val(e)$	side-effect free assignment; updates x with the value of e
$m(e; v)$	$[\![$`var` $in, out]\!]$; $[\![in{:=}e]\!]$; $[\![body(m)]\!]$; $[\![v{:=}out]\!]$; $[\![$`end` $in,\ out]\!]$	$m(in; out)$ is the signature with input parameters in and output parameters out; $body(m)$ is the body command of the procedure/method

In general, when defining a particular programming language, the preconditions are usually strengthen with some *well-definedness* conditions of the commands, and a program or command c is generally of the form

$$[\![c]\!] \stackrel{def}{=} D(c) \Rightarrow \mathit{Spec}$$

where $Spec$ is a design. Some of the well definedness conditions may even be dynamic.

Strengthening precondition with well-definedness conditions allows us to treat correcting a unwell-defined command to a well-formed one as refinement. This is essential to support incremental and iterative development as most cases of unwell-defined are due to the insufficiency of data or services. Therefore, adding more data, services and components, without altering the existing ones, will be refinement in our framework.

In this article, we will add variables about dynamic typing, visibility, etc, to define object-oriented programs. This ensures that the logic of rCOS is a conservative extension to that for imperative programs. Therefore, all the laws about imperative commands will remain valid without the need of reproving. rCOS can be further extended to deal with features of communication, interaction, real-time and resources. If we adding variables for traces, refusals and divergencies into the alphabet, different kinds of semantics of communicating processes can be defined as designs [19,10]. Also, using clock variables in the alphabet, we can define real-time programs as designs too [43]. It is possible to further extend rCOS to describe resource consumptions, such as memory and processor nodes, by introducing resource variables [21].

2.2 Refinement of Designs

The refinement relation between designs is then defined to be logical implication. A 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 if

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

where $x, x', \ldots, z, z'$ are variables contained in α. We write $D_1 = D_2$ if they refine each other.

If they do not have the same alphabet, we can use data refinement. Let ρ be a mapping 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 $(\rho; P_1) \sqsubseteq (P_2; \rho)$. It is easy to prove that *chaos* is the worst program, i.e. $chaos \sqsubseteq P$ for any program P. For more algebraic laws of imperative programs, please see [19].

The following theorem is the basis for the fact that the notion of designs can be used for defining a semantics of programs.

Theorem 1. *The notion of designs is closed under programming constructors:*

$$((p_1 \vdash R_1); (p_2 \vdash R_2)) = ((p_1 \wedge \neg(R_1; \neg p_2)) \vdash (R_1; R_2))$$
$$(p_1 \vdash R_1) \sqcap (p_2 \vdash R_2) = (p_1 \wedge p_2) \vdash (R_1 \vee R_2)$$
$$(p_1 \vdash R_1) \sqcup (p_2 \vdash R_2) = (p_1 \vee p_2) \vdash ((p_1 \Rightarrow R_1) \wedge (p_2 \Rightarrow R_2))$$
$$((p_1 \vdash R_1) \lhd b \rhd (p_2 \vdash R_2)) = ((p_1 \lhd b \rhd p_2)) \vdash (R_1 \lhd b \rhd R_2)$$

3 Object Systems

In this section we introduce to the syntax and semantics of rCOS for object systems.

3.1 Syntax

In rCOS, an object system (or program) *S* is of the form *Cdecls* • *Main*, consisting of class declaration section *Cdecls* and a main method *Main*. The main method is a pair (glb, c) of a finite set *glb* of *global variables declaration* and a command c. 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

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

Note that

- Each part in the body of the declaration is optional too.
- A class can be declared as *private* or *public*, but by default it is assumed to be *public*. We can understand the class section as a *Java-like package* and *Main* as an application program using the package. Only a public class or a primitive type can be used in the global variable declarations *glb* of *Main*. Later in Section 4.1, *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 behaviour of the system. Refinement is also allowed 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). 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 kinds 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 are public and can be inherited by a subclass.

Symbols

To make the presentation precise we assume the following disjoint infinite sets of symbols,

- *VNAME* denotes the set of symbols of variables names and we use *x*, *y*, and *z* and their versions with subscripts when we talk about arbitrary variables.
- *CNAME* is used for the set of class names. We use *C*, *D*, M and N with possible subscripts to range over this set.
- *ANAME* is the set of symbols to be used as names of attributes, ranged over by *a* with possible subscripts.

Commands

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

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

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. Unlike [41] 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*. Expressions $\underline{e}$, $\underline{v}$ and $\underline{u}$ are the actual value input parameters result parameters and actual value-result parameters, respectively. They can be changed during the execution of the method call and with final output returned in the actual result and value-result parameters. The command $C.new(x)$ is to create a new object of class *C* with the initial values of its attributes as declared in *C* and assign it to variable x. Thus, $C.new(x)$ uses x with type *C* to store the newly created object.

Expressions

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

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

where *null* represents the special object of the special class *NULL* and has *null* as its unique object, $e.a$ is the a-attribute of e, and $(C)e$ is type casting. Notice that we do not define *NULL* as a subclass of any other class as we do not allow multiple inheritance.

3.2 Semantics

rCOS adopts an observation-oriented and relational semantics. 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 on 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.
- The dynamic type *M* of an object can be casted up to any superclass *N* and later casted down to any class which is a subclass of *N* and a superclass of *M* or *M* itself. We therefore need to record both the casted type *N* and the dynamic type M of the object.

Static Semantics. The class declaration section *Cdecls* of a program defines the types (value space) and static structure of the program:

- *pricname*: the set $\{C \mid C \text{ is declared in } Cdecls\}$ of the private class names declared in *Cdecls*. We also use *pubcname* to record the sets of names of the public classes in declared in *Cdecls*. Let *cname* be the union of these two sets.
- *superclass*: the partial function $\{M \mapsto N \mid \text{Class } M \text{ extends } N \text{ is declared in } Cdecls\}$, recording that N is a direct superclass of M. We define the general superclass class relation $\succ$ as transitive closure of *superclass*, and $N \succeq M$ if $N \succ M$ or $N = M$.
- *pri*, *prot*, and *pub*: they associate each class name $C \in cname$ to its private attributes $pri(C)$, protected attributes $prot(C)$, and public attributes $pub(C)$, respectively:

$$\begin{aligned} pri(C) &\stackrel{def}{=} \{\langle a : T, d\rangle \mid T\ a = d \text{ is (declared as) a private attribute of } C\} \\ prot(C) &\stackrel{def}{=} \{\langle a : T, d\rangle \mid T\ a = d \text{ is a protected attribute of } D \succeq C \text{ for some } D \in cname\} \\ pub(C) &\stackrel{def}{=} \{\langle a : T, d\rangle \mid T\ a = d \text{ is a public attribute of } D \succeq C \text{ for some } D \in cname\} \end{aligned}$$

- *op*: it associates each class $C \in cname$ to its set of methods $(op)(C)$

$$\begin{aligned} (op)(C) \stackrel{def}{=} \{&m \mapsto (\underline{x} : \underline{T}_1, \underline{y} : \underline{T}_2, \underline{z} : \underline{T}_3, c) \mid \\ &m(\underline{x} : \underline{T}_1; \underline{y} : \underline{T}_2; \underline{z} : \underline{T}_3)\{c\} \text{ is declared as method of } C\} \end{aligned}$$

We define the following notations

1. The function *attr* is the union of *pri*, *prot* and *pub*; for each C, $attr(C)$ is the set of attributes declared in C itself.
2. The function *Attr* extends $attr(C)$ for each C to include the protected and public attributes that C inherited from its super classes, i.e. $Attr(C)$ contains all attributes directly accessible in methods of C.
3. The function *AAttr* extends $attr(C)$ for each C to include all attributes of C and those of it superclasses. Thus, $AAttr(C)$ determines the whole state space of an object of class C and when an object C is created, all attributes in $AAttr(C)$ need to be initialized.
4. $init(C.a)$ denotes the initial value of attribute a of C.
5. $dtype(C.a)$ denotes the *declared type* T if $\langle a : T, d\rangle \in AAttr(C)$. This is used to calculate the declared type of an *attribute expression* in C inductively:
   (a) $dtype(a) \stackrel{def}{=} dtype(C.a)$
   (b) $dtype(e.a) \stackrel{def}{=} dtype(dtype(e).a)$

We call the tuple $\langle cname, superclass, (pri, prot, pub), op\rangle$ a *program structure*, denoted by Ω_{Cdecls}. We take the whole declaration section as a command which sets up the structure:

$$[\![Cdecls]\!] \stackrel{def}{=} \{\Omega_{Cdecls}\} : true \vdash \Omega'_{Cdecls} = \langle cname, superclass, (pri, prot, pub), op\rangle$$

Definition 1. *A class declaration section Cdecls is* **well-defined**, *denoted* $\mathcal{D}(Cdecls)$, *if the following conditions hold*

1. *each class name* $M \in cname$ *and the name of its direct superclass* N *are distinct,*
2. *if* $M \in cname$ *and* $superclass(M) = N$, *then* $N \in cname$,

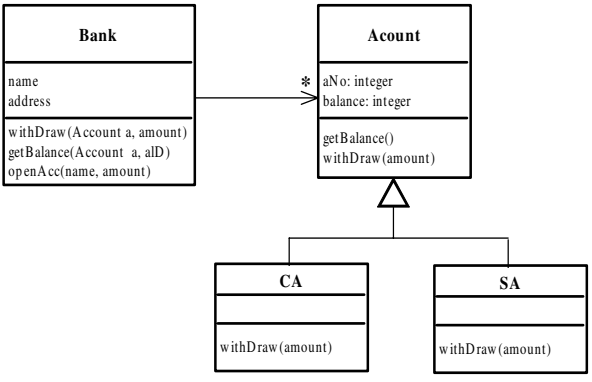

Fig. 1. A bank system

3. *any type used in declarations of attributes and parameters is either a primitive built-in type or a class in cname,*
4. *the superclass relation $\succ$ is acyclic,*
5. *any attribute of a class is not redeclared in its subclasses, i.e. we do not allow attribute hiding and alias in a subclass*[3]*,*
6. *the names of the attributes of each class are distinct,*
7. *the names of the methods of each class and the names of parameters of each methods are distinct respectively.*

A well-defined declaration section corresponds to a UML [4] class diagram, and thus it and its semantics can be used for formalisation of UML class diagrams, such as the one in Figure 1. For related work on formal support to UML-based development, we refer to our work in [32,33,47].

Type, Values and Objects. We assume a set $\mathcal{T}$ of *built-in primitive types*. We also assume an infinite set *REF* of *object identities* (or *references*), and *null* $\in$ *REF*. A *value* is either a member of a primitive type in $\mathcal{T}$ or an object identity in *REF* with its *dynamic typing information*. Let the set of values be

$$VAL \stackrel{def}{=} \bigcup \mathcal{T} \cup (REF \times CNAME)$$

For a value $v = \langle r, C \rangle \in REF \times CNAME$, we use *ref*$(v)$ to denote r and *type*(v) to denote C.

Definition 2. *An* **object** o *is either the special object null, or a structure* $\langle r, C, \sigma \rangle$*, where*

- *reference* r*, denoted by ref*(o)*, is in REF,*
- C*, denoted by type*(o)*, is a class names.*
- σ *is called the* state *of* o*, denoted by state*(o)*, and it is a mapping that assigns each* $a \in AAttr(C)$ *to a value in dtype*(a) *if dtype*$(a) \in \mathcal{T}$ *and otherwise to the null object or a value in REF* $\times$ *CNAME. We use* $o.a$ *to denote* $\sigma(a)$

[3] If we allow attribute hiding and alias, we have to introduce special object variables $this$ and $super$. We not consider this problem in this paper.

