Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 365 (2006) 109 – 142
Theoretical Computer Science
www.elsevier.com/locate/tcs

rCOS: A refinement calculus of object systems☆

He Jifeng[a], Xiaoshan Li[b], Zhiming Liu[c,*]

[a] *Software Engineering Institute, East China Normal University, Shanghai, China*
[b] *Faculty of Science and Technology, University of Macau, Macao, China*
[c] *United Nations University, International Institute for Software Technology, P.O. Box 3058, Macao SAR, China*

Abstract

This article presents a mathematical characterization of object-oriented concepts by defining an observation-oriented semantics for a relational object-based language with a rich variety of features including subtypes, visibility, inheritance, type casting, dynamic binding and polymorphism. The language can be used to specify object-oriented designs as well as programs. We present a calculus that supports both structural and behavioural refinement of object-oriented designs. The design calculus is based on the predicate logic in Hoare and He's Unifying Theories of Programming (UTP).

Keywords: Object orientation; Refinement; Semantics; UTP

1. Introduction

Software engineering is mainly concerned with using techniques to systematically develop large and complex program suites. In the search for techniques for making software development more productive and software systems more reliable, object-oriented programming and formal methods are two important but largely independent approaches which have been influential in recent years.

The concept of *objects* is an important concept in software development. Experimental languages of the 1970s provided various definitions of package, cluster, module, etc. They promote modularity and encapsulation, allowing the construction of software components which hide state representations and algorithmic mechanisms from users, and export only pertinent features. This produces components with a level of abstraction by separating the view of what a module does from the details of how it does them. It is clear that certain features of the objects, particularly *inheritance* and the use of *object references* as part of the data stored by an object, could be used to construct large system *incrementally* and efficiently, as well as making it possible to *reuse* objects in different contexts.

It is essential that software engineering is given the same basis in mathematics as other engineering disciplines. There has been good progress, resulting in three main paradigms: model-based, algebraic and process calculi. Practitioners of formal methods and experts in object technology have investigated how formal specification can supplement object-

☆ This is a combination of revised and extended versions of [26,39]. This work is partially supported by the project HighQSoftD funded by Macao Science and Technology Development Fund, the 973 projects 2002CB312001 and 2005CB321904 of the Ministry of Science and Technology of China.

* Corresponding author.
E-mail addresses: jifeng@sei.ecnu.edu.cn (H. Jifeng), xsl@umac.mo (X. Li), Z.Liu@iist.unu.edu (Z. Liu).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.07.034

oriented development [34], and how it may help to clarify the semantics of object-oriented notations and concepts. Examples of such work include the formalization of the OMG's core object model [29] using Z.

Model-based formalisms have been used extensively in conjunction with object-oriented techniques, via languages such as Object-Z [53], VDM++ [17], and methods such as Syntropy [16] which uses the Z notation and Fusion [15] that is based on VDM. Whilst these formalisms are effective at modelling data structures as sets and relations between sets, they are not designed for defining semantics of object-programs and thus do not deal with more sophisticated object-oriented mechanisms of object-oriented programming languages, such as dynamic binding and polymorphism.

Cavalcanti and Naumann defined an object-oriented programming language, called *ROOL*, with subtypes and polymorphism [13,45] using predicate transformers. Sekerinski [51,43] defined a rich object-oriented language by using a type system with subtyping and predicate transformers. However, neither reference types nor mutual dependency between classes are within the scope of these approaches. Because of the complex flow of control, it is not feasible to calculate the weakest precondition of an object-oriented program for a given postcondition. Thus, semantic proofs of refinement rules in ROOL are quite hard and complex even without references. Without the inclusion of reference types, some interesting refinement rules cannot be proved [10]. America and de Boer have given a logic for the parallel language POOL [4]. It applies to imperative programs with object sharing, but without subtyping and method overriding. Abadi and Leino have defined an axiomatic semantics for an imperative, object-oriented language with object sharing [1], but it does not permit recursive object types. Poetzsch-Heffter and Müller have defined a Hoare-style logic for object-oriented programs that relaxes many of the previous restrictions [47]. However, the specification of a method in the Poetzsch-Heffter and Müller logic is derived from the method's known implementation [36]. Leino has presented a logic in [36] with imperative features, subtyping, and recursive types. It allows the specification of methods, but inheritance is restricted and visibility is not considered.

In this article, we present part of a model and a refinement calculus (named as rCOS) for component and object systems. We focus on a mathematical characterization of object-oriented concepts, and provide a proper semantic basis essential for ensuring the correctness of programs and for developing tool support for formal techniques. We define an object-oriented language with subtypes, visibility, reference types, inheritance, type casting, dynamic binding and polymorphism. The language is similar to Java and C++. It has been used to develop meaningful case studies and to capture some of the central difficulties in modelling object-oriented designs and programs. However, we will not consider garbage collection, attribute hiding, multiple inheritance and exception handling.

rCOS is *class-based* and refinement is about making *correct* changes to the structure, methods of classes and the main program. The logic of rCOS is a *conservative extension* of standard predicate logic [28]. In our model, both commands and class declarations are identified as predicates whose alphabets include logic variables representing the initial and final values of program variables, as well as those variables representing the contextual information of classes and their links. A variable of a built-in primitive type, such as the type `Int` of integers, stores data of the corresponding type whereas a variable of an object type holds the identity or reference and the current type information of an object as its value. We define the traditional programming constructs, such as conditional, sequential composition and recursion, in exactly the same way as their counterparts in an imperative programming language without reference types. This makes our approach more accessible to users who are already familiar with the existing imperative languages. All the laws about imperative commands remain valid without the need of reproving.

Another contribution of this work is to relate the notions of refinement and data refinement [27,44,6] in imperative programming to refactorings [19] and object-oriented design patterns for *responsibility assignments* [20,35]. Initial attempts to formalize refactorings in [50,54] are advanced by providing a formal justification of the soundness of the refactoring rules. The theories in [13,36,5,10] on object-oriented refinement are also advanced by dealing with large scale object-oriented program refinement with refactorings, functionality delegation, data encapsulation and class decomposition. Our refinement rules have been strongly motivated by the formal treatment of transformations of multi-view models, such as UML [40,41] and rational unified process [31,33].

For simplicity, we do not consider attribute domain redefinition or attribute hiding. Our interest is in program requirement specification, design, verification and refinement; attribute domain redefinition and attribute hiding are language facilities mainly used for programming around defects in requirement specification or for the reuse of classes in a way that was not originally intended. For similar reasons, we ignore interfaces, throws clauses, concurrency, method name overloading, inner classes and method pointers. Some issues, such as concurrency and exception handling will be treated in a planned extension of this work.

The notion of *designs* in Unifying Theories of Programming [28] is introduced in Section 2. In Section 3, we define the syntax of rCOS. The semantics is given in Section 4, with a discussion about behavioural refinement of object-oriented designs (commands) under the same class declarations. The laws just extend the laws in UTP to object-oriented commands. In Section 5, we define a notion of object-oriented refinement that allows us to (i) refine both the class declarations and main methods and (ii) explore structural refinement. In Section 6, we present refinement laws that capture the essence of object-oriented design and programming. We provide proofs for some of these laws. The semantic definition of rCOS is essential for the precise justification of these laws. We will draw conclusions and discuss related and future work in Section 7.

2. Semantic basis

The execution of a program is modelled as a relation between *program states*. Here, the concept of state is more general than in a sequential language. For example, for a terminating sequential program, we are only interested in the initial inputs and final outputs. For a program which may not terminate, we need an observable by which we can describe whether or not the program terminates for its input. For concurrent and communicating programs, we observe the possible *traces* of interactions, *divergencies* and *refusals*, in order to verify if a program is deadlock free and livelock free. For real-time programs, we might observe time. Identifying what to observe in systems is one of the core ideas of UTP.

For a program P, we call what is to be observed the *observables* or *alphabet* of P, denoted by $\alpha(P)$ or 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*. Observables need not to appear in the program text but they are 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*, represented as $Pre \vdash Post$, with free variables in α. It is generally interpreted as if the value of observables satisfies the *precondition* Pre at the beginning of the execution, the execution will *generate* observables satisfying the *postcondition* $Post$.

2.1. Programs as designs

This subsection summarizes how the basic programming constructs can be defined as designs. For further details we refer the reader to the book on UTP [28].

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 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 $in\alpha \cup out\alpha \cup \{ok, ok'\}$, while a design is of the form

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

where

- p is a predicate over $in\alpha$ and R is a predicate over $in\alpha \cup out\alpha$,
- 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 initiation and termination of the program, respectively; they do not appear in the program texts.

The design represents a *contract* between the "user" and the program such that if the program has 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

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

where $\underline{w}$ contains all variables in $in\alpha$ except for those in β.

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

Fig. 1. Basic commands as designs.

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

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

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

$$D_1 \sqcap D_2 \stackrel{\text{def}}{=} D_1 \vee D_2, \qquad D_1 \sqcup D_2 \stackrel{\text{def}}{=} D_1 \wedge D_2.$$

- `while` b `do` D, also denoted by $b * c$, is defined as the worst fixed point of the relation expression $((D; X) \lhd b \rhd skip)$, where the worst fixed point of $F(X)$ is the lest upper bound of $\{F^i(true) | i = 0, 1, \ldots\}$.

Some primitive programming commands as framed designs are given in Table of Fig. 1. Composite statements are then defined by semantics operations on designs:

In general, when giving a semantics, preconditions are usually strengthened with some *well-definedness* conditions of the commands. Thus, the semantics of a program or command c is generally of the form

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

where $Spec$ is a design and $D(c)$ is the well-definedness condition of c. Well definedness may be dynamic.

Strengthening preconditions by conjoining well-definedness conditions allows us to modify an ill-defined command to a well-formed one by means of a refinement. This approach supports incremental development as most cases of ill-definedness commands are due to insufficient data or services. The addition of data, services and components can thus be considered as refinements in our framework.

In this article, variables capturing aspects of dynamic typing, visibility, etc, are used to define the semantics of object-oriented programs. This ensures that the logic of rCOS is a conservative extension to that used for imperative programs. All the laws about imperative commands remain valid without the need of revision.

2.2. Refinement of designs

The refinement relation between designs is defined to be logic implication.

Definition 1. 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, that is

$$\forall x_1, \ldots, x_n, x'_1, \ldots, x'_n, ok, ok' \cdot (P_2 \Rightarrow P_1),$$

where $x_1, \ldots, x_n, x'_1, \ldots, x'_n$ are the variables in α. $D_1 = D_2$ if $D_1 \sqsubseteq D_2 \wedge D_2 \sqsubseteq D_1$.

If the two designs do not have the same alphabet, we can use data refinement to relate their state spaces, as well as their behaviour

Definition 2. Let $\rho(\alpha_2, \alpha_1)$ be a many to one mapping from the state space of α_2 to the state space of 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

$$((true \vdash \rho(\alpha_2, \alpha_1'));\ P_1) \sqsubseteq (P_2;\ (true \vdash \rho(\alpha_2, \alpha_1'))).$$

Notice that both sides of the above refinement have the same alphabet $\alpha_1 \cup \alpha_2$.

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 [28].

The following theorem establish that designs can be used for defining a semantics of programs.

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

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

The proof can be found in [28].

3. Syntax of rCOS

In rCOS, an object system (or program) S is of the form $Cdecls \bullet Main$, consisting of class declaration section *Cdecls* and a main method *Main*. The main method is a pair $(extvar, c)$, where *extvar* is a finite set of *external variables* and c is a command. 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 form

$$\begin{array}{l}
[\texttt{private}]\ \texttt{class}\ M[\texttt{extends}\ N]\{ \\
\quad \texttt{private}\ \ T_{11}a_{11} = d_{11}, \ldots, T_{1m_1}a_{1m_1} = d_{1m_1}; \\
\quad \texttt{protected}\ T_{21}a_{21} = d_{21}, \ldots, T_{2m_2}a_{2m_2} = d_{2m_2}; \\
\quad \texttt{public}\ \ T_{31}a_{31} = d_{31}, \ldots, T_{3m_3}a_{3m_3} = d_{3m_3}; \\
\quad \texttt{method}\ \ m_1(\underline{T}_{11}\underline{x}_1; \underline{T}_{12}\underline{y}_1; \underline{T}_{13}\underline{z}_1)\{c_1\}; \\
\qquad\qquad \cdots; \\
\qquad\qquad m_\ell(\underline{T}_{\ell 1}\underline{x}_\ell; \underline{T}_{\ell 2}\underline{y}_\ell; \underline{T}_{\ell 3}\underline{z}_\ell)\{c_\ell\} \\
\quad \}
\end{array}$$

where

- A class can be declared as *private* or *public* (the default is public). The class section is a *Java-like package* and *Main* an application program using the package. Only a public class or a primitive type can be used in the external variable declarations of *Main*.
- N and M are distinct names of classes, and N is called the *direct superclass* of M.
- Attributes annotated with *private*, *protected* and *public* are private, protected and public attributes to the class, respectively. The types and initial values of attributes are given in the declaration.
- A *method* declaration declares the method, its value parameters $(\underline{T}_{i1}\ \underline{x}_i)$, result parameters $(\underline{T}_{i2}\ \underline{y}_i)$, value–result parameters $(\underline{T}_{i3}\ z_i)$ and bodies (c_i).

We use the Java convention, and assume that an attribute is *protected* when it is not tagged with *private* or *public*. We assume, for simplicity, that all methods are public and can be inherited by a subclass.

Symbols: We assume the following disjoint infinite sets of symbols:

- *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.
- *VNAME* denotes the set of *simple variables* names. We use x, y, and z, etc. for simple variable names.

3.1. Commands

rCOS supports typical object-oriented programming constructs. It also provides some commands for the purpose of specification and refinement. The syntax of rCOS commands is:

$$c ::= skip|chaos|\ \texttt{var}\ Tx[= e]|\ \texttt{end}\ x|c; c|c \lhd b \rhd c|c \sqcap c|b * c|le.m(\underline{e}; \underline{e}; \underline{e})|le := e|C.new(le),$$

where b is a Boolean expression, e a general expression, $\underline{e}$ a list of expressions and le an expression which may appear on the left-hand side of an assignment, obeying the form

$$le ::= x|self|le.a,$$

where

- x is a simple variable and a an attribute.
- $le.m(\underline{ve}; \underline{re}; \underline{vre})$ denotes a method m call within the object le. Expression lists $\underline{ve}$, $\underline{re}$ and $\underline{vre}$ are the actual value input parameters, result parameters and actual value–result parameters, respectively.
- The command $C.new(le)$ creates a new object of class C whose attributes have the initial values as declared in C and attaches the new object to le. When C has attributes whose types are classes, we allow nested object creation. For example, if $D\ a$ is an attribute of C, $C.new(le)[D.new(a)]$ creates a new object of class C and a new object of D attached to C's attribute a.
- Command $\texttt{var}\ Tx = e$ declares a local variable x of type T with an initial value e; $\texttt{end}\ x$ ends the scope of the local variable x.

A local variable can be declared a number of times with different types and values before it is undeclared. Thus, a local variable x may have a sequence of declared types and it may takes sequence of values.

3.2. Expressions

Expressions, which can appear on the right-hand side of an assignment, are constructed according to the rules below:

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

where *null* represents the special value (or object), *self* is used to denote the active object in the current scope (some object-oriented languages use *this*), $e.a$ is the attribute a of e, $(C)e$ is type casting, and f is a built-in operation for a built-in primitive type.

4. Semantics

We now show how to use the basic model of the UTP to define the semantics of rCOS. We use $[\![\mathcal{E}]\!]$ to denote the semantics of an element $\mathcal{E}$, such as a command and a class declaration. The semantics takes into account the following features:

- A program operates not only on variables of primitive types, such as integers and Booleans, but also on variables of *object reference types*.
- To protect attributes from illegal accesses, the model addresses the problem of *visibility*.
- An object can be associated with any subclass of its original declaration. To validate expressions and commands in a dynamic binding environment, the model keeps track of the *current type* of each object.
- The dynamic type M of an object can be cast up to any superclass N and later cast down to any class which is a subclass of N and a superclass of M (or M itself). We record both the *cast type* N and the current type M of the object.

4.1. Structure, value and object

The class declaration section *Cdecls* of a program defines the types (value space) and static structure of the program.
Structure: We introduce the following *structural* variables:

- $pricname = \{\texttt{private}\ C | C$ is declared in *Cdecls*$\}$. We use *pubcname* to record the sets of names of the public classes declared in *Cdecls*. Let *cname* be the union of these two sets.

- *superclass*: the partial function

$$\{M \mapsto N | \texttt{[private] class } M \texttt{ extends } N \text{ is declared in } Cdecls\}.$$

This function defines that N is a direct superclass of M. We define the general superclass class relation $\succ$ to be the transitive closure of *superclass*, and $N \succcurlyeq M$ if $N \succ M$ or $N = M$.

- *pri*, *prot*, and *pub*: these variables associate each class name $C \in cname$ to its private attributes $pri(C)$, protected attributes $prot(C)$, and public attributes $pub(C)$, respectively:

$$\begin{aligned} pri(C) &\stackrel{\text{def}}{=} \{\langle a : T, d\rangle | Ta = d \text{is a private attribute of} C\},\\ prot(C) &\stackrel{\text{def}}{=} \{\langle a : T, d\rangle | Ta = d \text{is a protected attribute of} C\},\\ pub(C) &\stackrel{\text{def}}{=} \{\langle a : T, d\rangle | Ta = d \text{is a public attribute of} C\}. \end{aligned}$$

We define the following functions over attributes:

(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 all the attributes that C inherited from its superclasses.
(3) $ATTR(Cdecls)$ denotes the set of $\{C.a | C \in cname \wedge a \in Attr(C)\}$
(4) $init(C.a)$ denotes the initial value of attribute a of C.
(5) $dtype(C.a)$ denotes the *declared type* T if $\langle a : T, d\rangle \in Attr(C)$.
(6) $ATTR(C)$ is the set of all attributes that are associated to class C: it is the smallest set such that:
   (a) $Attr(C) \subseteq ATTR(C)$.
   (b) $Attr(dtype(N.a)) \subseteq ATTR(C)$ if $N.a \in ATTR(C)$ and $dtype(N.a)$ is a class in *cname*.

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

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

The set of the above structural variables is denoted by Ω_{Cdecls}. A class declaration is a command that modifies these structural variables. However, the values of these variables remain unchanged during execution of the main method.

Attribute expression: The set $eATTR(C)$ of *attribute expressions* of class C is defined inductively below:

(1) $\varepsilon \in eATTR(C)$;
(2) $C.a \in eATTR(C)$ for each attribute a of C;
(3) if $C.a \in eATTR(C)$ and $dtype(C.a) \in cname$, then $dtype(C.a).b \in eATTR(C)$ for any $b \in Attr(dtype(C.a))$;
(4) if $e_i \in eATTR(C)$ for $i = 1, \ldots, n$, $dtype(e_i)$ are built-in primitive types and expression $f(x_1 : dtype(e_1), \ldots, x_n : dtype(e_n))$ is well-defined on these primitive types, then $f(e_1, \ldots, e_n) \in eATTR(C)$.

Value and object: We assume a set $\mathcal{T}$ of *built-in primitive types*. We also assume an infinite set *REF* of *object identities* (or *references*), with $null \in REF$. 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 *VAL* be the set of values

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

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 3. 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 name;
- σ is called the *state* of o, denoted by $state(o)$, and it is a mapping that assigns each $a \in Attr(C)$ 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 $\sigma(a)$.

We extend *equality* to a relation over both values and objects

$$(v_1 = v_2) \stackrel{\text{def}}{=} \left(\begin{array}{l} (type(v_1) = type(v_2) \wedge\\ (type(v_1) \in \mathcal{T} \wedge (v_1 = v_2)) \vee\\ \forall\, a \in Attr(type(v_1)) \cdot (v_1.a = v_2.a) \end{array}\right).$$

This equality ignores object references, but relating underlying primitive attributes.

Some notations: Let $\mathcal{O}$ be the set of all objects, including *null*. The following notations are employed:

- For sets S and S_1, $S_1 \succcurlyeq S$ is the set difference removing elements in S_1 from S. Let $\succcurlyeq$ have higher associativity [1] than the normal set operators like $\cup$ and $\cap$.
- For a mapping $f : D \longrightarrow E$, $d \in D$ and $r \in E$,

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

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

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

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

$$S \uplus \{\langle r, M, \sigma \rangle\} \stackrel{\text{def}}{=} \{o | ref(o) = r\} \succcurlyeq S \cup \{\langle r, M, \sigma \rangle\},$$
$$ref(S) \stackrel{\text{def}}{=} \{r | r = ref(o), o \in S\}.$$

For a given class declaration section *Cdecls*, Σ_{Cdecls}, called the *object space* of *Cdecls*, denotes the set of all objects declared in *Cdecls*. The pair $(\Omega_{Cdecls}, \Sigma_{Cdecls})$ is called a *program context* and denote it by Ξ_{Cdecls}. When there is no confusion, we omit the subscript *Cdecls*. All dynamic semantic definitions are given under a fixed 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 produced by execution of c in the context Ξ.

4.2. Static semantics

We treat each class declaration as a command and its semantics is defined as a design. A class declaration changes the values of the structural variables *pricname*, *pubcname*, *cname*, *superclass*, *pri*, *prot*, *pub* and *op*. We first define the well-definedness of a class declaration.

Definition 4. A class declaration *cdecl* is *well-defined* if the following conditions hold:
(1) M has not been declared before: $M \notin cname$.
(2) N and M are distinct: $N \neq M$.
(3) The attribute names in the class are distinct.
(4) The method names in the class are distinct.
(5) The parameters of every method are distinct.
We use $\mathcal{D}(cdecl)$ to denote the conjunction of the above conditions for the class declaration of *cdecl*.

A well-defined private class declaration for M with a superclass N will modify the structural variables:

$$[\![cdecl]\!] \stackrel{\text{def}}{=} \{pricname, pubcname, superclass, pri, prot, pub, op\} : \mathcal{D}(cdecl) \vdash$$
$$\begin{pmatrix} \quad modifyPriCname \wedge modifyPubCname \wedge modifySuper \\ \wedge \quad modifyPri \wedge modifyProt \wedge modidyPub \wedge modifyOp \end{pmatrix},$$

where

$$modifyPriCname \stackrel{\text{def}}{=} pricname' = pricname \cup \{M\},$$
$$modifyPubCname \stackrel{\text{def}}{=} pubcname' = pubcname,$$
$$modifySuper \stackrel{\text{def}}{=} superclass' = superclass \oplus \{M \mapsto N\},$$

[1] This is the purpose of using this "strange" notation for set difference.