We extend the *equality* relation on values to the relation on both values and objects

$$(v_1 = v_2) \stackrel{def}{=} \begin{pmatrix} (type(v_1) = type(v_2) \wedge type(v_1) \in \mathcal{T} \wedge (v_1 = v_2)) \vee \\ \forall a \in AAttr(type(v_1)) \cdot (v_1.a = v_2.a) \end{pmatrix}$$

Notice that this equality ignores the references of objects, but only concerns about the structure and the primitive attributes of the objects in the structure.

Some Notations. Let $\mathcal{O}$ be the set of all objects, including *null*. The following notations will be employed in the semantics definitions.

- For a non-empty finite sequence of elements $\underline{s} = \langle s_1, .., s_k \rangle$, we define the head element $head(\underline{s}) = s_1$, and the tail sequence $tail(\underline{s}) = \langle s_2, .., s_k \rangle$.
- For sets S and S_1, $S_1 \rhd S$ is the set difference removing elements in S_1 from S. Let $\rhd$ have higher associativity than the normal set operators like $\cup$ and $\cap$.
- For a mapping $f : D \longrightarrow E$, $d \in D$ and $r \in E$,

$$f \oplus \{d \mapsto r\} \stackrel{def}{=} f' \qquad \text{where } f'(b) \stackrel{def}{=} \begin{cases} r, \text{ if } b = d; \\ f(b), \text{ if } b \in \{d\} \rhd D. \end{cases}$$

- For an object $o = \langle r, M, \sigma \rangle$, an attribute a of M and a value d,

$$o \oplus \{a \mapsto d\} \stackrel{def}{=} \langle r, M, \sigma \oplus \{a \mapsto d\} \rangle$$

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

$$S \uplus \{\langle r, M, \sigma \rangle\} \stackrel{def}{=} \{o \mid ref(o) = r\} \rhd S \cup \{\langle r, M, \sigma \rangle\}$$
$$ref(S) \stackrel{def}{=} \{r \mid r = ref(o), o \in S\}$$

For a given class declaration section *Cdecls*, we use Σ_{Cdecls} to denote the set of all objects of the classes declared in *Cdecls*, called the *object space* of *Cdecls*. Σ_{Cdecls} corresponds to the set of all UML *object diagrams* [4] of the UML class diagram of *Cdecls* [32]. We call the pair $(\Omega_{Cdecls}, \Sigma_{Cdecls})$ a *program context* and denote it by Ξ_{Cdecls}. When there is no confusion, we omit the subscript *Cdecls* from these notations. All the dynamic semantic definitions in the rest of this section are given under a fixed context, that is defined by a given class declaration section. Therefore the evaluation $value(e)$ of an expression e is carried out in the context Ξ and the semantics $[\![c]\!]_\Xi$ defines the state change by the execution of c in the context Ξ.

Dynamic Semantics. In rCOS, we define the behavior of an object program by a design over a set of observables or state variables. We first identify the state variables and define their states.

Variables. Now we look at what variables can be changed during the execution of the program.

System Configuration. First, we introduce a variable Π whose value is the set of objects created so far. We call Π the *current configuration* of the program in [41]. During the execution of the program, the value of Π is set in the powerset 2^{Σ} that satisfies the following conditions:

1. *objects in Π are complete*: if $o \in \Pi$ and $a \in \mathit{AAttr}(\mathit{type}(o))$ with a class type, then $o.a$ is either *null* or there is an object $o_1 \in \Pi$ and $\mathit{ref}(o.a) = \mathit{ref}(o_1)$, and
2. *Objects are uniquely identified by their references:* for any objects o_1 and o_2 in Π if $\mathit{ref}(o_1) = \mathit{ref}(o_2)$ then
   (a) $\mathit{type}(o_1) = \mathit{type}(o_2)$, and
   (b) $\mathit{ref}(\mathit{state}(o_1)) = \mathit{ref}(\mathit{state}(o_2))$, where for each $a : T \in \mathit{AAttr}(\mathit{type}(o))$
   $$\mathit{ref}(\mathit{state}(o))(a) \stackrel{def}{=} \begin{cases} \mathit{ref}(o.a) & \text{if } T \in \mathit{cname} \\ o.a & \text{if } T \in \mathcal{T} \end{cases}$$

When a new object is created or the value of an attribute of an existing object is modified, the system configuration Π will be changed. For each class C, we use variable $\Pi(C)$ to denote the set of existing objects of class C.

External Variables. A set $\mathit{glb} = \{x_1 : T_1, \ldots, x_k : T_k\}$ of variables with their types are declared in the main method of the program, where each type T_i is called the *declared type* of x_i, denoted as $\mathit{dtype}(x_i)$, and it is either a built-in primitive type or a public class in *pubcname*. Their values will be modified by methods and commands of the main method containing them.

Local Variables. A set *localvar* identifies the local variables which occur in the local variable declaration and undeclaration commands. This set includes *self* whose current value represents the current active object, parameters of methods of classes, and other variables introduced by the local declaration command. We assume that *localvar* and *glb* are disjoint.

Because method calls may be nested inside a method body, *self* and a parameter of a method may be declared a number of times with possible different types before it is undeclared. A local variable x has a sequence of declared types and is syntactically represented in the form of $(x : \langle T_1, \ldots, T_n \rangle)$. We use *TypeSeq* to denote the sequence of types of x, and T_1 is the most recently declared type of x and denoted by $\mathit{dtype}(x)$.

We use $\overline{x}$ as a variable to denote the value of a local variable x. This value comprises a finite sequence of values, whose first (head) element, which is simply denoted by x itself, represents the current value of the variable. We use the conventions that $x : \langle T \rangle$ and $\overline{x}$ for x for an external variable $x : T \in \mathit{glb}$.

Visibility. We introduce a variable *visibleattr* to hold the set of attributes which are visible to the command under execution. There the value of *visibleattr* defines the current execution environment. Before executiing a method of an object o, *visibleattr* is set to set $\mathit{Attr}(o)$ of the attributes of the current type of o, including all the declared attributes of the class, the protected and public attributes of its super classes and all public attributes of public classes; and it will be reset to the global environment consisting of all the public attributes of the public classes after the execution of the method. We will define auxiliary commands that set and reset the execution environments when we define

the semantics of a method invocation. Notice that the value space of *visibleattr* is the powerset of $\{C.a \mid C \in CNAME, a \in ANAME\}$.

We use

- *var* to denote the union of *glb* and *localvar*,
- *VAR* to denote the union of *var* plus Π and *visibleattr*, and we call it the set of *dynamic variables*,
- *glb* is the set of elements of *VAR* excluding those out of *glb*,
- for a set *V* of variables, *V'* to denote the set of the primed versions of the variables of *V*.

States. We now define the notion of states in the object-oriented setting.

Definition 3. *For a program $S = Cdecls \bullet Main$, a (***dynamic***) state of S is a mapping Γ from the variables VAR to their value spaces that satisfies the following conditions:*

1. *If $x \in VAR$ and $dtype(x) \in \mathcal{T}$ then $\Gamma(x)$ is a value in $dtype(x)$,*
2. *If $x \in VAR$ and $dtype(x) \in cname$ then $\Gamma(x)$ is*
   (a) *either null, or*
   (b) *a value in $v \in REF \times CNAME$ such that there exists an object $o \in \Gamma(\Pi)$ for which $ref(o) = ref(v)$ and $type(o) \preceq type(v)$.*
   This attachment of an object o to a variable x provides the information about type casting: $type(o)$ is the current *(based) type of x, denoted as $atype(x)$, and $type(v)$ is the* casted type *of x.*

Two states Γ_1 and Γ_2 are equal*, denoted by $\Gamma_1 = \Gamma_2$, if*

1. *$\Gamma_1(x) = \Gamma_2(x)$ for any $x \in VAR$ such that $dtype(x) \in \mathcal{T}$,*
2. *for any $x \in VAR$ and $dtype(x) \in cname$*
   (a) *$\Gamma_1(x) = null$ if and only if $\Gamma_2(x) = null$, and*
   (b) *if $o_i \in \Gamma_i(\Pi)$ and $ref(\Gamma_i(x)) = ref(o_i)$, then $type(\Gamma_1(x)) = type(\Gamma_2(x))$ and $o_1 = o_2$.*

For state Γ and a subset $V \subseteq VAR$, $\Gamma(\Pi{\downarrow_V})$ projects Π onto *V* and it is defined as follows:

1. if $x : C \in V$, $C \in cname$, $o \in \Gamma(\Pi)$ and $ref(\Gamma(x)) = ref(o)$, then $o \in \Gamma(\Pi{\downarrow_V})$
2. if $o \in \Gamma(\Pi{\downarrow_V})$ and a is an attribute of $type(o)$ with a class type, $o_1 \in \Gamma(\Pi)$ and $ref(o.a) = ref(o_1)$, then $o_1 \in \Gamma(\Pi{\downarrow_V})$
3. $\Gamma(\Pi{\downarrow_V})$ only contains objects constructed from $\Gamma(\Pi)$ and the values of the external variables following the above two rules.

In particular, when we restrict a state Γ on the external variables *glb* and projects Π onto these variables, we obtain an *external state* in which all objects in the system configuration are attached to variables. Therefore, the restriction plays the role of *garbage collection*.

For a given state, each expression e, $visible(e)$ is true if and only if one of the following conditions holds:

1. e is a declared simple variable $x \in$ *var*, or
2. $e \equiv$ *self*.a and there exists a class name $N \in$ *cname* such that $N \succeq$ *atype*(*self*) and $N.a \in$ *visibleattr*, or
3. e is of the form $e_1.a$ and e_1 is not *self* such that *visible*(e_1), there exists a $N \succeq$ *type*(e_1) and $N.a \in$ *visibleattr*.

Condition (2) says that if *type*(*self*) is *C* and *atype*(*self*) is *D*, then the attributes of *D* can be accessed in the method bodies of the methods *D* which are inherited or over rewritten from the casted class *C*. Condition (3) ensures an attribute of an object other than *self* can be directly accessed if and only if it is an attribute in the casted type, i.e. the type of the expression itself. This would become clearer after understanding the semantics of a method invocation.

3.3 Evaluation of Expressions

The evaluation of an expression e under a given state determines its type *type*(e) and its value that is a member of *type*(e) if this type is a built-in primitive type, otherwise a value in $REF \times CNAME$. The evaluation makes use of the system configuration. An expression can only be evaluated when it is well-defined. Some well-definedness conditions are static that can be checked at compiling time, but some are dynamic. The evaluation results of expressions are given in Table. 2, where we only give an example (at the bottom of the table) about well-defined expression on built-in primitive types.

Notice the definition of type casting $(C)e$ requires that the base type e be a subclass of *C*. This is implemented in Java by the testing command $C.class.isInstance(e)$. This covers all the following both casting up when the casted type *type*(e) is a subclass of *C* too, and casting down when *type*(e) is superclass of *C*.