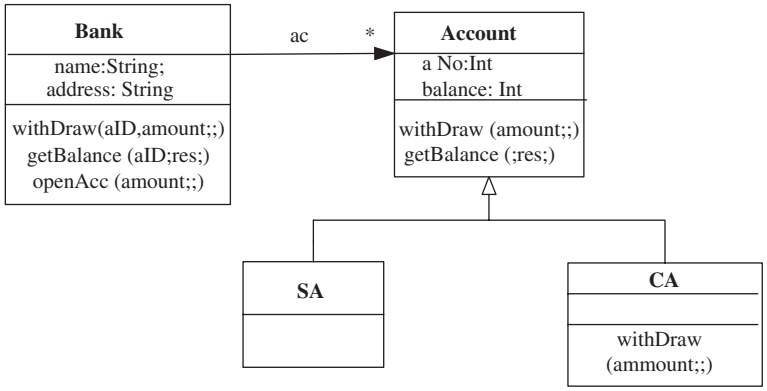

Fig. 2. A bank system.

$$modifyPri \stackrel{\text{def}}{=} pri' = pri \oplus \{M \mapsto \{\langle a_{11} : T_{11}, d_{11}\rangle, \ldots, \langle a_{1m_1} : T_{1m_1}, d_{1m_1}\rangle\}\},$$

$$modifyProt \stackrel{\text{def}}{=} prot' = prot \oplus \{M \mapsto \{\langle a_{21} : T_{21}, d_{21}\rangle, \ldots, \langle a_{2m_2} : T_{2m_2}, d_{2m_2}\rangle\}\},$$

$$modifyPub \stackrel{\text{def}}{=} pub' = pub \oplus \{M \mapsto \{\langle a_{31} : T_{31}, d_{31}\rangle, \ldots, \langle a_{3m_3} : T_{3m_3}, d_{3m_3}\rangle\}\},$$

$$\begin{aligned} modifyOp \stackrel{\text{def}}{=}\ & op' = op \oplus \{M \mapsto \{m_1 \mapsto (\langle \underline{x}_1 : \underline{T}_{11}; \underline{y}_1 : \underline{T}_{12}; \underline{z}_1 : \underline{T}_{13}\rangle, c_1), \ldots, \\ & m_\ell \mapsto (\langle \underline{x}_\ell : \underline{T}_{\ell 1}; \underline{y}_\ell : \underline{T}_{\ell 2}; \underline{z}_\ell : \underline{T}_{\ell 3}\rangle, c_\ell)\}\}. \end{aligned}$$

We can similarly define a class declaration for the cases when the class M is declared as a public class and when it is not declared as a subclass of another.

Definition 5. Let $Cdecls \equiv (cdecl_1; \ldots; cdecl_n)$ be a class declaration section. Its *semantics* is defined by the sequential composition of the designs of the individual class declarations starting with all structural variables initialized to the empty set

$$[\![Cdecls]\!] \stackrel{\text{def}}{=} Empty; [\![cdecl_1]\!]; \ldots; [\![cdecl_n]\!],$$

where

$$Empty \stackrel{\text{def}}{=} true \vdash \left(\begin{array}{l} pricname' = \emptyset \wedge pubcname' = \emptyset \wedge superclass' = \emptyset \\ \wedge\ pri' = \emptyset \wedge prot' = \emptyset \wedge pub' = emptyset \wedge op' = \emptyset \end{array} \right).$$

Definition 6. 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 $N \in cname$;
(3) any type used declarations of attributes and parameters is either a built-in primitive 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 in a subclass;
(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 method are distinct, respectively.

A well-defined rCOS declaration section corresponds to a UML [9] class diagram. For related work on formal support to UML-based development, we refer to our work in [40,41,58].

Example 1. Consider a bank system illustrated by the UML class diagram in Fig. 2. *Account* has two subclasses: a current account *CA* and a savings account *SA*.

The declaration of public class *Bank* has three attributes: *name* and *address* are of primitive types, say `String`, and association *ac* which is of the power type $\mathbb{P}$`Account` of class type `Account`. A specification of class declaration for *Bank* is given below:

`class` *Bank* {

`private` : `String` *name*, *address*;

`private` : $\mathbb{P}$`Account` $ac = \emptyset$;

`method` : *withDraw*(`Int` *aID*, `Int` *amount*){

$\exists a \in ac \cdot a.aNo = aID \vdash$

$$\bigvee_{a \in ac \wedge a.aNo = aId} (a.balance' = a.balance - amount)$$

};

getBalance(`Int` *aID*; `Int` *res*){

$\exists ac \cdot a.aNo = aID \vdash$

$$\bigvee_{a \in ac \wedge a.aNo = aId} (res' = a.balance)$$

}

openAcc(`Int` *amount*){

`var` `Account` $a = null$;

$$\exists n, \forall b \in ac \cdot n \neq b.aNo \wedge \begin{pmatrix} Account.new(a); \\ a.aNo := n; \\ a.balance := amount \end{pmatrix}$$

}

}

Note designs can appear in the body of a method. We need to make a few remarks about the above specification:

(1) At the level of specification of the methods, we assume the attributes of class *Account* are all public and can be directly referred in the specification of the methods of call *Bank*.
(2) In a later design stage, the specification of these methods are refined into statement in which invocation of methods of *Account* are allowed, and after such refinements, the attributes of *Account* can be encapsulated and become protected.
(3) To refine the specification of method *openAcc*, we need to add a method, say named by *openAc*, that implements the code in the big brackets.

The declaration of class *Account*, denoted by *declAccount*, is written as follows:

`private class` *Account* {

`protected` : `Int` $aNo = 0$, `Int` $balance = 0$;

`method` : *getBalance*($\emptyset$; `Int` b; $\emptyset$){$b := balance$};

withDraw(`Int` x; $\emptyset$; $\emptyset$){$balance \geqslant x \vdash balance' = balance - x$}

}

The declaration *declCA* of *CA* is given as

`private class` *CA* `extends` *Account* {

`method` : *withDraw*(`Int` x; $\emptyset$; $\emptyset$){$balance := balance - x$}

}

We can write the declarations of *SA* (in which method *withDraw* is inherited from *Account*) and *Bank* (which has a set of accounts associated with it) in a similar way.

It is easy to see that both *declAccount* and *declCA* are well-formed. The semantics of *declAccount* is defined by the following design, where unchanged variables are omitted:

$$[\![declAccount]\!] = true \vdash \left(\begin{array}{l} pricname' = \{Account\} \cup pricname \\ \wedge\ prot' = prot \oplus \{Account \mapsto \{\langle aNo : \texttt{Int},\ 0\rangle, \langle balance : \texttt{Int},\ 0\rangle\}\} \\ \wedge\ op' = op \oplus \{Account \mapsto \{getBalance \mapsto (\langle\emptyset; b : \texttt{Int}; \emptyset\rangle, b := balance), \\ \qquad\qquad withDraw \mapsto (\langle x : \texttt{Int}; \emptyset; \emptyset\rangle, \\ \qquad\qquad\quad balance \geqslant x \vdash balance' = balance - x)\}\} \end{array} \right).$$

The semantics of *declCA* is the following:

$$[\![declCA]\!] = true \vdash \left(\begin{array}{l} pricname' = \{CA\} \cup pricname \\ \wedge\ op' = op \oplus \{CA \mapsto \{withDraw \mapsto \\ (\langle x : \texttt{Int}; \emptyset; \emptyset\rangle, balance := balance - x)\}\} \end{array} \right).$$

The semantics of *declSA* and *declBank* for classes *SA* and *Bank* can be defined in the same way, but with *Bank* declared as public class. Their composition

$$[\![declAccount; declCA; declSA; declBank]\!]$$

combines the class names, attributes and methods together. The composition is well-defined.

4.3. Dynamic variables

Now consider the variables that can be changed during program execution.

System configuration: First, we introduce a variable Π whose value is the set of objects created so far. We call Π the *current configuration* [46]. During the execution of the program, Π takes a value in the powerset 2^{Σ} that satisfies the following conditions:

(1) *objects in Π are complete*: if $o \in \Pi$ and $a \in Attr(type(o))$ with a class type, then $o.a$ is either *null* or there is an object $o_1 \in \Pi$ and $ref(o.a) = ref(o_1)$ and
(2) *objects are uniquely identified by their references*: for any objects o_1 and o_2 in Π if $ref(o_1) = ref(o_2)$ then:
   (a) $type(o_1) = type(o_2)$ and
   (b) $ref(state(o_1)) = ref(state(o_2))$, where for each $a : T \in Attr(type(o))$

$$ref(state(o))(a) \stackrel{\text{def}}{=} \begin{cases} ref(o.a) & \text{if } T \in cname, \\ o.a & \text{if } T \in \mathcal{T}. \end{cases}$$

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 $extvar = \{x_1 : T_1, \ldots, x_k : T_k\}$ 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 $dtype(x_i)$. A declared type is either a built-in primitive type or a public class in *pubcname*. Their values can be modified by methods and commands of the main method containing them.

Local variables: A set *localvar* identifies the local variables which are declared by local variable declaration commands. This set includes *self* (whose value represents the current active object), and parameters of methods. The sets *localvar* and *extvar* are disjoint.

Method calls may be nested. Thus, *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 represented as $(x : \langle T_1, \ldots, T_n\rangle)$. We use *TypeSeq* to denote the sequence of types of x, with T_1 being the most recently declared type $dtype(x_i)$.

We use $\overline{x}$ 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 $x : T \in extvar$.

Visibility: We introduce a variable *visibleattr* to hold the set of attributes which are visible to the command under execution. The value of *visibleattr* defines the current execution environment. A method of an object o sets *visibleattr*

to $Attr(o)$ (the attributes of the current type of o) which including all the declared attributes of the class, the protected and public attributes of its superclasses and all public attributes of public classes; and the method resets *visibleattr* to the global environment (consisting of all the public attributes of the public classes) when exit its execution. Notice that the value space of *visibleattr* is the powerset of $\{C.a | C \in CNAME, a \in ANAME\}$.

We use

- *var* to denote the union of *extvar* and *localvar*,
- *VAR* is the set of *dynamic variables* consisting of the variables in *var* plus Π and *visibleattr*,
- *internalvar* is the set of elements of *VAR* excluding those of *extvar*.

4.4. Dynamic states

Definition 7. 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 $type(o) \preccurlyeq type(v)$.

  This attachment of an object o to a variable x provides the information about type casting: $type(o)$ is the *current* (base) type of x, denoted as $atype(x)$, and $type(v)$ is the *cast 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 $ref(\Gamma_i(x)) = ref(o_i)$, where $1 \leqslant i \leqslant 2$, then $o_1 = o_2$ and $type(\Gamma_1(x)) = type(\Gamma_2(x))$.

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 $ref(\Gamma(x)) = ref(o)$, $o \in \Gamma(\Pi\downarrow_V)$;
(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 to the external variables *extvar* and project Π onto these variables, we obtain an *external state* in which all objects in the system configuration are attached to variables.

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, i.e. e is x, where $x \in var$, or
(2) $e \equiv self.a$ and there is a class name $N \in cname$ such that $N \succcurlyeq atype(self)$ 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 $N \succcurlyeq type(e_1)$ 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 of D which are inherited or overwritten 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 cast type, i.e. the type of the expression itself.

4.5. 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. Only well-defined expressions are evaluated. Well-definedness conditions can be static and dynamic. The evaluation results of expressions are given in table of Fig. 3.