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 [38]). An attachment is made by an assignment, the object creation of an object or passing a parameter in a method invocation. With the approach of UTP, these different cases are unified as an assignment of a value to a program variable. Also, all other programming constructs will be defined in exactly the same way as their counter-parts in a procedural language. We only define the commands which are typical for object-orientation and the definition for the other commands remains same as in the imperative programming as we introduced in Section 2, provided they are well-defined. The semantics $[\![c]\!]$ of each command c has its well-defined condition $\mathcal{D}(c)$ as part of its precondition and thus has the form of $\mathcal{D}(c) \Rightarrow (p \vdash R)$ or $\mathcal{D}(c) \wedge p \vdash R$.

Assignments. An assignment *le* $:= e$ is well-defined if both *le* and e are well-defined and current type of e matches the declared type of *le*

$$\mathcal{D}(le := e) \stackrel{def}{=} \mathcal{D}(le) \wedge \mathcal{D}(e) \wedge (type(e) \in cname \Rightarrow (e = null) \vee (type(e) \preceq dtype(le)))$$

Notice that the well-definedess checking here includes dynamic type matching. However, for a language with strong typing, the strong static typing condition would be

Table 2. Evaluation of Expressions

Expression	Evaluation
null	$\mathcal{D}(\mathit{null}) \stackrel{def}{=} \mathit{true}, \quad \mathit{type}(\mathit{null}) \stackrel{def}{=} \mathit{NULL}, \quad \mathit{ref}(\mathit{null}) \stackrel{def}{=} \mathit{null}$
x	$\mathcal{D}(x) \stackrel{def}{=} \mathit{visible}(x) \wedge (\mathit{dtype}(x) \in \mathcal{T} \vee \mathit{dtype}(x) \in \mathit{cname})$ $\wedge \ \mathit{dtype}(x) \in \mathcal{T} \Rightarrow \mathit{head}(\overline{x}) \in \mathit{dtype}(x)$ $\wedge \ \mathit{dtype}(x) \in \mathit{cname} \Rightarrow$ $\mathit{ref}(\mathit{head}(\overline{x})) \in \mathit{ref}(\Pi(\mathit{dtype}(x)))$ $\mathit{type}(x) \stackrel{def}{=} \begin{cases} \mathit{dtype}(x) & \mathit{dtype}(x) \in \mathcal{T} \\ \mathit{type}(\mathit{head}(\overline{x})) & \text{otherwise} \end{cases}$
$le.a$	$\mathcal{D}(le.a) \stackrel{def}{=} \mathcal{D}(le) \wedge le \neq \mathit{null}$ $\wedge \ \mathit{dtype}(le) \in \mathit{cname} \wedge \mathit{visible}(le.a)$ $\mathit{type}(le.a) \stackrel{def}{=} \mathit{type}(\mathit{state}(le)(a))$ $\mathit{ref}(le.a) \stackrel{def}{=} \mathit{ref}(\mathit{state}(le)(a))$
$(C)e$	$\mathcal{D}((C)e) \stackrel{def}{=} \mathcal{D}(e) \wedge \mathit{type}(e) \notin \mathcal{T} \wedge \mathit{atype}(e) \preceq C$ $\mathit{type}((C)e) \stackrel{def}{=} C$ $\mathit{ref}((C)e) \stackrel{def}{=} \mathit{ref}(e)$
e/f	$\mathcal{D}(e/f) \stackrel{def}{=} \mathcal{D}(e) \wedge \mathcal{D}(f) \wedge \mathit{dtype}(e) = \texttt{Real}$ $\wedge \ \mathit{dtype}(f) = \texttt{Real} \wedge \mathit{value}(f) \neq 0$ $\mathit{value}(e/f) \stackrel{def}{=} \mathit{value}(e)/\mathit{value}(f)$

enough $\mathit{dtype}(e) \preceq \mathit{dtype}(le)$, as it implies $\mathit{type}(e) \preceq \mathit{dtype}(le)$. Also, together with the well-definedness $\mathcal{D}(e)$, when e is an object $\mathcal{D}(le := e)$ ensures that $\mathit{atype}(e) \preceq \mathit{dtype}(le)$.

There are two cases of assignment. The first is to (re-)attach a value to a variable (i.e. change the current value of the variable), but 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.

$$[\![x{:=}e]\!] \stackrel{def}{=} \{x\} : \mathcal{D}(x{:=}e) \vdash (\overline{x}' = \langle \mathit{value}(e) \rangle \cdot \mathit{tail}(\overline{x}))$$

As we do not allow attribute hiding or redefinition in subclasses, the assignment to a simple variable has not side-effect, and thus the Hoare triple

$$\{o_2.a = 3\}\ o_1 := o_2\ \{o_1.a = 3\}$$

is valid in our model, where $o_1 : C_1$ and $o_2 : C_2$ are variables, $C_2 \preceq C_1$ and $a : \texttt{Int}$ is protected attribute of C_1. This has made the theory simpler than the Hoare-Logic based semantics for object-oriented programming in [41].

The second case is to modify the value of an attribute of an object attached to an expression. This is done by finding the attached object in the system configuration Π and modifying its state accordingly. Thus, all variables attached to the reference of this object will be updated.

$$[\![le.a := e]\!] \stackrel{def}{=} \{\Pi(\mathit{dtype}(le))\} : \mathcal{D}(le.a{:=}e) \vdash \left(\begin{array}{l} \Pi(\mathit{dtype}(le))' = \Pi(\mathit{dtype}(le)) \uplus \\ \{o \oplus \{a \mapsto \mathit{value}(e)\} \mid o \in \Pi \wedge \mathit{ref}(o) = \mathit{ref}(le)\} \end{array} \right)$$

For example, let x be a variable of type C such that C has an attribute d of D and D has an attribute a of integer type. $x.d.a := 4$ will change state of $x = \langle r_1, C, \{d \mapsto r_2\}\rangle$, where reference r_2 is the identity of $\langle r_2, D, \{a \mapsto 3\}\rangle$ to $x = \langle r_1, C, \{d \mapsto r_2\}\rangle$, but the r_2 is now the identity of the object $\langle r_2, D, \{a \mapsto 4\}\rangle$.

This semantic definition shows the side-effect of an assignment and does reflect the object-oriented feature pointed out by Broy in [6] that an invocation to a method of an object which contains such an assignment or an instance creation defined later on, changes the system configuration Π.

Law 1. $(le := e; le := f(le)) = (le := f(e))$

Law 2. $(le_1 := e_1; le_2 := e_2) = (le_2 := e_2; le_1 := e_1)$, *provided* le_1 *and* le_2 *are distinct simple names which do not occur in* e_1 *or* e_2.

Note that the law might not be valid if le_i are composite expressions. For instance, the following equation is not valid when x and y have the same reference:

$$(x.a := 1;\ y.a := 2) = (y.a = 2;\ x.a = 1)$$

Object Creation. The $C.new(le)$ is well-defined if

$$C \in cname \wedge \mathcal{D}(le) \wedge dtype(le) \succeq C$$

The command creates a new object, attaches the object to x and set the initial values of the attributes to the attributes of x too.

$$[\![C.new(le)]\!] \stackrel{def}{=} \{le, \Pi(C)\}: \\ D(C.new(le)) \vdash \exists r \notin ref(\Pi) \cdot (AddNew(C, r) \wedge Modify(le))$$

where

$$\begin{aligned} AddNew(C, r) &\stackrel{def}{=} \Pi(C)' = \Pi(C) \\ &\quad \cup \{\langle r, C, \{a_i \mapsto init(C.a_i)\}\rangle \mid a_i \in AAttr(C)\} \\ Modify(le) &\stackrel{def}{=} \begin{pmatrix} le \in localvar \wedge \overline{le}' = \langle r, C\rangle \cdot tail(\overline{le}) \wedge \\ TypeSeq'(le) = \langle C\rangle \cdot tail(TypeSeq(le)) \\ \vee\ le \notin localvar \wedge \{le\} : true \vdash (le' = (r, C)) \end{pmatrix} \end{aligned}$$

Here we assume if $dtype(C.a_i) = M$, the assignment $a_i \mapsto init(C.a_i)$ is $a_i \mapsto M.new(C.a_i)$.

For creation of objects, we have the following laws

Law 3. $C_1.new(x); C_2.new(y) = C_2.new(y); C_1.new(x)$, *provided* x *and* y *are distinct.*

Law 4. *If* x *is not free in the Boolean expression* b, *then*

$$C.new(x); (P \lhd b \rhd Q) = (C.new(x); P) \lhd b \rhd (C.new(x); Q)$$

Local Variable Declaration and Undeclaration. Command $\texttt{var}\ T\ x = e$ declares a variable and initialises it:

$$[\![\texttt{var}\ T\ x = e]\!] \stackrel{def}{=} \{x\} : \mathcal{D}(\texttt{var}\ T\ x = e) \vdash \\ (\overline{x}' = \langle value(e)\rangle \cdot \overline{x}) \wedge TypeSeq'(x) = \langle T\rangle \cdot TypeSeq(x)$$

where

$$\mathcal{D}(\texttt{var}\ T\ x = e) \stackrel{def}{=} (x \in \mathit{localvar}) \wedge \mathcal{D}(e) \wedge \mathit{type}(e) \notin \mathcal{T} \Rightarrow \mathit{type}(e) \preceq T$$

We define $[\![\texttt{var}\ T\ x]\!] \stackrel{def}{=} \sqcap_{d \in T} [\![\texttt{var}\ T\ x = d]\!]$.

Command `end` terminates the block of permitted use a variable:

$$[\![\texttt{end}\ x]\!] \stackrel{def}{=} \{x\}{:}\mathcal{D}(\texttt{end}\ x) \vdash \overline{x}' = \mathit{tail}(\overline{x}) \wedge \mathit{TypeSeq}'(x) = \mathit{tail}(\mathit{Tseq}(x))$$

where $\mathcal{D}(\texttt{end}\ x) \stackrel{def}{=} x \in \mathit{localvar}$.

Declaration and undeclaration distribute over conditional choice.

Law 5. *If x is not free in b, then*

$$\texttt{var}\ T\ x = e; (P \lhd b \rhd Q) = (\texttt{var}\ T\ x = e; P) \lhd b \rhd (\texttt{var}\ T\ x = e; Q)$$
$$\texttt{end}\ x; (P \lhd b \rhd Q) = (\texttt{end}\ x; P) \lhd b \rhd (\texttt{end}\ x; Q)$$

Initialisation becomes void if the declared variable is updated immediately.

Law 6. $(\texttt{var}\ T\ x = e; x := f) \sqsubseteq \texttt{var}\ T\ x = f$

Note that the two commands in the above law are not equivalent it is possible that e is not well-defined.

Assignment to a variable just before the end of its scope is irrelevant if it is well-defined.

Law 7. $(x := e;\ \texttt{end}\ x) \sqsubseteq \texttt{end}\ x$

Both declaration and undeclaration are commutative.

Law 8. $(\texttt{var}\ T_1\ x = e_1; \texttt{var}\ T_2\ y = e_2) = (\texttt{var}\ T_2\ y = e_2; \texttt{var}\ T_1\ x = e_1)$, *provided y is not in e_1 and x does not appear in e_2.*

Law 9. $(\texttt{end}\ x; \texttt{end}\ y) = (\texttt{end}\ y; \texttt{end}\ x)$

Law 10. $(\texttt{var}\ T\ x = e; \texttt{end}\ y) = (\texttt{end}\ y; \texttt{var}\ T\ x = e)$, *provided y is not in e.*

Method Call. For a method signature $m(T_1\ x; T_2\ y; T_3\ z)$, let *ve*, re and *vre* be lists of expressions. Command $le.m(ve; re; vre)$ is well-defined if *le* is well-defined and it is a non-null object such that a method $m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)$ is in the casted type $\mathit{type}(le)$ of *le*:

$$\begin{aligned}\mathcal{D}(le.m(ve; re; vre)) \stackrel{def}{=}\ & \mathcal{D}(le) \wedge \mathit{type}(le) \in \mathit{cname} \wedge (le \neq \mathit{null}) \\ & \wedge N \in \mathit{cname} \cdot N \succeq \mathit{type}(le) \\ & \wedge \exists(m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)) \in \mathit{op}(N)\end{aligned}$$

The execution of this method invocation 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 under the environment of the class owning method $m()$. After it terminates, the value of the result and value-result parameters of *m* are passed back to the actual parameters *r* and *vr*.

$$\begin{aligned}[\![le.m(ve; re; vre)]\!] \stackrel{def}{=}\ & (\mathcal{D}(le.m(ve; re; vre)) \Rightarrow \\ & \exists C \in \mathit{cname} \cdot (\mathit{atype}(le) = C) \\ & \wedge \begin{pmatrix} [\![\texttt{var}\ T_1\ x = ve, T_2\ y, T_3\ z = vre]\!]; \\ [\![\texttt{var}\ C\ \mathit{self} = le]\!]; \\ [\![\mathit{Execute}(C.m)]\!];\ [\![re, vre := y, z]\!]; \\ [\![\texttt{end}\ \mathit{self}, x, y, z]\!] \end{pmatrix}\end{aligned}$$

where *Execute*(*M.m*) sets the execution environment, then executes the body and reset the environment afterwards. There are the following cases:

Case 1: If $m(T_1\ x; T_2\ y; T_3\ z)$ is not declared in C but in a superclass of C, i.e. there exists a command c such that $(m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)) \in op(N)$ for some $N \succeq C$, then

$$Execute(C.m) \stackrel{def}{=} Ex_C(superclass(C).m)$$

where if $m() \in op(M)$ then

$$Ex_C(M.m) \stackrel{def}{=} Set(C, M); SELF_C^M(body(M.m)); Reset$$

else

$$Ex_C(M.m) \stackrel{def}{=} Ex_C(superclass(M).m)$$

Case 2: If $m(T_1\ x; T_2\ y; T_3\ z)$ is declared in class C itself, that is for some command c $(m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)) \in op(C)$, then

$$Execute(C.m) \stackrel{def}{=} Set(C, C); SELF_C^C(body(C.m)); Reset$$

where

- $body(C.m)$ is the body c of the method being called.
- The design $Set(C, M)$ finds out all attributes visible to class M in order for the invocation of method m of M to be executed properly, whereas *Reset* resets the environment to be the set of variables that are accessible to the main program only:

$$Set(C, C) \stackrel{def}{=} \{\texttt{visibleattr}\} : true \vdash$$
$$\texttt{visibleattr}' = \begin{pmatrix} \{C.a \mid a \in pri(C)\} \cup \\ \bigcup_{C \preceq N} \{C.a \mid a \in prot(N) \cup pub(N)\} \cup \\ \bigcup_{N \in pubcname} \{N.a \mid a \in pub(N)\} \end{pmatrix}$$

  and when C and M are different

$$Set(C, M) \stackrel{def}{=} \{\texttt{visibleattr}\} : true \vdash$$
$$\texttt{visibleattr}' = \begin{pmatrix} \{C.a \mid a \in pri(M)\} \cup \\ \bigcup_{M \preceq N} \{C.a \mid a \in prot(N) \cup pub(N)\} \cup \\ \bigcup_{N \in pubcname} \{N.a \mid a \in pub(N)\} \end{pmatrix}$$

$$Reset \stackrel{def}{=} \{\texttt{visibleattr}\} : true \vdash$$
$$\texttt{visibleattr}' = \bigcup_{N \in pubcname} \{N.a \mid a \in pub(N)\}$$

  Set and *Reset* are used to ensure data encapsulation that is controlled by *visibleattr* and the well-definedness condition of an expression.
- The transformation $SELF_C$ on a command is defined in Table 3, which adds a prefix *self* to each attribute and each method in the command. Notice that as a method call may occur in a command that will change the execution environment, therefore after the execution of the nested call is completed the environment needs to be set back to that of C.

Table 3. The Definition of *SELF*

c or e	$SELF^M_C(c)$ or $SELF^M_C(e)$
skip	*skip*
chaos	*chaos*
$c_1 \lhd b \rhd c_2$	$SELF^M_C(c_1) \lhd SELF^M_C(b) \rhd SELF^M_C(c_2)$
$c_1 \sqcap c_2$	$SELF^M_C(c_1) \sqcap SELF^M_C(c_2)$
$\texttt{var}\ T\ x = e$	$T\ \texttt{var}\ x = SELF^M_C(e)$
$\texttt{end}\ x$	$\texttt{end}\ x$
$C.new(x)$	$C.new(SELF^M_C(x))$
$le := e$	$SELF^M_C(le) := SELF^M_C(e)$
$le.m(ve; re; vre)$	$SELF^M_C(le).m(SELF^M_C(ve); SELF^M_C(re); SELF^M_C(vre))$
$m(ve; re; vre)$	$self.m(SELF^M_C(ve); SELF^M_C(re); SELF^M_C(vre))$
$c_1; c_2$	$SELF^M_C(c_1); Set(C, M); SELF^M_C(c_2)$
$b * c$	$SELF^M_C(b) * (SELF^M_C(c); Set(C, M))$
$le.a$	$SELF^M_C(le).a$
$f(e)$	$f(SELF^M_C(e))$
null	*null*
self	*self*
x	$\begin{cases} self.x, & x \in \bigcup_{C \preceq N} Attr(N) \\ x, & otherwise \end{cases}$

Notice that semantics of a method call defines the method binging rules to ensure that

- only a method with a signature declared in the casted type or above the casted type in the inheritance hierarchy can be accessed, and
- method that is executed is the one defined in the lowest position the inheritance hierarchy from the current type of the active object.

We did not introduce the syntax $super.m$ to explicitly indicate the call to a method according to its definition in the superclass. There is no difficulty to introduce $super.m$ and define its semantics accordingly.

Example 1. To illustrate the semantics of a method invocation, we can consider the bank system with the UML class diagram in Figure 1. We define $Execute(C.m)$ for the method *withdraw*() in the classes of current account and saving account *CA* and *SA*. We assume all classes, except for *Bank*, are private classes, and further notice that

1. the body of *withdraw*() in the superclass *Account* is

   $$balance > x \vdash balance' = balance - x$$

2. subclass *SA* inherits *withdraw*() from *Account*, and
3. subclass *CA* overwrites the body of *withdraw*() into

   $$balance := balance - x$$

For class *CA*,

$$\begin{aligned} \mathit{Execute}(\mathit{CA.withdraw}) &= \mathit{Set}(\mathit{CA},\mathit{CA}); \mathit{SELF}^{\mathit{CA}}_{\mathit{CA}}(\mathit{balance} := \mathit{balance} - x); \mathit{Reset} \\ &= \mathit{visibleattr} := \{\mathit{CA.blance}, \mathit{CA.aNo}\}; \\ &\quad \mathit{self.balance} := \mathit{self.balance} - x; \\ &\quad \mathit{visibleattr} := \emptyset \end{aligned}$$

According to the semantics of a method call to $o.\mathit{withdraw}(e)$, where o is an object of *CA*, the execution of this method call first attaches o to *self*, and then executes the method according to the semantics of $\mathit{Execute}(CA.\mathit{withdraw})$ defined above. It shows that the method is executed according to the current type *CA* and the method is the method of the subclass.

For the case of a saving account

$$\begin{aligned} &\mathit{Execute}(\mathit{SA.withdraw}) \\ &= \mathit{Set}(\mathit{SA},\mathit{Account}); \mathit{SELF}^{\mathit{Account}}_{\mathit{SA}}(\mathit{Account.withdraw}); \mathit{Reset} \\ &= \mathit{visibleattr} := \{\mathit{SA.blance}, \mathit{SA.aNo}\}; \\ &\quad \mathit{self.balance} > x \vdash \mathit{self.balance}' = \mathit{self.balance} - x; \\ &\quad \mathit{visibleattr} := \emptyset \end{aligned}$$

Thus, the invocation to a withdraw method of a saving account is executed according to the definition of the method in the superclass *Account*. ♣

Semantics of Object Systems. Having defined the semantics of a class declaration section and a command, we combine them to define the semantics of an object program $(\mathit{Cdecls} \bullet \mathit{Main})$.

Recall that *Main* consists of a set *externalvar* of the external variables with their types and a command c. For simplicity but without loss of expressive power, we assume that any primitive command in c is in one of the following forms:

1. an assignment $x := e$ such that $x \in \mathit{externalvar}$ and e does not contain sub-expressions of the form $le.a$. That is, we do not allow direct access to object attributes in the main method.
2. a creation of a new object $C.\mathit{New}(x)$ for a variable $x \in \mathit{externalvar}$,
3. a method call $x.m(\mathit{ve}; \mathit{re}; \mathit{vre})$, where x is a variable in *externalvar*.

Main is well-defined if the types of all variables in *externalvar* are either built-in primitive types or public classes declared in *pubcname*:

$$\mathcal{D}(\mathit{Main}) \stackrel{def}{=} \bigwedge_{x \in \mathit{externalvar}} (\mathit{dtype}(x) \in \mathit{pubcname} \vee \mathit{dtype}(x) \in \mathcal{T})$$

The semantics of *Main* is then defined to be

$$[\![\mathit{Main}]\!] \stackrel{def}{=} \mathcal{D}(\mathit{Main}) \Rightarrow [\![c]\!]$$

Before *Main* is executed, the well-definedness of the declaration section has to be checked and the local variables have to be initialised to empty sequences. For this we define a design *Init*:

$$\begin{aligned} \mathit{Init} \stackrel{def}{=} \mathcal{D}(\mathit{Cdecls}) \vdash \mathit{visibleattr}' = \emptyset \wedge (\Pi' = \emptyset) \wedge \\ \bigwedge_{x \in \mathit{var}} (\overline{x}' = <> \wedge \mathit{TypeSeq}'(x) = <>) \end{aligned}$$

Definition 4. *The* **semantics of an object program** *Cdecls • Main is defined to be the following sequential composition*

$$[\![Cdecls \bullet Main]\!] \stackrel{def}{=} \exists\, \Omega, \Omega', glb, glb' \cdot ([\![Cdecls]\!]; Init; [\![Main]\!])$$

This definition of the *closed* semantics allows us to hide the internal information in the execution of a program, only observing the relation between the pre-state and post-state of the external variables whose types are built-in primitive types, and the object type information of the external variables whose types are declared as classes. We cannot observe the information of the objects attached to these variables. We have a less abstract definition for the semantics of an object program.

We define the *open semantics* $[\![Cdecls \bullet Main]\!]_o$ of *Cdecls • Main* as

$$\exists \{\Pi\} \gg glb, \{\Pi'\} \gg glb', \Omega, \Omega' \cdot \\ ([\![Cdecls]\!]; Init; [\![Main]\!]; [\![\Pi' := \Pi\downarrow_{externalvar}]\!])$$

The open semantics allows us to observe the full information about the states of external variables. We can insert the command $\Pi' := \Pi\downarrow_{externalvar}$ at any point of the main method without changing the open and close semantics of a program.