4.6. Semantics of commands

An important aspect of an execution of an object-oriented program is the attachment of objects to program variables (or entities [42]). An attachment is made by an assignment, the creation of an object or passing a parameter in a method

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

Fig. 3. Evaluation of expressions.

invocation. With the approach of UTP, these different cases are unified as an assignment of a value to a program variable. All other programming constructs are defined in exactly the same way as their counter parts in a procedural language. We only define the object-oriented commands. The definition of other commands remains the same as in an imperative language. 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 $\mathcal{D}(c) \Rightarrow (p \vdash R)$.

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

$$\mathcal{D}(le := e) \stackrel{\text{def}}{=} \mathcal{D}(le) \wedge \mathcal{D}(e) \wedge type(e) \in cname \Rightarrow type(e) \preccurlyeq dtype(le)).$$

Notice that this definition requires *dynamic type matching*. In fact the semantics ensures that if $dtype(e) \preccurlyeq dtype(le)$ then $type(e) \preccurlyeq dtype(le)$. When the value of e is an object $\mathcal{D}(le := e)$ ensures that $atype(e) \preccurlyeq dtype(le)$.

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). This can be done 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.

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

As we do not allow attribute hiding or redefinition in subclasses, an assignment to a simple variable does not have side-effect. 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, $C_2 \preccurlyeq C_1$ and $a : \texttt{Int}$ is protected attribute of C_1. These assumptions make the theory simpler than alternative Hoare-logic based semantics, e.g. [46].

The second case is the modification of 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. All variables attached to the reference of this object are updated:

$$[\![le.a := e]\!] \stackrel{\text{def}}{=} \{\Pi(dtype(le))\} : \mathcal{D}(le.a := e) \vdash \begin{pmatrix} \Pi(dtype(le))' = \Pi(dtype(le)) \uplus \\ \{o \oplus \{a \mapsto value(e)\} | o \in \Pi \wedge ref(o) = ref(le)\} \end{pmatrix}.$$

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. $x.d.a := 4$ changes the 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 the state $x = \langle r_1, C, \{d \mapsto r_2\}\rangle$, where x is as before but the underlying reference r_2 is modified and it is now the identity of the object $\langle r_2, D, \{a \mapsto 4\}\rangle$. This semantic definition also shows that an assignment can have side effects.

Law 1. $(le_1 := e_1; le_2 := e_2) = (le_2 := e_2; le_1 := e_1)$, *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 either le_1 or le_2 is 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

$$\mathcal{D}(C.new(le)) \stackrel{\text{def}}{=} C \in cname \wedge \mathcal{D}(le) \wedge dtype(le) \succcurlyeq C.$$

The command creates a new object, attaches the object to le and sets the initial values of the attributes of class C to those of object le.

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

where

$$AddNew(C, r) \stackrel{\text{def}}{=} \Pi(C)' = \Pi(C) \cup \{\langle r, C, \{a_i \mapsto init(C.a_i)\}\rangle | a_i \in Attr(C)\},$$

$$Modify(le) \stackrel{\text{def}}{=} \begin{pmatrix} \overline{le}' = \langle r, C\rangle \cdot tail(\overline{le}) \wedge \\ TypeSeq'(le) = \langle C\rangle \cdot tail(TypeSeq(le)). \end{pmatrix}.$$

Here 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 2. $C_1.new(x); C_2.new(y) = C_2.new(y); C_1.new(x)$, provided x and y are distinct.

Law 3. 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 $\texttt{var}\ Tx = e$ declares a variable and initializes it:

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

where

$$\mathcal{D}(\texttt{var}\ T\,x = e) \stackrel{\text{def}}{=} (x \in localvar) \wedge \mathcal{D}(e) \wedge type(e) \notin \mathcal{T} \Rightarrow type(e) \preccurlyeq T.$$

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

Command $\texttt{end}\ x$ terminates the block (i.e. the current scope) of variable x:

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

where $\mathcal{D}(\texttt{end}\ x) \stackrel{\text{def}}{=} x \in localvar$. Please refer to [28] for the algebraic laws of declaration and undeclaration.

Method call: For a method signature $m(T_1x; T_2y; T_3z)$, 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 $m \mapsto (T_1x; T_2y; T_3z, c)$ is in the casted type $type(le)$ of le:

$$\begin{aligned}\mathcal{D}(le.m(ve; re; vre)) \stackrel{\text{def}}{=} \quad & \mathcal{D}(le) \wedge type(le) \in cname \wedge (le \neq null)\\ \wedge \quad & \exists N \in cname \cdot (N \succcurlyeq type(le)\\ \wedge \quad & \exists (m \mapsto (T_1x; T_2y; T_3z, c_1)) \in op(N)).\end{aligned}$$

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 $m()$. Before termination, the value of the result and value–result parameters of m are passed back to the actual parameters r and vr.

$$\begin{aligned}[\![le.m(ve; re; vre)]\!] \stackrel{\text{def}}{=} \quad & (\mathcal{D}(le.m(ve; re; vre)) \Rightarrow \exists C \in cname \cdot (atype(le) = C)\\ \wedge \quad & \left(\begin{array}{l}[\![\texttt{var}\ T_1\ x = ve, T_2y, T_3z = vre]\!];\\ [\![\texttt{var}\ C\ self = le]\!];\\ [\![Execute(C.m)]\!]; [\![re, vre := y, z]\!];\\ [\![\texttt{end}\ self, x, y, z]\!]\end{array}\right),\end{aligned}$$

where $Execute(M.m)$ sets the execution environment, then executes the body and finally resets the environment. This is formalized by considering the following cases:

Case 1: If $m(T_1x; T_2y; T_3z)$ is not declared in C but in a superclass of C, i.e. there exists a command c such that $(m \mapsto (T_1x; T_2y; T_3z, c_1)) \in op(N)$ for some $N \succcurlyeq C$, then

$$Execute(C.m) \stackrel{\text{def}}{=} Execute(M.m),$$

where $M = superclass(C)$ is the direct superclass of C.

Case 2: If $m(T_1x; T_2y; T_3z)$ is declared in class C itself, i.e. there is a command c such that $(m \mapsto (T_1x; T_2y; T_3z, c_1)) \in op(C)$, then

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

where

- $body(C.m)$ is the body c of the method being called.
- The design $Set(C)$ determines those attributes visible to class M. *Reset* resets the environment to the set of variables that are accessible by the main program:

$$\begin{aligned}&Set(C) \stackrel{\text{def}}{=} \{\texttt{visibleattr}\} : true \vdash\\ &\texttt{visibleattr}' = \left(\begin{array}{l}\{C.a | a \in pri(C)\} \cup\\ \bigcup_{C \preccurlyeq N} \{N.a | a \in prot(N) \cup pub(N)\} \cup\\ \bigcup_{N \in pubcname} \{N.a | a \in pub(N)\}\end{array}\right),\end{aligned}$$

$$\begin{aligned}&Reset \stackrel{\text{def}}{=} \{\texttt{visibleattr}\} : true \vdash\\ &\qquad \texttt{visibleattr}' = \bigcup_{N \in pubcname} \{N.a | a \in pub(N)\}.\end{aligned}$$

  Set and *Reset* are used to ensure data encapsulation is controlled by *visibleattr* and the well-definedness condition of an expression.
- The transformation $SELF_C$ on a command is defined in Fig. 4, 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, after the execution of the nested call is completed the environment needs to be set back to that of C.

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

Fig. 4. The definition of *SELF*.

Notice that the semantics of a method call defines a method binding mechanism to ensure that

- only a method with a signature declared in the cast type or above the cast type in the inheritance hierarchy can be accessed and
- the method executed is the lowest one in the inheritance hierarchy of the current type of the active object.

Example 2. We illustrate the semantics of method invocation. Consider the bank system in **Example 1** again. We define $Execute(C.m)$ for the method *withDraw*() in the classes *CA* and *SA*. Assume all classes, except for *Bank*, are private classes. For class *CA*,

$$\begin{aligned} Execute(CA.withDraw) &= Set(CA); SELF_{CA}(balance := balance - x); Reset \\ &= visibleattr := \left\{ \begin{array}{l} CA.balance, CA.aNo, \\ Account.balance, Account.aNo \end{array} \right\}; \\ &\quad self.balance := self.balance - x; \\ &\quad visibleattr := \emptyset. \end{aligned}$$

Let o be an object of *CA*. The semantics of the method call $o.withDraw(e)$ attaches o to *self* and then performs $Execute(CA.withDraw)$ as defined above.

For the case of a saving account

$$\begin{aligned} &Execute(SA.withDraw) \\ &= Set(SA); SELF_{SA}(Account.withDraw); Reset \end{aligned}$$

$$= visibleattr := \left\{ \begin{array}{l} SA.blance, SA.aNo, \\ \\ Account.balance, Account.aNo \end{array} \right\};$$
$$self.balance > x \vdash self.balance' = self.balance - x;$$
$$visibleattr := \emptyset.$$

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

4.7. Semantics of a program

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

Recall that *Main* consists of a set of external variables and a command c. For simplicity, we assume that any primitive command in c is in one of the following forms:

(1) an assignment $x := e$ such that $x \in extvar$ and e does not contain subexpressions 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 $C.New(x)$ for a variable $x \in extvar$,
(3) a method call $x.m(ve; re; vre)$, where x is a variable in *extvar*.

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

$$\mathcal{D}(Main) \stackrel{\text{def}}{=} \bigwedge_{x \in extvar} (dtype(x) \in pubcname \vee dtype(x) \in \mathcal{T}).$$

The semantics of *Main* is then defined to be

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

Before *Main* is executed, the local variables have to be initialized to empty sequences:

$$Init \stackrel{\text{def}}{=} \mathcal{D}(Cdecls) \vdash visibleattr' = \emptyset \wedge (\Pi' = \emptyset) \wedge \bigwedge_{x \in var} (\overline{x}' = \langle\,\rangle \wedge TypeSeq'(x) = \langle\,\rangle).$$

Definition 8. The *semantics of an object program* $Cdecls \bullet Main$ is defined as

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

This *black box* semantics hides the internal information, including the objects states of the external variables in the execution of a program, only observing the relation between the prestate and poststate of the external variables. We cannot observe information about states of objects attached to these variables.

We define the *white box semantics* $[\![Cdecls \bullet Main]\!]_o$ as

$$\exists\, \{\Pi\} \geqslant internalvar, \{\Pi'\} \geqslant internalvar', \Omega, \Omega' \cdot$$
$$([\![Cdecls]\!]; Init; [\![Main]\!]; [\![\Pi' := \Pi{\downarrow_{extvar}}]\!]).$$

The white box semantics allows us to observe all information about the external variables including the states of the objects that are attached to them. We can insert the command $\Pi' := \Pi{\downarrow_{extvar}}$ at any point of the main method without changing the white box and close box semantics of a program.

Lemma 1. *The white box semantics has the following properties.*

For any object program $S = Cdecls \bullet Main$ *with main command* c, *we have*:

(1) $[\![Cdecls \bullet c]\!] = \exists \Pi, \Pi' \cdot [\![Cdecls \bullet c]\!]_o$.
(2) $[\![Cdecls \bullet c_1; c_2]\!]_o = [\![Cdecls \bullet c_1; \Pi' := \Pi{\downarrow_{extvar}}; c_2]\!]_o$.
(3) $[\![Cdecls \bullet (c_1; b * (c_2; c_3); c_4)]\!]_o = [\![Cdecls \bullet c_1; b * (c_2; \Pi' := \Pi{\downarrow_{extvar}}; c_3); c_4]\!]_o$.
(4) $[\![Cdecls \bullet (c_1; (c_2; c_3) \lhd b \rhd c_4; c_5)]\!]_o = [\![Cdecls \bullet (c_1; (c_2; \Pi' := \Pi{\downarrow_{extvar}}; c_3) \lhd b \rhd c_4); c_5]\!]_o$.
(5) $[\![Cdecls \bullet (c_1; (c_2; c_3) \sqcap c_4)]\!]_o = [\![Cdecls \bullet c_1; (c_2; \Pi' := \Pi{\downarrow_{extvar}}; c_3) \sqcap c_4]\!]_o$.