Lemma 1. *For any object program $S = Cdecls \bullet Main$ with c as the command in the main method, we have*

1. $[\![S]\!] = \exists \Pi, \Pi' \cdot [\![S]\!]_o$.
2. *If c is of the form $c_1; c_2$, let S_2 be the program which replaces the command c with $c_1; \Pi' := \Pi\downarrow_{externalvar}; c_2$, then $[\![S]\!]_o = [\![S_2]\!]_o$.*
3. *If c is of the form $c_1; b * (c_2; c_3); c_4$, let S_3 be the program which replaces the loop in Main with $b * (c_2; \Pi' := \Pi\downarrow_{externalvar}; c_3)$, then $[\![S]\!]_o = [\![S_3]\!]_o$.*
4. *If c is of the form $c_1; (c_2; c_3) \lhd b \rhd c_4; c_5$, let S_4 be the program which replaces the conditional choice in Main with $(c_2; \Pi' := \Pi\downarrow_{externalvar}; c_3) \lhd b \rhd c_4$, then $[\![S]\!]_o = [\![S_4]\!]_o$.*
5. *If c is of the form $c_1; (c_2; c_3) \sqcap c_4$, let S_5 be the program which replaces command c in Main with $c_1; (c_2; \Pi' := \Pi\downarrow_{externalvar}; c_3) \sqcap c_4$, then $[\![S]\!]_o = [\![S_5]\!]_o$.*

4 Object-Oriented Refinement

We would like the refinement calculus to cover not only the early development stages of requirements analysis and specification but also the later stages of design and implementation. This section presents the results of our exploration on three kinds of refinement:

1. Refinement relation between object systems.
2. Refinement relation between declaration sections (*structural refinement*).

We only present the definitions and some laws. For detailed study with proofs, we refer to the full version of the paper in [16]. The refinement calculus is used in a case study of the development of a Point of Sale Terminal (POST) [36].

4.1 Refinement of Object Systems

We have defined the refinement relation between commands and shown some examples in the previous section. We now define what we mean by a refinement between two object programs and then focus on the structural refinement. The notation of structural refinement is actually an extension to the notion of data refinement [19].

Definition 5. *Let $S_i = Cdecls_i \bullet Main_i$, $i = 1, 2$, be object programs which have the same set of external variables externalvar. S_1 is a* **refinement** *of S_2, denoted by $S_1 \sqsupseteq_{sys} S_2$, if the following implication holds:*

$$\forall externalvar, externalvar', ok, ok' \cdot ([\![S_1]\!] \Rightarrow [\![S_2]\!])$$

Example 2. For any class declaration *Cdecls*, we have the following:

1. $S_1 = Cdecls \bullet (\{x : C\}, C.new(x))$ and $S_2 = Cdecls \bullet (\{x : C\}, C.new(x); C.new(x))$ are equivalent.
2. We assume class $C \in pubcname$, $\langle a : \texttt{Int}, d\rangle \in attr(C)$, $get(\emptyset; \texttt{Int}\ z; \emptyset)\{z := a\}$ and $update()\{a := a + c\}$ in $op(C)$, then

$$Cdecls \bullet (\{x : C, y : \texttt{Int}\}, C.new(x); x.update(); x.get(y))$$

and

$$Cdecls \bullet (\{x : C, y : \texttt{Int}\}, C.new(x); x.update(); x.get(y); C.new(x))$$

are equivalent.

Proof. We give a proof for item (2) of this example. We denote the first program by S_1 and the second by S_2. Assume the declaration section is well-defined, as otherwise both programs are chaos. Then it is easy to check the main methods are both well-defined. The structural variables Ω are calculated according to the definition. Let d be the initial value of attribute a of C and σ_0 denote the initial state of an object of C when it is created. We calculate the semantics of S_1:

$$\begin{aligned}
&[\![C.new(x); x.update(), x.get(y)]\!] \\
=&\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0\rangle\} \wedge x' = \langle r, C\rangle); \\ [\![x.update(); x.get(y)]\!] \end{pmatrix} \\
=&\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0\rangle\} \wedge x' = \langle r, C\rangle) \wedge \\ self' =<> \wedge \Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle \mid r = ref(x)\}); \\ [\![x.get(y)]\!] \end{pmatrix} \\
=&\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge (self' =<>); \\ [\![x.get(y)]\!] \end{pmatrix} \\
=&\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<>); \\ true \vdash self' =<> \wedge z' =<> \wedge y' = d + c \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\} \end{pmatrix} \\
=&\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\} \end{pmatrix}
\end{aligned}$$

The semantics $[\![S_1]\!]$ hides Ω, Π, *self* and z by existential quantification. Let $[\![Cdecls]\!]$ be $true \vdash \Omega = \emptyset \wedge \Omega' = \Omega_0$, we have $[\![S_1]\!]$ equals to

$$\exists \left\{ \begin{array}{l} \Omega, \Omega', self, self', z, z', \\ visibleattr, visibleattr' \end{array} \right\} \cdot ([\![Cdecls]\!]; Init; [\![C.new(x); x.update(), x.get(y))]\!])$$
$$= true \vdash \exists r \in REF \cdot x' = \langle r, C\rangle \wedge y' = c + d$$

The main method of S_2 is the main method of S_1 followed by command $C.new(x)$ and thus its semantics equals

$$\begin{array}{l} [\![C.new(x); x.update(), x.get(y)]\!]; [\![C.new(x)]\!] \\ = \left(\begin{array}{l} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\}; \\ [\![C.new(x)]\!] \end{array} \right) \\ = \left(\begin{array}{l} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\}; \\ true \vdash \exists p \notin ref(\Pi) \cdot \Pi' = \Pi \cup \{\langle p, C, \sigma_0\rangle\} \wedge (x' = \langle p, C\rangle) \end{array} \right) \\ = \left(\begin{array}{l} true \vdash \exists r, p \in REF \cdot ((p \neq r) \wedge \\ \qquad \Pi' = \{\langle p, C, \sigma_0\rangle, \langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle p, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\} \end{array} \right) \end{array}$$

Hiding the internal variables, $[\![S_2]\!]$ equals

$$true \vdash \exists p \in REF \cdot x' = (p, C) \wedge y' = c + d$$

Thus, we have proved, S_1 and S_2 refines each other.

However, If we change the main methods of these two programs by adding another $x.get(y)$ to the end of both of them. They are not equivalent anymore. The final value of y for the first program remains will be still $d + c$, but for the second one, the final value of y gets the initial value d after the execution. ♣

The discussion at the end of the example shows that program refinement is not quite compositional. In other words, for two main methods, $Main_i = (externalvar, c_i)$, $i = 1, 2$,

$$Cdecls_1 \bullet Main_1 \sqsupseteq_{sys} Cdecls_2 \bullet Main_2$$

does not in general imply

$$Cdecls \bullet (externalvar, c_1; c) \sqsupseteq_{sys} Cdecls \bullet (externalvar, c_2; c)$$

The main reason for this is the global internal variable Π is hidden in the semantics. In fact any program that has internal variables does not have such compositionality.

Theorem 2. *Let Cdecls • Main, C be a public class declared in Cdecls and $Cdecls_1$ be obtained from Cdecls by changing C to a private class. Then if C is not referred in Main,*

$$Cdecls \bullet Main =_{sys} Cdecls_1 \bullet Main$$

where $=_{sys}$ is the equivalence relation $\sqsupseteq_{sys} \cap \sqsubseteq_{sys}$.

The relation $\sqsupseteq_{sys}$ is reflexive and transitive.

4.2 Structure Refinement

The proof in **Example** 2 shows that the local variables and *visibleattr* of a program are constants after each method invocation. When the main methods in the programs are syntactically identical, the relation between their system states is determined by the relation between the structure of these programs, i.e. their class names, attributes, sub-superclass relations, and methods in the classes.

An object-oriented program design is mainly about designing classes and their methods, and a class declaration section can in fact support many different application main programs. The rest of this section focuses on structural refinement.

Definition 6. *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:*

$$Cdecls_1 \sqsupseteq_{class} Cdecls_2 \stackrel{def}{=} \forall Main \cdot (Cdecls_1 \bullet Main \sqsupseteq_{sys} Cdecls_2 \bullet Main)$$

Intuitively, it states that $Cdecls_1$ supports at least the same set of services as $Cdecls_2$. It is obvious that $\sqsupseteq_{class}$ is reflexive and transitive. We use $=_{class}$ to denote the equivalence relation $\sqsupseteq_{class} \cap \sqsubseteq_{class}$. When there is no confusion, we omit the subscript when we discuss about structural refinement.

A structural refinement does not allow to change the main method. So every public class in $Cdecls_2$ has to be declared in the refined declaration section $Cdecls_1$, and every method signature in a public class of $Cdecls_2$ has to be declared in $Cdecls_1$, otherwise there are main methods which are well-defined under $Cdecls_2$ but not under $Cdecls_1$. Also recall that a main method only change objects by method invocations to public classes.

In the full version of rCOS for object systems [16], we have shown how structural refinement between two class declaration sections by *structural transformations* and *upwards* and *downwards simulations* of public class methods. A structural transformation between two declaration sections is actually a transformation between the corresponding UML class diagrams of the declaration sections. In [16], a proof is given to show that the two classes diagrams (class declaration sections) in Figure 2 refine each other if *C* is the only public class.

4.3 Laws of Structural Refinement

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:

1. 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.
2. 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.
3. 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.

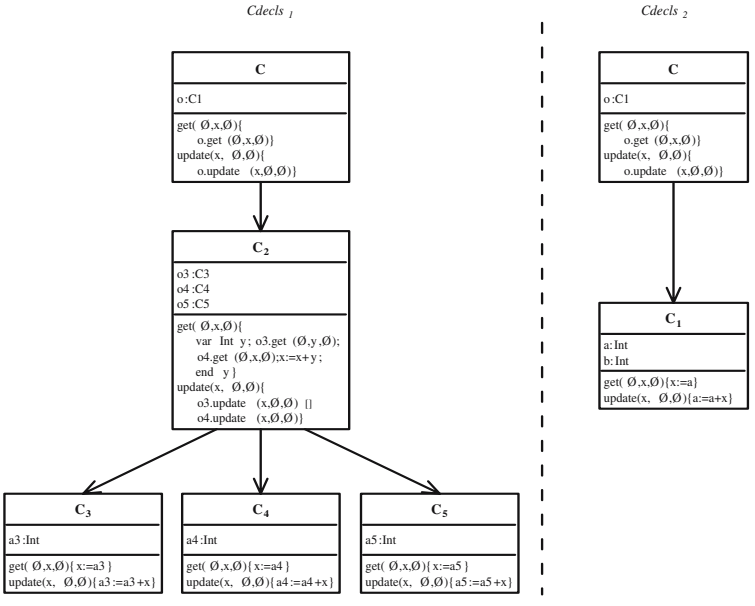

Fig. 2. Example of Structural Refinement

4. 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.
5. 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.
6. 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*.
7. Moving some attributes from a class to its direct superclass.
8. 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*.
9. Moving common attributes of classes which are direct subclasses of a class to the superclass.
10. Moving a method from a class to its direct superclass.
11. Copying (**not** removing) a method of a class to its direct subclass.
12. 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) [25] 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 11. (**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 3).*

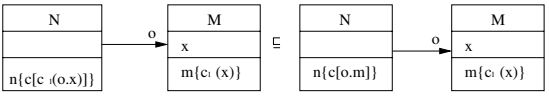

Fig. 3. Expert Pattern

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 12. (**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 4.*

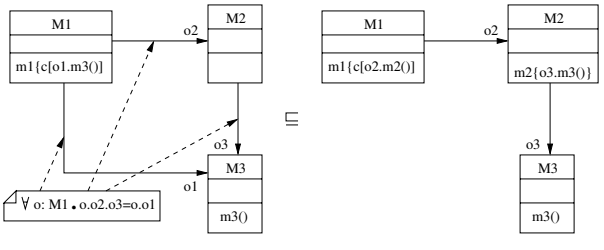

Fig. 4. Low Coupling Pattern

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 13. (**High Cohesion**) *Assume two methods $m_1()$ and $m_2()$ in 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 5.*

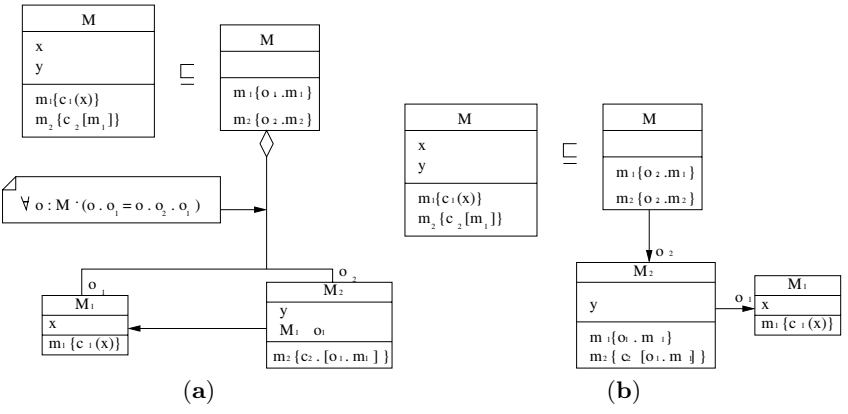

Fig. 5. High Cohesion pattern

The case (a) in Law 13 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 [13], such as *Adaptor Pattern*, *Observer Pattern*, *Strategy Pattern* and *Abstract Factory Pattern* can also be formalized.

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 [26,32]. In [25,29], the five GRASP patterns are systematically used for the development of a case study.

5 Component Systems

Using components to build and maintain software systems is not a new idea. However, it is today's growing complexity of these systems that forces us to turn this idea into practice [45,9,18]. While component technologies such as COM, CORBA, and Enterprise JavaBeans are widely used, there is so far no agreement on standard technologies for designing and creating components, nor on methods for composing them. Finding appropriate formal approaches for specifying components, the architectures for composing them, and the methods for component-based software construction, is correspondingly challenging. In this section, we consider a contract-oriented approach to the specification, design and composition of components. Component specification is essential as it is impossible to manage change, substitution and composition of components if components have not been properly specified.

5.1 Introduction

When we specify a component, it is important to separate different views about the component. From its user's (i.e. external) point of view, a component *Comp* consists

of a set of *provided services* [45]. The syntactic specification of the provided services is described by an interface, defining the operations that the component provides with their signatures. This is also called the *syntactic specification* of a component. COM and CORBA use IDL, and JavaBeans uses Java Programming Language to specify component interfaces. Such a syntactic specification of a component does not provide any information about the effect, i.e. the *functionality* of invoking an operation of a component or the *behavior*, i.e. the temporal order of the interface operations, of the component.

For the functional specification of the operations in an interface, it is however necessary to know the conceptual state of the component. Consequently, the interface specification contains a so-called *information model* [9,11]. In the context of such a model, we still specify an operation m by a *design* $p(x) \vdash R(x, x')$ that is seen as a *contract* between the component and its client [9,18]. This definition of a *contract* also agrees with that of [39,40]. To use the service m, a client has to ensure the pre-condition $p(x)$, and when this is true the component must guarantee the post-condition Q. We then define a *contract* of an interface by associating the interface with a set of *features* that we will call *fields* and assigning each a design $MSpec(m)$ to each interface operation m. The types of the fields are given in a *data/class model*.

The contract for the provided interface of a component allows the user to check whether the component provides the services required by other components in the system, without the need to know the design and implementation of the component. It also commits (or requires) the designers of the component who have to design the component's provided services. A designer of the component under consideration (*CuC*) may decide to use services provided by other components. These services are called *required services* [45] of CuC. Components that provide the required services of CuC can be built by another team or bought as a component-off-the-shelf (COTS). To use a component to assemble a system, one needs to know the specifications of both its provided and required services.

We will specify the design of a component by giving each operation m in the provided interface a program specification text $MImpl(m)$ in rCOS. In *MImpl(m)*, calls to operations in a required interface are allowed. We can then verify whether $MImpl(m)$ refines the specification of m given in a contract of the provided interface. The *verifier* of a component needs to know the contracts of the provided interfaces, the contracts of the required interfaces, and the specification text for each operation m of the provided interface. We can thus understand a component as a relation between contracts of the required interfaces and contracts of the provided interface: given a contract for each required interface, we can calculate a design of an m from $MImpl(m)$ and check whether it conforms to the specification $MSpec(m)$ defined by the contract of the provided interface. A design of a component can be further refined into an implementation by refining the data/class model and then operation specifications $MImpl(m)$.

A component assumes an architectural context defined by its interfaces. We *connect* or *compose* two components $Comp_1$ and $Comp_2$ by linking the operations in the provided interface of one component to the matching operations of a required interface of another. For this, we have to check whether the provided interface of component $Comp_1$ contains the operations of a required interface of component $Comp_2$, and whether the contract of the provided interface of $Comp_1$ meets the contract of the required interface of $Comp_2$.

If $Comp_1$ and $Comp_2$ match well, the composition $Comp_1||Comp_2$ forms another component. The provided interface of $Comp_1||Comp_2$ is the *merge* of the provided interfaces of $Comp_1$ and $Comp_2$. The required interfaces of $Comp_1||Comp_2$ are the union of required interfaces of $Comp_1$ and $Comp_2$, excluding (*by hiding*) the matched interfaces of $Comp_1$ and $Comp_2$. For defining composition, interfaces can be *hidden* and *renamed*.

A component is also replaceable, meaning that the developer can replace one component with another, may be *better*, as long as the new one provides and requests the same services. A component is better than another if it can provide more services, i.e. the contracts for its provided interfaces refine those of the other, with the same required services. Component replaceability is based on the notion of *component refinement*.

In this section we present a model of components that allows us to

- describe and check the syntactic dependency and composability among components in terms of interfaces,
- specify and reason about function composability and substitutability of a component in terms contracts,
- correctness and substitutability of component designs and implementation with respect the contract specification of the component.

We leave the specification, correctness and substitutability of interaction protocols of components in future work.

5.2 Interfaces

An *interface I* is a set of *operation* (or *method*) *signatures* $m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z})$, where m is called the *name* of the operation. An interface can be specified as a family of operation signatures in the following format:

$$
\begin{array}{l}
\textbf{Interface } I \{ \\
\qquad \textbf{Method} : m_1(\underline{U}_1\ \underline{x}_1; \underline{V}_1\ \underline{y}_1; \underline{W}_1\ \underline{z}_1); \\
\qquad\qquad \ldots; \\
\qquad\qquad m_k(\underline{U}_k\ \underline{x}_k; \underline{V}_k\ \underline{y}_k; \underline{W}_k\ \underline{z}_k) \\
\}
\end{array}
$$

Merge Interfaces

It is often the case that there are a number of components, each providing a part of the operations in the required interface of another component. We thus need to *merge* these components to provide one single interface to match the interface required by the other component.

Two interfaces I_1 and I_2 are *composable* provided that every operation name that appears in both I_1 and I_2 must be declared with the same signature. This condition is not too restrictive as to use a component designed for an application in another or specialize a generic component for a special application, renaming or adding a *connector* component [2,42] can be used to *customize* the component.

Definition 7. *Let* $\{I_k : \mid k \in K\}$ *be a finite family of composable interfaces. Their* **merge** $\uplus_{k\in K} I_k$ *is defined by* $\uplus_{k\in K} I_k \stackrel{def}{=} \cup_{k\in K} I_k$.

5.3 Contracts

Only a *syntactic specification* of its interface is not enough for the use or the design of a component. We also need to specify the effect, i.e. the *functionality*, of invoking an interface operation. This requires one to associate the interface to a *conceptual state space*, and a specification of how the states are changed by the operation under certain pre-conditions. We view such a *functional specification* of an interface as a contract between the component *client* and the component *developer*. The contract is the specification of the component that the developer has to implement. The contract is also between a user of the component and a provider of an implementation of the interface: the component has to provide the services promised by the specification *provided* that the user uses the component according to the precondition. To define the conceptual state space of a contract for an interface and the types for the parameters of the interface operations, we assume that a type is either a primitive built-in or a class of objects. This allows our framework to support both imperative and object-oriented programming in the design of a component. The type definitions is given by a class declaration section and it declares a class structure Ω, called the *information model* of the component.

Given an interface *I*, an information model Ω declared by a class declaration section, a set *A* of variable declarations of the form $T\ x$ where T is either a primitive type or a class declared in Ω, called the type of x, we define the alphabet α as the union of sets of the variables, the input and output parameters of the operations of *I*.

$$
\begin{array}{ll}
in\alpha & \stackrel{def}{=} A \cup \{x \in \underline{x} \cup \underline{z} \mid m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z}) \in I\} \\
out\alpha & \stackrel{def}{=} A \cup \{y \in \underline{y} \cup \underline{z} \mid m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z}) \in I\} \\
out\alpha' & \stackrel{def}{=} \{x' \mid x \in out\alpha\} \\
\alpha & \stackrel{def}{=} in\alpha \cup out\alpha
\end{array}
$$

A *conceptual state* for $\langle I, \Omega \rangle$ is a well-typed mapping from the variables α to their value spaces. It is in fact the state space Ξ determined by Ω, plus values of variables in $out\alpha$ of primitive types that is a snapshot of the models consisting the current objects of the classes and links by the associations or attributes that relate these objects, as well as the values of variables of primitive types. As before, a *specification* of an operation $m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z})$ in an alphabet α is a *framed design* $\beta : Spec$.

Definition 8. *A* **contract** *is a tuple Contr* $= (I, \Omega, A, MSpec, Init)$ *where I is an interface,* Ω *is the information model, A is a set of variables, called the* fields *of Contr, whose types are either declared in* Ω *or primitive types, and MSpec a function that maps each operation of I to a specification, and Init an initial condition that defines some values to fields as their initial values.*

If no field is of an object type, we will omit the information model from the specification of a contract. In modular programming, a primitive contract is a specification of a module that defines the behavior of the operations in its interface. However, later we will see that contracts can be *merged* to form another contract and this corresponds to the merge of a number of modules. In object-oriented programming, a primitive contract specifies an initialized class, i.e. an object, whose public methods are operations in the interface. This class *wraps* the classes in the information model Ω, and provides the

interface operations to the environment. In the Java-like rCOS syntax, such a contract can be written as

Interface I {**Meth** : $\{m() \mid m() \in I\}$};
Cdecls;
Class C implements I {**Attr** : $A = Init$;
Method : $\{m()\{MSpec(m)\} \mid m \in I\}$;
main()$\{C.New(x)\}$
}

where **main** provides the condition *Init* when creating the new object of *C* attached to x with the initial values of the attributes in *A*.