5. Object-oriented refinement

We would like the refinement calculus to cover all stages of requirements analysis and specification. This section presents the results of our exploration on two kinds of refinement:
(1) Refinement relation between object systems.
(2) Refinement relation between declaration sections (*structural refinement*).

5.1. Object system refinement

We define what we mean by a refinement between two object programs.

Definition 9. Let $S_i = Cdecls_i \bullet Main_i$, $i = 1, 2$, be object programs which have the same set of external variables $extvar$. S_1 is a *refinement* of S_2, denoted by $S_1 \sqsupseteq_{\text{sys}} S_2$, if the following implication holds:

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

Example 3. 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) 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. 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:

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

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

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

After hiding the internal variables, $[\![S_2]\!]$ is simplified to

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

Thus, S_1 and S_2 refine each other. However, program $S_1; x.get(y)$ is not equivalent to $S_2; x.get(y)$. The final value of y for the first program still remains $d + c$. For the second program, the final value of y is d. On the other hand, if we take the white box semantics, S_1 and S_2 would not be equivalent in the first place. □

This example shows that program refinement is non-compositional. Given two main methods, $Main_i = (extvar, c_i)$, $i = 1, 2$,

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

then it does not necessarily follow that

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

Non-compositionality is caused by the global internal variable Π being hidden in the semantics. However, if we define the refinement relation by the white box semantics, the above non-compositionality would disappear if s only refers to calls to methods of objects attached to the external variables. Therefore, refinement according to the white box is a subrelation of the refinement according to the black box semantics and it is more compositional.

Theorem 2. *Let $Cdecls \bullet Main$, 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.

5.2. Structure refinement

The proof in Example 3 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. A class declaration section can in fact support many different application main programs. The rest of this section focuses on *structural refinement*.

Definition 10. Let $Cdecls_1$ and $Cdecls_2$ be two declaration sections. $Cdecls_1$ is a *refinement* of $Cdecls_2$, denoted by $Cdecls_1 \sqsupseteq_{\text{class}} cdecls_2$, if the former can replace the later in any object system:

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

Informally, $Cdecls_1$ supports at least as many services as $Cdecls_2$. It is obvious that $\sqsupseteq_{\text{class}}$ is reflexive and transitive. We use $=_{\text{class}}$ to denote the equivalence relation $\sqsupseteq_{\text{class}} \cap \sqsubseteq_{\text{class}}$. When there is no confusion, we omit the subscript.

A structural refinement does not change the main method. 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 $Cdecls_1$. Recall that a main method only changes objects by method invocations to public classes.

When considering a refinement between $Cdecls_i$, $i = 1, 2$, we use Ω_i, Π_i, $cname_i$, etc. to denote the structural variables and configuration of $Cdecls_i$ and $[\![\mathcal{E}]\!]_i$ to denote the semantic definition of $\mathcal{E}$ under the declaration $Cdecls_i$. The notation of structural refinement is actually an extension to the notion of data refinement [28].

Definition 11. For $i = 1, 2$, let $Cdecls_i$ be two class declaration sections. A *structural transformation* from $Cdecls_1$ to $Cdecls_2$, is a relation between the object space Σ_1 of $Cdecls_1$ and the object space Σ_2 of $Cdecls_2$ that can be represented as a design $true \vdash \rho(\Omega_1, \Omega_2')$ such that the following conditions hold:

(1) $Cdecls_1$ declares at least those public classes declared in $Cdecls_2$. That is ρ implies

$$true \vdash pubcname_2' \subseteq pubcname_1.$$

(2) For each public class C declared in both $Cdecls_1$ and $Cdecls_2$, $Cdecls_1$ offers at least those methods offered by C than $Cdecls_2$. That is for every $C \in pubcname'$

$$Sig(op_2'(C)) \subseteq Sig(op_1(C)),$$

where *Sig* returns the set of method signatures of a set of method declarations.

(3) The restriction of ρ on the attributes $\rho(ATTR_1(C), ATTR_2'(C))$ for each public class C in both declaration sections can be described in terms of attribute expressions over $ATTR_1(C)$ in $Cdecls_1$ and $ATTR_2'(C)$ in $Cdecls_2$ that
   (a) the attributes' initial values $\rho(init(ATTR_1(C))\ and\ init(ATTR_2(C)))$ are preserved
   (b) the operations on attribute expressions are preserved: if $\rho(\gamma_i, \beta_i)$ hold for all i=1, ..., n, then $\rho(\gamma_1 \cdot \gamma_2, \beta_1 \cdot \beta_2)$ and $\rho(f(\gamma_1, \ldots, \gamma_n), f(\beta_1, \ldots, \beta_n))$ hold.

A structural transformation corresponds to a consistent transformation between the corresponding UML class diagrams [37].

Example 4. Fig. 5 provides two class declaration sections, $Cdecls_1$ on the left and $Cdecls_2$ on the right. Fig. 6 shows the class diagrams of the two declaration sections.

In the "abstract" version $Cdecls_1$ contains two classes, C and C_1. C_1 has two integer attributes a and b, and two methods: $get_a()$ which returns the value of attribute a and $update_a()$ which increments attributes a with the input value parameter. Correspondingly, class C has an attribute o linked to C_1, and a method $get_a()$ which calls o's method $get_a()$ and a method $update_a()$ which simply calls the updating method of o.

Class declaration section $Cdecls_2$ implements C_1 using four classes in which

- C_2 acts as an interface to C as C_1 without storing or manipulating attributes. Each of C_3, C_4 and C_5 stores and manipulates an attribute.
- The attribute $C_1.a$ is implemented by the sum of $C_3.a_3$ and $C_4.a_4$
- The attribute $C_1.b$ is implemented by $C_5.a_5$.
- The $get_a()$ method in C_2 is implemented by getting each of the two attributes in C_3 and C_4 and then adding them together.

$Cdecls_1$	$Cdecls_2$
class C {	
private C_2 o;	
method $get_a(\emptyset; \mathtt{Int}\ x; \emptyset)\{o.get_a(\emptyset; x; \emptyset)\}$;	
$update_a(\mathtt{Int}\ x; \emptyset; \emptyset)\{$	**class** C {
$o.update_a(x; \emptyset; \emptyset)\}$	**private** C_1 o;
};	**method** $get_a(\emptyset; \mathtt{Int}\ x; \emptyset)\{$
private class C_2 {	$o.get_a(\emptyset; x; \emptyset)\}$;
private C_3 o_3, C_4 o_4, C_5 o_5;	$update_a(\mathtt{Int}\ x; \emptyset; \emptyset)\{$
method $get_a(\emptyset; \mathtt{Int}\ x; \emptyset)\{$	$o.update_a(x; \emptyset; \emptyset)\}$
var Int $y; o_3.get(\emptyset; y; \emptyset)$;	};
$o_4.get(\emptyset; x; \emptyset); x := x + y;$ **end** $y\}$;	**private class** C_1 {
$update_a(\mathtt{Int}\ x; \emptyset; \emptyset)\{$	**private Int** $a = 0, \mathtt{Int}\ b = 0$;
$o_3.update(x; \emptyset; \emptyset) \sqcap o_4.update(x; \emptyset; \emptyset)\}$	**method** $get_a(\emptyset; \mathtt{Int}\ x; \emptyset)\{$
};	$x := a\}$;
private class C_i $\{/ * *i = 3, 4, 5$	$update_a(\mathtt{Int}\ x; \emptyset; \emptyset)\{$
private Int $a_i = 0$;	$a := a + x\}$
method $get(\emptyset; \mathtt{Int}\ x; \emptyset)\{x := a_i\}$;	}
$update(\mathtt{Int}\ x; \emptyset; \emptyset)\{a_i := a_i + x\}$	
}	

Fig. 5. Example 4.

- The $update_a()$ in C_2 is implemented by non-deterministically updating and attribute of C_3 and C_4. We define a structural transformation ρ_1 from $Cdecls_1$ to $Cdecls_2$ as

$$true \vdash \begin{pmatrix} C.o' = C.o \\ \wedge\ C_1.a' = C_2.o_3.a_3 + C_2.o_4.a_4 \\ \wedge\ C_1.b' = C_2.o_5.a_5 \end{pmatrix}.$$

Note that the primed attributes of C and C_1 are about attributes in $Cdecls_2$. □

Consider a structural transformation ρ from $Cdecls_1$ to $Cdecls_2$. Let C be a public class in both declaration sections, $o_1 : C$ an object of $Cdecls_1$ and $o_2 : C$ an object of $Cdecls_2$. We say $\rho(o_1, o_2)$ holds if $\rho(ATTR_1(C)[o_1/C], ATTR'_2(C)[o_2/C])$ holds, where $ATTR_i(C)[o_i/C]$ is obtained from $ATTR_i(C)$ by replacing

(1) $C.a$ with $o_i.a$ for each attribute a of C.
(2) $D.b$ with $o_i.a_1.\ldots.a_k.b$ if there exists $a_1, \ldots, a_k, b$ such that $C.a_1.\ldots.a_k.b$ is an attribute expression over $ATTR_i(C)$ and D is the type of a_k.

We say that ρ is a *many-to-one* transformation if for each object $o_1 : C$ under $Cdecls_1$ there is only one $o_2 : C$ under $Cdecls_2$ such that $\rho(o_1, o_2)$.

Theorem 3 (*Upwards simulation implies refinement*). *$Cdecls_1$ is a refinement of $Cdecls_2$ if there is a many-to-one structural transformation $true \vdash \rho(\Omega_1, \Omega'_2)$ such that for any public class name declared in both $Cdecls_1$ and $Cdecls_2$, any variable $x : C$ and any method $m(\underline{x} : \underline{T}_1; \underline{y} : \underline{T}_2; \underline{z} : \underline{T}_3)\{c_1\}$ in a public class C of $Cdecls_1$ and its corresponding method $m(\underline{x} : \underline{T}_1; \underline{y} : \underline{T}_2; \underline{z} : \underline{T}_3)\{c_2\}$ in $Cdecls_2$,*

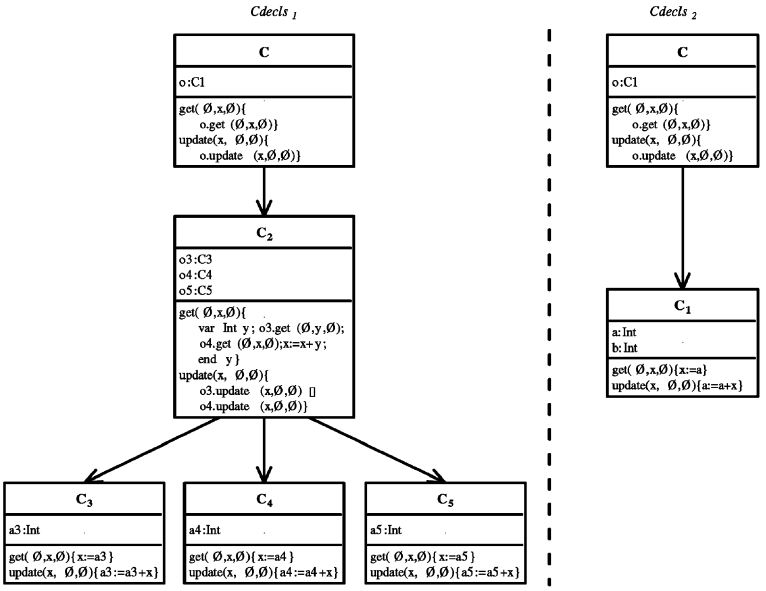

Fig. 6. Example 4.

$$\begin{array}{l} \quad (\llbracket x.m(ve; re; vre)\rrbracket_1; \llbracket \Pi_1 := \Pi_1{\downarrow_{\{x,re,rve\}}}\rrbracket; \rho(\Pi_1, \Pi_2')) \\ \sqsupseteq \ (\rho(\Pi_1, \Pi_2'); \llbracket x.m(ve; re; vre)\rrbracket_2; \llbracket \Pi_2 := \Pi_2{\downarrow_{\{x,re,rve\}}}\rrbracket), \end{array} \tag{1}$$

where $\rho(\Pi_1, \Pi_2')$ *holds iff for each variable* y *and* $o_1 \in \Pi_1$ *such that* $ref(o_1) = ref(y)$ *there is exactly one* $o_2 \in \Pi_2'$ *and* $\rho(o_1, o_2)$.