Contracts of interfaces can be merged only when their interfaces are composable and the specifications of the common methods are *consistent*. This merge will be used to calculate the provided and required services when components are composed.

Definition 9. *Contracts* $(I_i,\ \Omega_i,\ A_i,\ MSpec_i,\ Init_i)$, $i = 1, 2$, *are* **consistent** *if*

1. I_1 *and* I_2 *are composable.*
2. *If* x *is declared in both* A_1 *and* A_2*, it has the same type; and* $Init_1(x) = Init_2(x)$.
3. *Any class name C in both* Ω_1 *and* Ω_2 *has the same class declaration in them.*
4. $MSpec_1(m) \Leftrightarrow MSpec_2(m)$ *for all* $m \in I_1 \cap I_2$.

This definition can be extended to a finite family of contracts.

Definition 10. *Let* $\{Contr_k = (I_k, \Omega_k, A_k, MSpec_k, Init_k)\}$ *be a consistent finite family of contracts. Their* **merge***, (denoted by* $\|_{k \in K}\ Contr_k$*), is defined by*

$$I \stackrel{def}{=} \uplus_k I_k, \qquad \Omega \stackrel{def}{=} \otimes_k \Omega_k, \qquad A \stackrel{def}{=} \otimes_k A_k,$$
$$Init \stackrel{def}{=} \otimes_k Init_k,\ MSpec \stackrel{def}{=} \otimes_k MSpec_k$$

where $\otimes$ *denotes the overriding operator, e.g.* $(MSpec_k \otimes MSpec_{k+1})(m) = MSpec_{k+1}(m)$ *if* $m \in I_k \cap I_{k+1}$*;* $MSpec_k(m)$ *if* $m \in I_k$ *but* $m \notin I_{k+1}$*;* $MSpec_{k+1}(m)$ *otherwise.*

A merge of a family of contracts corresponds the construction of a conceptual model from the partial models of the application domain in the contracts. There are three cases about the partial models:

1. The contracts do not share any fields or modelling elements in their conceptual models. In this case, the system formed by the components of these contracts are most loosely coupled. All communications are via method invocations. Such a system is easy to design and maintain. Composing these components is only plug-in composition.
2. The contracts may share fields, but their conceptual models do not share any common model elements. In this case, application domain is partitioned by the conceptual models of these contracts. And components of the system are also quite loosely coupled and easy to construct and maintain. When composing these components, some simple wiring is needed.

3. The contracts share common model elements in their conceptual models. The refinement/design of the contracts has to preserve the consistency and integrity, generally specified by state invariants, of the model. The more elements they share, the more tightly the components are coupled and the more wiring is needed when composing these components.

Definition 11. *We say that a contract* $Contr_1 = (I_1, \Omega_1, A_1, MSpec_1, Init_1)$ *is* **refined** *by* $Contr_2 = (I_2, \Omega_2, A_2, MSpec_2, Init_2)$*, denoted by* $Contr_1 \sqsubseteq Contr_2$*, if there is a mapping* ρ *from* A_1 *to* A_2 *satisfying*

1. *The initial state is preserved:* $(\underline{x} := Init_1(\underline{x}); \rho) \sqsubseteq (\rho; \underline{y} := Init_2(\underline{y}))$*, where* $\underline{x}$ *is the list of variables defined in* A_1*, and* $\underline{y}$ *the list of variables in* A_2*.*
2. *The behavior of the operations of* $Contr_1$ *are preserved: every operation* m *declared in* I_1 *is also declared in* I_2 *and* $(MSpec_1(m); \rho) \sqsubseteq (\rho; MSpec_2(m))$*.*

The refinement relation between contracts will be used to define component refinement. The state mapping ρ allows that a component developed in an application domain can be used in another application domain if such a mapping can be found.

Theorem 3. *Contract refinement enjoys the properties of program refinement.*

1. $\sqsubseteq$ *is reflexive and transitive and a pre-order.*
2. *(***An upper bound condition***) The merge of a family of contracts refines any contract in the family.*
3. *(***A monotonicity condition***) The refinement relation is preserved by the merge operation on contracts. That is for two consistent families of contracts* $\{Contr_k^i \mid k \in K\}$*,* $i = 1, 2$*. If they do have shared fields and* $Contr_k^1 \sqsubseteq Contr_k^2$ *for all* $k \in K$*, then we have* $\|_{k \in K} Contr_k^1 \sqsubseteq \|_{k \in K} Contr_k^2$*.*

5.4 Component

A component consists of a provided interface and optionally a required interface, and an executable code which can be coupled to the codes of other components via their interfaces.

Definition 12. *A* **component** *Comp is a tuple* $< O, I, \Omega, A, MImpl, Init, R >$ *where*

- O *is an interface, called the* provided *or (*output*)* interface *of Comp.*
- I *is an interface disjoint from* O*, called the* internal interface *of Comp*
- Ω *is an information model*
- A *is a set of fields whose types are all declared in* Ω*.*
- *MImpl maps each operation declared in* $O \cup I$ *to a pair* (α, Q)*, where* Q *is a command written in* rCOS*, and* α *is the alphabet obtained from* A *and the input and output parameters of the operations in* $O \cup I$*.*
- R *is the interface that is disjoint from* O *and* I *and consists of the* operations *(not methods of classes in* Ω*) which are referenced in the program text* $MImpl(m)$ *and bodies of methods of classes in* Ω *but not in* $O \cup I$*, where* $m \in O \cup I$*.* R *is called the* input *or* required *interface of Comp.*

We call $Contr = (O, I, \Omega, A, MImpl, Init)$ *a* **generalized contract***, as it has internal operations and MImpl provides the specification of each operation of O in terms a general* rCOS *command.*

We will use the 4-tuple $(Contr, I, O, R)$ to denote a component, where *Contr* is a generalized contract for the interface $O \uplus I$.

A contract for R is called a *required services* of the component and a contract of the interface O a *provided services*. Operations in R can be seen as *holes* in the component where their specifications or implementation given in other components that are to be plugged in. Therefore, the provided services of a component depends on its required services plugged in from other components. This leads to the definition of our semantics of a component.

In the above definition, we introduced private operations so that we can hide an output operation by making it a private operation. This will keep the definition *MImpl* valid as the hidden operations may be called in $MImpl(m)$. Hiding interface operations allows to offer different services to different clients.

Definition 13. *(***Hiding***) Let* $Contr = (O, I, A, \Omega, MImpl, Init)$ *be a general contract, and* $H \subseteq O$ *a set of operations. The notation* $Contr\backslash H$ *represents the contract*

$$(O \setminus H, I \cup H, \Omega, A, MImpl, Init)$$

where $S\backslash S_1$ *is set-subtraction.*

Theorem 4. *The hiding operator enjoys the following properties.*

1. $(Contr\backslash H) \sqsubseteq Contr$.
2. $Contr\backslash\emptyset = Contr$.
3. $Contr\backslash H = Contr\backslash(H \cap O)$, *where I is the interface of Contr.*
4. $(Contr\backslash H_1)\backslash H_2 = Contr\backslash(H_1 \cup H_2) = (Contr\backslash H_2)\backslash H_1$
5. $(\|_{k \in K} Contr_k)\backslash H = \|_{k \in K}(Contr_k \backslash H)$

5.5 Semantics Components

Definition 14. *The* **semantics** *of a component Comp is defined as a binary relation between its required services and their corresponding provided services*

$$[\![Comp]\!](Contr_R, Contr'_O) \stackrel{def}{=} (Contr_R >> Comp) \sqsubseteq Contr'_O$$

where the variable $Contr_R$ *takes an arbitrary required service as its value,* $Contr'_O$ *takes a provided service for O, and the notation* $Contr_R >> Comp$ *denotes the provided service*

$$(O, F(\Omega), A, MSpec, Init)$$

where $F(\Omega)$ *is the class model obtained from* Ω *by removing the methods of its classes, and mapping MSpec is defined from the given required service*

$$Contr_R =< R, \Omega_R, A_R, MSpec_R, Init_R >$$

by the recursive equations $MSpec(m) = \mathcal{M}(MImpl(m))$*, where* $\mathcal{M}$ *replaces every call of* $m(inexp, outvar)$ *with the actual input parameters* $inexp$*, output parameters* $outvar$ *and value-result parameters* $vrexp$ *of* O *by its corresponding specification.*

$$\mathcal{M}(m(inexp; outvar; vrexp)) \stackrel{def}{=} \left(\begin{array}{l} \textbf{var}\ T_1\ x = inexp, T_2\ y = outvar, T_3\ z = vrexp; \\ MSpec_R(m); outvar, vrexp := y, z; \\ \textbf{end}\ x,\ y,\ z \end{array}\right)$$
$$\textit{if} \quad m(T_1\ x; T_2\ y; T_3\ z) \in R$$

$$\mathcal{M}(m(inexp; outvar; vrexp)) \stackrel{def}{=} \left(\begin{array}{l} \textbf{var}\ T_1\ x = inexp, T_2\ y = outvar, T_3\ z = vrexp; \\ MImpl(m); outvar, vrexp := y, z; \\ \textbf{end}\ x,\ y,\ z \end{array}\right)$$
$$\textit{if} \quad m(T_1\ x; T_2\ y; T_3\ z) \in O \cup I$$

$$\mathcal{M}(v := e) \stackrel{def}{=} v := e$$

$$\mathcal{M}(\mathcal{F}(\mathsf{c})) \stackrel{def}{=} \mathcal{F}(\mathcal{M}(\mathsf{c})) \quad \textit{for any comand c and context } \mathcal{F}$$

Notice that when a component *Comp* has an empty set of required interface operations, *Comp* is a *closed component* and the notation $Contr_{\emptyset} >> Comp$ becomes a constant that is the semantics of the closed program *Comp*.

For a given contract $Contr_R$ for the required interface of *Comp*, $Contr_R >> Comp$ is a closed component. Let $Contr_O$ be a contract of the provided interface of *Comp* which serves as the specification of the component. We say that *Comp correctly realizes* or *implements* $Contr_O$ with a given required service $Contr_R$ if $Contr_O \sqsubseteq (Contr_R >> Comp)$.

In a modular programming paradigm, a component can be designed and implemented as a module in which each of the operations in the output interface is "programmed" using procedures or functions that are defined either locally in the module or externally in other modules. In this case, the external modules that the component calls methods from must be declared, as well as the types of the attribute values and parameters of its methods. Therefore, a component is in fact not a single module, but an artifact that contains all these declared types and modules. In an object-oriented paradigm, such as Java, a component can be seen as a class that implements the interfaces in O. Including the notation for interfaces and contracts in rCOS, the language provides a formal model for components and the calculus of contract refinement and component refinements.

5.6 Refinement and Composition of Components

For a component *Comp* with provided and required interfaces O and R, the semantics $[\![Comp]\!]$ is a binary relation between the input services and output services.