Proof. Let V be a set of variables and $Main = (V, c)$ be the main method for both $S_i = Cdecls_i \bullet Main$, $i = 1, 2$. From the general theory in UTP [28], we only need to prove there exists a many-to-one mapping $\hat{\rho}$ from the state space of $\{\Pi_1, visibleattr_1\}$ to that of $\{\Pi_2, visibleattr_2\}$ such that

$$\llbracket Init\rrbracket_1; \llbracket c\rrbracket_1; \llbracket \Pi_1 := \Pi_1{\downarrow_V}\rrbracket; \hat{\rho} \sqsupseteq \hat{\rho}; \llbracket Init\rrbracket_2; \llbracket c\rrbracket_2; \llbracket \Pi_2 := \Pi_2{\downarrow_V}\rrbracket. \tag{2}$$

For this, we define

$$\hat{\rho}(\Pi_1, \Pi_2') \stackrel{\text{def}}{=} \rho(\Pi_1, \Pi_2'),$$
$$\hat{\rho}(visibleattr_1, visibleattr_2') \stackrel{\text{def}}{=} visibleattr_2' = \{C.a | C \in pubcname_2 \wedge a \in pub(C)\}.$$

Because of the syntactic definition of the main method of a program, if c is a well-defined primitive command, it can only be one of the following two cases:
(1) It is a command that only involves variables of built-in primitive types. In this case, the theorem obviously holds.
(2) It is an object creation $C.new(x)$ for some $x \in V$ and public class C.
In the case when c is an object creation, $C.new(x)$ does not change $visibleattr$. We also notice both $\llbracket Init\rrbracket_i$ set Π_i to be empty. So after the initialization, $\rho(\Pi_1, \Pi_2)$ holds. We thus have for $i = 1, 2$

$$\begin{array}{l} \quad \llbracket C.new(x)\rrbracket_i; \llbracket \Pi_i := \Pi_i{\downarrow_V}\rrbracket \\ = \ true \vdash \left(\begin{array}{l} \exists\, r \notin ref(\Pi_i) \cdot (\Pi_i' = \emptyset \cup \{\langle r, C, init_i(C)\rangle\} \wedge \\ (x' = \langle r, C\rangle)); \llbracket \Pi_i := \Pi_i{\downarrow_V}\rrbracket \end{array} \right) \\ = \ true \vdash \exists r \notin ref(\Pi_i) \cdot (\Pi_i' = \Pi_i{\downarrow_{\{x\} \gg V}} \cup \{\langle r, C, init_1(C)\rangle\}) \wedge (x' = \langle r, C\rangle) \\ = \ true \vdash \exists r \in REF \cdot ((\Pi_i' = \{\langle r, C, init_i(C)\rangle\}) \wedge (x' = \langle r, C\rangle)). \end{array}$$

So we have

$$(\llbracket C.new(x)\rrbracket_1; \llbracket \Pi_1 := \Pi_1{\downarrow_V}\rrbracket; \rho(\Pi_1, \Pi_2')) \Rightarrow (\llbracket C.new(x)\rrbracket_2; \llbracket \Pi_2 := \Pi_2{\downarrow_V}\rrbracket).$$

Assume that Refinement (2) holds for command c, we need to prove it holds for command $c; c_1$. As the mapping on *visibleattr* is constant, we can ignore it in the proof. Furthermore, from Lemma 1, we can equivalently take c to be $c; \Pi' := \Pi{\downarrow_V}$. Let $[\![c]\!]_i = p_i \vdash R_i(V \cup \{\Pi_i\}, V' \cup \{\Pi'_i\})$ for $i = 1, 2$. The proof heavily use the definition of sequential composition of designs

$$(p_1(\alpha) \vdash R_1(\alpha, \alpha'); p_2(\alpha) \vdash R_2(\alpha, \alpha')) \stackrel{\text{def}}{=} \exists s_m \cdot (p_1(\alpha) \vdash R_1(\alpha, s_m) \wedge p_2(s_m) \vdash R_2(s_m, \alpha')).$$

Case 1: If c_1 only involves external variables of built-in primitive types, the refinement obviously holds as it does not change the system configuration.

Case 2: Command c is an object creation $C.new(x)$. We have

$$\begin{aligned}
&[\![c; C.new(x)]\!]_i; [\![\Pi_i := \Pi_i{\downarrow_V}]\!] \\
&= [\![c]\!]_i; true \vdash \begin{pmatrix} \exists r \notin ref(\Pi_i) \cdot ((\Pi'_i = \Pi_i \cup \{\langle r, C, init_i(C)\rangle\}) \wedge \\ (x' = \langle r, C\rangle)) \end{pmatrix}; [\![\Pi_i := \Pi_i{\downarrow_V}]\!] \\
&= \exists V_m, \Pi_i^{m_i} \cdot \begin{pmatrix} p_i \vdash \begin{pmatrix} R_i(V \cup \{\Pi_i\}, V_m \cup \{\Pi_i^{m_i}\}) \wedge \\ \exists r \notin ref(\Pi_i^{m_i}) \cdot ((x' = \langle r, C\rangle) \wedge \\ (\Pi'_i = \Pi_i^{m_i}{\downarrow_{\{x\} \geqslant V}} \cup \{\langle r, C, init_i(C)\rangle\})) \end{pmatrix} \end{pmatrix}.
\end{aligned}$$

The induction assumption implies that for any $V, \Pi_1, \Pi_2, \Pi_1^{m_1}, \Pi_2^{m_2}$,

$$p_1 \vdash R_1(V \cup \{\Pi_1\}, V_m \cup \{\Pi_1^{m_1}\}) \wedge \rho(\Pi_1^{m_1}, \Pi_2^{m_2}) \Rightarrow p_2 \vdash R_2(V \cup \{\Pi_2\}, V_m \cup \{\Pi_2^{m_2}\}). \tag{3}$$

Also the structural transformation ensures that $\rho(\Pi_1^{m_1}, \Pi_2^{m_2})$ implies

$$\rho(\Pi_1^{m_1}{\downarrow_{\{x\} \geqslant V}} \cup \{obj_1^0(C)\}, \Pi_2^{m_2}{\downarrow_{\{x\} \geqslant V}} \cup \{obj_2^0(C)\}),$$

where $obj_i^0(C)$ is the object of C with its initials state defined in $Cdecls_i$ for $i = 1, 2$. This proves the refinement for this case.

Case 3: c_1 is $x.m(ve; re; vre)$. For $i = 1, 2$, let

$$[\![x.m(ve; re; vre)]\!]_i \stackrel{\text{def}}{=} p_1^i \vdash R_1^i(V \cup \{\Pi_i\}, V' \cup \{\Pi'_i\}).$$

By the definition of composition, $[\![c; x.m(ve; re; vre)]\!]_i$ equals

$$\exists V_m, \Pi_i^{m_i} \cdot \begin{pmatrix} p_i \vdash R(V \cup \{\Pi_i\}, V_m \cup \{\Pi_i^{m_i}\}) \wedge \\ p_1^i(V_m \cup \Pi_i^{m_i}) \vdash R_1^i(V_m \cup \Pi_i^{m_i}, V' \cup \{\Pi'_i\}) \end{pmatrix}. \tag{4}$$

Notice that the method call $x.m(ve; re; vre)$ only changes the object attached to x and those variables whose reference values are the same as x, and it may modify the objects attached to *re* and *vre* if they and their types are classes.

The structural transformation ensures that if $\rho(\Pi_1^{m_1}, \Pi_2^{m_2})$ and $\rho(\Pi_1^{n_1}{\downarrow_{V_1}}, \Pi_2^{n_2}{\downarrow_{V_1}})$ for a subset V_1 of V, we then have

$$\rho(\Pi_1^{m_1} \oplus \Pi_1^{n_1}{\downarrow_{V_1}}, \Pi_2^{m_2} \oplus \Pi_2^{n_2}{\downarrow_{V_1}}), \tag{5}$$

where $\oplus$ replace the objects in $\Pi_i^{m_i}$ that are attached to the variables in V_1 with those in $\Pi_i^{n_i}$.

From formula 4

$$\begin{aligned}&[\![c; x.m(ve; re; vre); \Pi_1 := \Pi_1{\downarrow_V}]\!]_1; \rho(\Pi_1, \Pi_2')\\ =\ & \exists V_m, \Pi_1^{m_1}, \Pi_1^m \cdot \begin{pmatrix} p_1 \vdash R(V \cup \{\Pi_1\}, V_m \cup \{\Pi_1^{m_1}\}) \wedge \\ p_1^1(V_m \cup \Pi_1^{m_1}) \vdash R_1^1(V_m \cup \Pi_1^{m_1}, V' \cup \{\Pi_1^m\}) \wedge \\ \rho(\Pi_1^m, \Pi_2') \end{pmatrix}.\end{aligned}$$

Notice that $\Pi_1^m = \Pi_1^{m_1} \oplus \Pi_1^m{\downarrow_{\{x,re,vre\}}}$. Property 5 of structural transformation together with Condition 1 and the induction assumption 3 proves the refinement for this case.

Case 4: If c_1 is a command only involved in variables of built-in primitive types, the refinement obviously holds.

Case 5: If c_1 is an assignment $x := y$ of one object variable to another, the execution of $\Pi_i := \Pi{\downarrow_V}$ after the execution of c_1 only removes from Π_i the object originally attached to y.

Case 6: If c_1 is $x := (C)y$, it changes Π_i in the same way as in Case 4, but assign the value $\langle ref(y), C\rangle$ to x in both programs.

Case 7: Let c_1 be a conditional choice $c_{11} \lhd b \rhd c_{12}$ and b an expression of variables of built-in primitive types (and constants). b is evaluated to *true* after the execution of c in S_1 if and only if it is evaluated to *true* after the execution of c in program S_2 because of the induction assumption. This case can then be proven for each c_{11} and c_{12} separately.

Case 8: If c_1 is a loop $b * c_{11}$, the refinement can then be proven by the induction and the properties of the weakest fixed point. □

Theorem 4 (*Downwards simulation implies refinement*). *$Cdecls_1$ is a refinement of $Cdecls_2$ if there is a one-to-many structural transformation $true \vdash \rho(\Omega_2, \Omega_1')$ such that for any public class name declared in both $Cdecls_1$ and $Cdecls_2$, any variable $x : C$ and any method $m(\underline{x} : \underline{T}_1; \underline{y} : \underline{T}_2; \underline{z} : \underline{T}_3)\{c_1\}$ in a public class C of $Cdecls_1$ and its corresponding method $m(\underline{x} : \underline{T}_1; \underline{y} : \underline{T}_2; \underline{z} : \underline{T}_3)\{c_2\}$ in $Cdecls_2$,*

$$\begin{aligned}&(\rho(\Pi_2, \Pi_1'); [\![x.m(ve; re; vre)]\!]_1; [\![\Pi_1 := \Pi_1{\downarrow_{\{x,re,rve\}}}]\!])\\ \sqsupseteq\ & ([\![x.m(ve; re; vre)]\!]_2; [\![\Pi_2 := \Pi_2{\downarrow_{\{x,re,rve\}}}]\!]; \rho(\Pi_2, \Pi_1')).\end{aligned} \tag{6}$$

Example 5. For the class declaration sections in Example 4, we can also define a structural transformation ρ_2 from $Cdecls_2$ to $Cdecls_1$:

$$true \vdash \begin{pmatrix} & C.o = C.o' \wedge C_1.b = C_2.o_5.a_5' \\ \wedge & C_1.a = C_2.o_3.a_3' + C_2.o_4.a_4' \end{pmatrix}.$$

It is a one-to-many transformation. With this transformation, we can check if $Cdecls_2$ is also a refinement of $Cdecls_1$.

In the same way that we prove Theorem 3 we can prove the following theorem.

Theorem 5. *Let $Cdecls_1 \sqsupseteq Cdecls_2$ and Cdecls be a class declaration such that if $a : C \in Attr(M)$ for some M in Cdecls and C in $Cdecls_1$ then C is a public class. We have*

$$Cldecls_1; Cdecl \sqsubseteq Cdecls_2; Cdecls.$$

The proof is similar to Theorem 3.

Remarks. A structural refinement corresponds to a consistent transformation between the corresponding UML class diagrams, sequence diagrams and state diagrams [37]. A (upwards) structural refinement of a program under ρ is shown in Fig. 7.

Theorems 3 and 4 do not appear very helpful as refinement does not directly mention refinement of private classes. However, the theorems allow us to take a method m in a public class C as a "main method". This method may call methods of classes that are directly linked to C. Treating these classes as "'public classes" with respect to C and

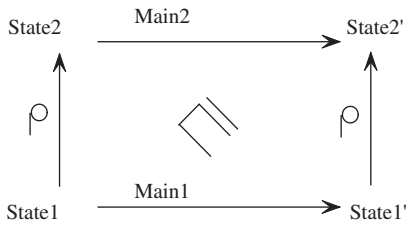

Fig. 7. Commuting diagram for class refinement.

these classes together with their associated classes as a declaration section, the refinement Conditions 1 and 6 can be established for this subdeclaration section.

In general, finding and formulating a refinement mapping ρ is design step. It is easier to develop a system in a stepwise process in which each step is modest. This approach leads itself to establishing correctness in an incremental manner.

This framework suggests that a development process should first focus on structural refinement and then carries out further refinement of methods of classes and the main method of the program under a fixed class declaration, without hiding the internal states. This can be done entirely within the classical theories of programming provided in UTP [28].

6. Refinement rules

We have given some refinement laws for refining commands in Section 4.6. Those laws are about command refinement under the same class declaration sections. They can all be proven in the classical theory of programming [28].

We now present refinement rules for program structures that capture the nature of *incremental* development in object-oriented programming. Most of the laws are intuitively understandable. Their proof involves finding structural transformations and then using Theorems 3 and 4 (refinement by upwards or downwards simulations). The structural transformations are quite obvious for the laws presented and we omit most of the proofs.

We first introduce some notations. We use $N[supclass, pri, prot, pub, op]$ to denote a well-formed class declaration that declares the class N that has *supclass* as its direct superclass; *pri*, *prot* and *pub* as its sets of private, protected and public attributes; and *op* as its set of methods. *supclass* is always of either a class name M, when M is the direct superclass of N, or $\emptyset$ when N has no superclass. We may only refer to some, or even none of $M, pri, prot, pub, op$ when we talk about a class declaration. For example, N denotes a class declaration for N, and $N[pri]$ a class declaration that declares the class N that has *pri* as its private attributes.

Law 4. *The order of the class declarations in a declaration section is not essential*:

$$N_1; \ldots; N_n = N_{i_1}; \ldots; N_{i_n},$$

where N_i is a class declaration and $i_1, \ldots, i_n$ is a permutation of $\{1, \ldots, n\}$.

A law like this may look utterly trivial after we formalize the structural variables Ω, but it is not so obvious that a semantic definition of a class declaration to guarantee this law. For example, if the precondition of the class declaration requires that the direct superclass be declared before this class declaration, this law would not hold.

The next law says that more services may become available after adding a class definition.

Law 5. *If a class name N is not in Cdecls, but M is in Cdecls*

$$Cdecls \sqsubseteq N[M, pri, prot, pub, op]; Cdecls$$

provided the right-hand side is well-defined.

The structural transformation only extends the set *cname*. The consequence is only that a command c in the main method which is not well-defined in the original declaration becomes well-formed in the extended declaration.

The next law states that the introduction of a private attribute has no effect.

Law 6. *If neither N nor any of its superclasses and subclasses in Cdecls has x as an attribute*

$$N[pri]; Cdecls \equiv N[pri \cup \{Tx = d\}]; Cdecls$$

provided d lies in T and either T is a primitive type, or T is declared in Cdecls or $T = N$.

Although adding an attribute has no effect, it will allow more well-defined classes and methods to be introduced using other laws.

Law 7. *Changing a private attribute into a protected one may support more services.*

$$N[pri \cup \{T\,x = d\}, prot]; Cdecls \sqsubseteq N[pri, prot \cup \{Tx = d\}]; Cdecls.$$

This refinement becomes equivalence if both sides are well-defined. This condition is required as we do not allow a protected attribute of a class to be redeclared in its subclass.

Similarly, changing a protected attribute to a public attribute refines the declaration too. This together with the above two laws allow us to add new attributes as long as the well-definedness is not violated.

Law 8. *Adding a new method can refine a declaration. If m is not defined in N, let* $m(paras)\{c\}$ *be a method with distinct parameters paras and a command c. Then*

$$N[op]; Cdecls \sqsubseteq N[op \cup \{m(paras)\{c\}\}]; Cdecls.$$

The structural transformation only extends $op(N)$ in the new declaration section, and does not change the dynamic state variables.

Law 9. *We can refine a method. If* $c_1 \sqsubseteq c_2$,

$$N[op \cup \{m(paras)\{c_1\}\}]; Cdecls \sqsubseteq N[op \cup \{m(paras)\{c_2\}\}]; Cdecls.$$

The refinement of the command is done under the same dynamic variables.

Law 10. *Inheritance introduces refinement. If none of the attributes of M is defined in N or any superclass of N in Cdecls*,

$$M[\emptyset, pri, prot, pub, op]; Cdecls \sqsubseteq M[N, pri, prot, pub, op]; Cdecls$$

provided the right-hand side is well-formed.

Introducing an inheritance in this way in fact enlarges the set of attributes of N (and those of the subclasses of N). A structural transformation from the new declaration section just projects the enlarged set attribute back to the original attributes.

Law 11. *We can introduce a superclass. Let*

$$C_1 = M[\emptyset, pri \cup A, prot, pub, op],$$
$$C_2 = M[\{N\}, pri, prot, pub, op].$$

Assume N is not declared in Cdecls,

$$C_1; Cdecls \sqsubseteq C_2; N[\emptyset, \emptyset, A, \emptyset, \emptyset]; Cdecls.$$

This can be in fact derived from adding a class and then introducing inheritance. After introducing a subclass this way, we can continue to apply other laws to introduce more attributes and methods.

Law 12. *We can move some attributes of a class to its superclass. If all the subclasses of N but M do not have attributes in A, then*

$$N[prot_1]; M[\{N\}, prot \cup A]; Cdecls \sqsubseteq N[prot_1 \cup A]; M[\{N\}, prot]; Cdecls.$$

This only enlarges the set of attributes of N. This law and the law for promoting an attribute to a protected attribute allow us to move a private attribute to the superclass too. Repeated application of this law allows us to move the common attributes of the direct subclasses of a class to the class itself.

Law 13. *If N has $M_1, \ldots, M_k$ as its direct subclasses,*

$$N[prot]; M_1[prot_i \cup A]; \ldots; M_k[prot_k \cup A]; Cdecls \sqsubseteq N[prot \cup A]; M_1[prot_1]; \ldots; M_k[prot_k]; Cdecls.$$

Law 14. *We copy (but not remove) a method of a class to its superclass. Let $m(paras)\{c\}$ be a method of M, but not a method of its superclass N:*

$$N[op]; M[\{N\}, op_1 \cup \{m(\underline{paras})\{c\}\}]; Cdecls \sqsubseteq N[op \cup \{m(\underline{paras})\{c\}\}]; M[\{N\}, op_1 \cup \{m(\underline{paras})\{c\}\}]; Cdecls.$$

Copying a method subclass to its direct of a class does not change any dynamic variable.

Law 15. *Let $m(paras)\{c\}$ be a method of N, then*

$$N[op]; M[\{N\}, op_1]; Cdecls \sqsubseteq N[op]; M[\{N\}, op_1 \cup \{m(\underline{paras})\{c\}\}]; Cdecls.$$

We can remove a redundant method from a subclass.

Law 16. *Assume class N is the direct superclass of M, $m(paras)\{c\} \in op \cap op_1$, and c only involves in the protected attributes of N,*

$$N[op]; M[\{N\}, op_1]; Cdecls \sqsubseteq N[op]; M[\{N\}, \{m(\underline{paras})\{c\}\} \gg op_1]; Cdecls.$$

Similarly, we can remove any unused private attributes.

Law 17. *If (Tx) is a private attribute of $N[pri]$ that is not used in any command of N,*

$$N[pri]; Cdecls \sqsubseteq N[\{Tx = d\} \gg pri]; Cdecls.$$

We can also remove any unused protected attributes.

Law 18. *If $(Tx = d)$ is a protected attribute of $N[prot]$ that is not used in any command of N and any subclass of N,*

$$N[prot]; Cdecls \sqsubseteq N[\{Tx = d\} \gg prot]; Cdecls.$$

Law 19. *We can change a private class into a public class.*

$$\texttt{private}\ N; Cdecls \sqsubseteq N; Cdecls.$$

A class is allowed to delegate some tasks to its associated classes. [2]

Law 20 (*Expert pattern for responsibility assignment*). *Suppose $M[op_1]$ is declared in Cdecls, where Csecls has*
(1) *an attribute x,*
(2) *a method $m()\{c_1(x)\} \in op_1$ which may manipulate attribute x through execution of command c_1.*

[2] This law is very useful in object-oriented system designs [35].

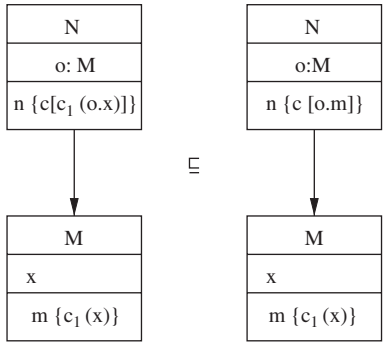

Fig. 8. Object-oriented functional decomposition.