Theorem 5. ***(Monotonicity and Upwards Closure* [44]*)*** *Let* $Comp =< Contr, I, O, R >$ *and* $\sqsubseteq_R$ *and* $\sqsubseteq_O$ *are the refinement relations among contracts of* R *and among contracts of* O *respectively. Then* $\sqsubseteq_R \circ [\![Comp]\!] \circ \sqsubseteq_O \ = \ [\![Comp]\!]$*, where* $\circ$ *denotes relational composition.*

Thus, for any required services $Contr_R \sqsubseteq Contr'_R$, and provided services $Contr_O \sqsubseteq Contr'_O$, then

$$[\![Comp]\!](Contr_R, Contr'_O) \Rightarrow [\![Comp]\!](Contr'_R, Contr_O)$$

A component $Comp_1$ is a *refinement* of a component $Comp_2$, denoted by $Comp_2 \sqsubseteq Comp_1$, if $Comp_1$ is a sub-relation of $Comp_2$.

Definition 15. *Component $Comp_1$ is a* **refinement** *of $Comp_2$ if $R_1 = R_2 \wedge O_1 = O_2$ and for any required service $Contr_{R_1}$,*

$$(Contr_{R_1} >> Comp_2) \sqsubseteq (Contr_{R_1} >> Comp_1)$$

We therefore have when $Comp_1$ refines $Comp_2$, then for any given required service $Contr_R$ and a contract a provided service $Contr_O$ as the specification, $Comp_1$ realizes $Contr_O$ with $Contr_R$ if $Comp_2$ realizes $Contr_O$ with $Contr_R$.

Definition 16. *Let $Comp_i = (Contr_i, I_i, O_i, R_i)$, $i = 1, 2$, be two components with contracts $Contr_i = (O_i \cup I_i, \Omega_i, A_i)$. Assume that $I_1 \cap I_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$ and $R_1 \cap R_2 = \emptyset$. The* **composition** *$Comp_1 || Comp_2$ is defined to merge their contracts, output interfaces and input interfaces, and to remove those input interfaces of each component that are matched by the output interfaces in another:*

$$Comp_1 || Comp_2 \stackrel{def}{=} < Contr_1 \| Contr_2, I_1 \cup I_2, O_1 \uplus O_2, R_1 \backslash O_2 \cup R_2 \backslash O_1 >$$

Let $I \stackrel{def}{=} I_1 \cup I_2$, $R \stackrel{def}{=} R_1 \backslash O_2 \cup R_2 \backslash O_1$ and $O \stackrel{def}{=} O_1 \cup O_2$. The composition of $Comp_1$ and $Comp_2$ is defined by

$$\begin{aligned}
[\![Comp_1 \| Comp_2]\!](Contr_R, Contr'_O) \stackrel{def}{=} \; & \exists Contr_{R_1}, Contr'_{O_1}, Contr_{R_2}, Contr'_{O_2} \bullet \\
& [\![Comp_1]\!](Contr_{R_1}, Contr'_{O_1}) \wedge \\
& [\![Comp_2]\!](Contr_{R_2}, C'_{O_2}) \wedge \\
& Contr_{R_1} \backslash (R_1 \backslash O_2) = Contr'_{O_2} \backslash (O_2 \backslash R_1) \wedge \\
& Contr_{R_2} \backslash (R_2 \setminus O_1) = Contr'_{O_1} \backslash (O_1 \setminus R_2) \wedge \\
& Contr_R = Contr_{R_1} \backslash (R_1 \backslash O_2) \| Contr_{R_2} \backslash (R_2 \backslash O_1) \wedge \\
& Contr'_O = Contr'_{O_1} \backslash (R_2 \backslash O_1) \| Contr'_{O_2} \backslash (R_1 \backslash O_2)
\end{aligned}$$

This definition allows an output interface and thus part of provided service of one component to be shared among a number other components. Hiding can be used to *internalize* the part of a provided service of one component that is used in another component: $(Comp_1 \| Comp_2) \backslash (R_1 \cap O_2) \backslash (R_2 \cap O_1)$.

Examples of component specifications and composition can be found in [31]. Client-server systems are often seen as applications in component software. The architecture of such a system is organized as a layered structure and can be model with in our model as shown in the full version [30] of paper [31].

6 Conclusion

We have proposed a classical relational model (rCOS) for component-based and object-oriented development. This model provides a smooth link between component-based design and object-oriented development. It 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 [19]. 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: in order 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.

The semantic model allows us to specify a system at different levels of abstraction. At the requirement, we can specify the functional requirements as use-case operations defined as methods of use-case controller classes. Each of these operations can be abstractly specified as a design in terms object creation, object destruction, and object attributes modification. Objects at these levels can be decomposed latter. These use-case operations can then refinement using the refinement laws for expert pattern, low coupling, high cohesion and attribute encapsulation. For details of requirement analysis and design by refinement, we refer the reader to [26,32]. With the refinement laws, algebraic reasoning is also supported.

This paper not only presents a semantics, but also provides a definitional approach to defining different semantics with different constraints and features.

6.1 Related Work

In the framework of ROOL [8], 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.

There is an increasing amount of research in formal techniques for component-based development, e.g. [7,3,14]. These models are channel-based and process oriented. They can easily related to state machine models or automata and thus existing verification techniques and model checking tools can be readily applied. These models are flexible in describing interactions and coordinations among components due to the fine granularity of the interaction actions, that is channel-based message passing. We are aiming at a definitional semantic model that is easier to be related to software engineering concepts and terminology, such as *provided services*, *required services* and *contracts*, and programming languages, object-oriented languages in particular. We hope that this model will provide effective formal support to model-based development by pattern-guided transformations.

rCOS is motivated by our work on formal support to UML-based software development. We have studied the application of rCOS to support UML-based requirement modelling, analysis and design process. rCOS is used in [32] for formalisation of 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 [28] 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 [27]. Algorithms are also designed for consistency checking and executable code generation from a system model [35]. The technical report [16] presents detailed study of rCOS for object systems with examples and proofs of laws. A case study of the use of the refinement laws in software development is given in [36].

These publications show how rCOS can be used to support engineering methods and processes in software development.

6.2 Future Work

Future work includes the completion of the calculus rCOS for synchronization and concurrency in both object and component systems. This requires the model of components to be extended with the specification of the protocol in the interface, contract of a component. We plan to use traces (or regular expressions of operations for this purpose. The notion of designs of operations of *active* and *reactive* components have to be extended to *reactive designs* to capture synchronization. Consistency between the protocol and the reactive designs have to be checked to avoid from deadlock and divergence.

We will work on case studies to test the theory and the method. We are also interested in a theory of tool integration within this framework.

Acknowledgments

We are grateful to the organizers of FMCO'04 to invite Zhiming Liu to give a talk at the simposium and to the participants of the simposium for their comments. We would like thank Dines Bjorner at Technical University of Denmark, Kung-Kiu Lau at Manchester University, Anders Ravn at 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 Zhiming Liu gave on parts of the works when he visited them. Our UNU-IIST fellows Jing Liu, Xiaojian Liu, Quan Long, Bhim Upadhyaya and Jing Yang contributed to the whole research project. They also read and gave useful comments on this article. Zhiming Liu 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 courses on Software Engineering and System Development with UML for their feedback on the understanding of the use-case driven, incremental and iterative object-oriented development and design patterns. Part of the work was also presented at the Workshop on Predictable Software Component Assembly held in the University of Manchester in 2004, and as an invited talk in Brazilian Symposium on Formal Methods (SBMF 2004). The discussion and comments were very much useful in bringing the presentation of the work into its current form.

References

1. M. Abadi and L. Cardeli. *A Theory of Objects*. Springer-Verlag, 1996.
2. R. Allen and D Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 1997.
3. F. Arbab. Reo: A channel-based coordination of model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
4. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 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. M. Broy and K. Stolen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer, 2001.
8. 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.
9. J. Cheesman and J. Daniels. *UML Components. Component Software Series*. Addison-Wesley, 2001.
10. Y. Chen and J.W. Sanders. The weakest specifunction. *Acta Informatica*, 41(7), 2005.
11. J.K. Filipe. A logic-based formalization for component specification. *Journal of Object Technology*, 1(3):231–248, 2002.
12. 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.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addlison Wesley, 1995.
14. G. Goessler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*.
15. J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.
16. J. He, Z Liu, and X. Li. rCOS: A refinement calculus for object systems. Technical Report 322, UNU/IIST, P.O. Box 3058, Macao SAR China, 2005.
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), LNCS 3302*, pages 415–436, Taiwan, March 2004. Springer.
18. G.T. Heineman and W.T. Councill. *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Wesley, 2001.
19. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
20. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
21. N. Jin and J. He. Resource models and pre-compiler specification for hardware/software. In J.R. Cuellar and Z. Liu, editors, *Proc. 2nd International Conference on Software Engineering and Formal Methods (SEFM'04)*, Beijing, China, 28-30 September, 2004. IEEE Computer Society.
22. C.B. Jones. Process algebra arguments about an object-oriented design notation. In A.W Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice-Hall, 1994.
23. C.B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, 1996.
24. P. Kruchten. *The Rational Unified Process – An Introduction (2nd Edition)*. Addison-Wesly, 2000.
25. C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
26. 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.
27. 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), LNCS 3347*, pages 255–265, Bhubaneswar, India, 2004. Springer.
28. 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.
29. 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.

30. Z. Liu, J. He, and X. Li. Contract-oriented component software development. Technical Report UNU/IIST, Report No 298, 2004. http://www.iist.unu.edu/newrh/III/1/page.html.
31. 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.
32. 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.
33. Z. Liu, J. He, X. Li, and J. Liu. Unifying views of UML. *Electronic Notes of Theoretical Computer Science (ENTCS)*, 101:95–127, 2004.
34. Q. Long, J. He, and Z. Liu. Refactoring and pattern-directed refactoring: A formal perspective. Technical Report 318, UNU-IIST, P.O.Box 3058, Macau, January 2005.
35. Q. Long, Z. Liu, X. Li, and J. He. Consistent code generation from UML models. In *Pro. of Australian Software Engineering Conference (ASWEC'2005)*, pages 168–177, Brisbane, Australia, 2005. IEEE Computer Sciety.
36. Q. Long, Z. Qiu, Z. Liu, L. Shao, and J. He. POST: A case study for rcos incremental development. Technical Report 324, UNU/IIST, P.O. Box 3058, Macao SAR China, 2005.
37. S.J. Mellor and M.J. Balcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley, 2002.
38. B. Meyer. From structured programming to object-oriented design: the road to Eiffel. *Structured Programming*, 10(1):19–39, 1989.
39. B. Meyer. Applying design by contract. *IEEE Computer*, May 1992.
40. B. Meyer. *Object-oriented Software Construction (2nd Edition)*. Prentice Hall PTR, 1997.
41. 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.
42. B. Selic. Using UML for modelling complex real-time systems. In F. Muller and A. Bestavros, editors, *Languages, compilers, and Tools for Embedded Systems, Volume 1474 of Lecture Notes in Computer Science*, pages 250–262. Springer Verlag, 1998.
43. A. Sherif, J. He, A Cavalcanti, and A. Sampaio. A framework for specification and validation of real-time systems using circus actions. In Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing ICTAC 2004. First International Colloquium Guiyang, China, September 2004, Revised Selected Papers. LNCS 3407*, pages 478 – 494. Springer, 2005.
44. M. Smyth. Powerdomain. *Journal of Computer Science and System Sciences*, 16:23–36, 1978.
45. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.
46. J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.
47. J. Yang, Q. Long, Z. Liu, and X. Li. A predicative semantic model for integrating UML models. In Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing – ICTAC 2004, First International Colloquium, Guiyang, China, September 2004, Revised Selected Papers, LNCS 3407*, pages 170–186. Springer, 2005.