Assume that (Mo) *is an attribute of* N, *then*

$$N[op \cup \{n(paras)\{c[\tilde{c_1}]\}\}]; Cdecls \sqsubseteq N[op \cup \{n(paras)\{c[o.m()]\}\}]; Cdecls.$$

Here, c_1 *is obtained from* $\tilde{c_1}$ *by replacing* $o.x$ *with* x, *that is*, $c_1 = \tilde{c_1}[x/o.x]$. *Assume that* $\tilde{c_1}$ *does not refer to any attribute of* N. *While* $c[\tilde{c_1}]$ *denotes that* $\tilde{c_1}$ *occurs as part of command* c, *and* $c[o.m]$ *denotes that the command obtained from* $c[\tilde{c_1}]$ *by substituting* $o.m$ *for* $\tilde{c_1}$.

Proof. Assume that M and N are public classes. It is easy to see there is a structural transformation that is identical except for $op(N)$. The dynamic state variables are the same in both declaration sections. For the left-hand side declaration section to be well-defined, x has to be a public attribute of M.

Without losing any generality, assume that in the left hand side declaration section,

$$[\![c_1(o.x)]\!]_2 = p(y_1, y_3, o.x, \Pi) \vdash R(y_1, y_3, o.x, y_2', y_3', o.x', \Pi') \wedge (y_1' = y_1),$$

where y_1 does not appear in the left side of an assignment, the initial value of y_2 is not relevant in the execution of c_1 and y_3 is a general variable. We assume that they are not attributes of M. In this case y_1, y_2 and y_3 are the actual parameters of $o.M()$ in the declaration section on left-hand side of the law. According to the semantics of a method call, we calculate the design for $[\![o.m()]\!]_2$ in the right-hand side of the law.

$$\begin{aligned}
[\![o.m()]\!]_1 \;=\;& \texttt{var}\ M\ self = o, T_1 f_1 = y_1, T_2 f_2, T_3 f_3 = y_3; Set(M);\\
& p(f_1; f_2; f_3, self.x, \Pi) \vdash R(f_1, f_3, self.x, f_1', f_2', f_3', self.x', \Pi');\\
& y_2 := f_2; y_3 := f_3; \ \texttt{end}\ self, f_1; f_2; f_3; Reset\\
\Rightarrow\;& p(y_1; y_3, o.x, \Pi) \vdash R(y_1, y_3, o.x, y_2', y_3', o.x', \Pi') \wedge (y_1' = y_1)\\
=\;& [\![c_1(o.x)]\!]_2.
\end{aligned}$$

This implies that method $n()$ in class N satisfies the condition of Theorem 3 for the structural transformation. In case one or both of N and M are private, the refinement law holds because of Theorem 2. □

This law is illustrated by the UML class diagram in Fig. 8. It will become an equation if x is a public attribute of M. To understand this law, let us consider the simple example from the aforementioned bank system in Examples 1 and 2.

Consider the method *getBalance* of class *Bank*. Initially, we might have the following design for it:

$$getBalance(\texttt{Int}\ aID,\ \texttt{Int}\ res, \emptyset) \stackrel{\text{def}}{=}$$
$$\exists a \in \Pi(Account) \cdot a.aNo = aID \vdash \exists a \in \Pi(Account) \cdot a.aNo = aID \Rightarrow res' = a.balance.$$

Note that it requires the attributes of class *Account* to be visible (public) to other classes (like *Bank*). Applying Law 20 to it, we can get the following design:

$$getBalance(\texttt{Int}\ aID, \texttt{Int}\ res, \emptyset) \stackrel{\text{def}}{=}$$
$$\exists\, a \in \Pi(Account) \cdot a.aNo = aID \vdash \exists a \in \Pi(Account) \cdot a.aNo = aID \Rightarrow a.getBalance(\emptyset; res; \emptyset).$$

The refinement delegates the task of balance lookup to the *Account* class.

It is important to note that method invocation, or in other words, object interaction takes time. Therefore, this object-oriented refinement (and the one described in Law 22) usually exchanges efficiency for ease of reuse and maintainability, and data encapsulation.

After functionalities are delegated to associated classes, data encapsulation can be applied to increase security and maintainability. The visibility of an attribute can be changed from public to protected, or from protected to private under certain circumstances.

Law 21 (*Data encapsulation*). *Suppose* $M[pri, prot, pub]$, *and* $(T_1 a_1 = d_1) \in pub$, $(T_2 a_2 = d_2) \in prot$.
(1) *If no operations of other classes have expressions of the form* $le.a_1$, *except for those of subclasses of* M, *we have*

$$M[pri, prot, pub]; Cdecls \sqsubseteq M[pri, prot \cup \{T_1 a_1 = d_1\}, \{T_1 a_1 = d_1\} \geqslant pub]; Cdecls.$$

(2) *If no operations of any other classes have expressions of the form* $le.a_2$, *we have*

$$M[pri, prot, pub]; Cdecls \sqsubseteq M[pri \cup \{T_2 a_2 = d_2\}, \{T_2 a_2 = d_2\} \geqslant prot, pub]; Cdecls.$$

The structural transformation only changes the different kind of attributes, it may thus affect visibility of attributes, and thus the well-definedness of commands. However, this will not happen because of the side conditions.

After applying Law 20 exhaustively to method *getBalance*, and applying Law 21 to the class diagram on the right-hand side of Fig. 8, we achieve the encapsulation of the attribute *balance* of the class *Account*. The attribute *aNo* can be encapsulated in a similar way.

Another principle of object-oriented design is to make classes simple and highly cohesive. This means that the responsibilities (or functionalities) of a class, i.e. its methods, should be strongly related and focused. We therefore often need to decompose a complex class into a number of associated classes, so that the system will be

- easy to comprehend,
- easy to reuse,
- easy to maintain,
- less delicate and less effected by changes.

We capture the *High Cohesion* design pattern [35] by the following refinement rule.

Law 22 (*High cohesion pattern*). *Assume* $M[pri, op]$ *is a well-formed class declaration,* $pri = \{x, y\}$ *are (or are lists of) attributes of* M, $m_1()\{c_1(x)\} \in op$ *only contains attribute* x, *method* $m_2()\{c_2[m_1]\} \in op$ *can only change* x *by calling* m_1 *(or it does not have to change it at all). Then*
(1) $M; Cdecls \sqsubseteq M[pri_{new}, op_{new}]; M_1[pri_1, op_1]; M_2[pri_2, op_2]; Cdecls$, *where*

- $pri_{new} = \{M_1 o_1, M_2 o_2\}$,
- $op_{new} = \{m_1()\{o_1.m_1\}, m_2()\{o_2.m_2\}\}$,
- $pri_1 = \{x\}$, $op_1 = \{m_1()\{c_1(x)\}\}$,
- $pri_2 = \{y, M_1 o_1\}$, $op_2 = \{m_2()\{c_2[o_1.m_1()]\}\}$

such that $\forall o : M \cdot (o.o_1 = o.o_2.o_1)$ *is an invariant of* M. *This invariant has to be established by the constructors of these three classes.*

This refinement is illustrated by the diagram in Fig. 9.
(2) $M; Cdecls \sqsubseteq M[pri_{new}, op_{new}]; M_1[pri_1, op_1]; M_2[pri_2, op_2]; Cdecls$, *where*

- $pri_{new} = \{M_2 o_2\}$,
- $op_{new} = \{m_1()\{o_1.m_1()\}, m_2()\{o_2.m_2()\}\}$,
- $pri_1 = \{x\}$, $op_1 = \{m_1()\{c(x)\}\}$,

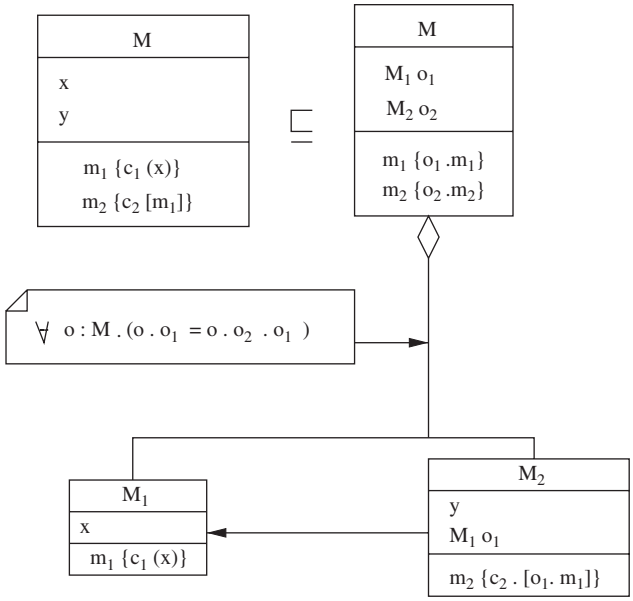

Fig. 9. Class decomposition (1).

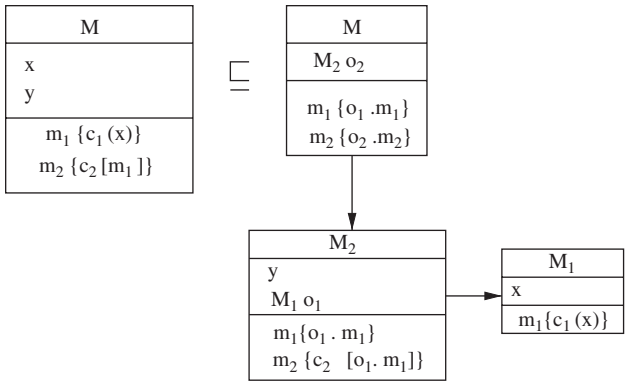

Fig. 10. Class decomposition (2).

- $pri_2 = \{y, M_1 o_1\}$,
- $op_2 = \{m_1()\{o_1.m_1()\}, m_2()\{c_2[o_1.m_1()]\}\}$.

This refinement is illustrated by the diagram in Fig. 10.

The structural transformations for the two cases have been nearly given in the law. The proofs of the two refinements in the law are similar to that for the expert pattern. First, take M to be a public class and then use Theorem 2.

Notice that the first refinement in Law 22 requires that M be coupled with both M_1 and M_2; and in the second refinement M is only coupled with M_2, but more interaction between M_2 and M_1 is needed than in the first refinement. We believe that the above three laws, together with the other simple laws for incremental programming effectively support the use-case driven and iterative RUP development process [35]. The use of the patterns for responsibility assignment in object-oriented software development is clearly demonstrated in Larman's book [35].

For each of the laws, except for Law 9, let *LHS* and *RHS* denote the declarations on the left- and right-hand sides, respectively. For any main program *Main*, each refinement law becomes an equational law: $LHS \bullet Main \equiv RHS \bullet Main$, provided $LHS \bullet Main$ is well-defined.

7. Conclusions

We have shown how Hoare and He's design calculus [28] can be used to define an object-oriented language. A program or a command is represented as a predicate called a *design*, and the refinement relation between designs is defined as logic implication. Our model reflects most of the features of object-oriented designs [11]. For example, the model shows that inheritance with attribute hiding and method overriding makes system analysis difficult, while method invocation on an object may change external states. The good news is that we have been able to impose constraints on system development so that the "bad" features are not used.

7.1. Related work

Formal techniques for object-orientation have been extensively studied [3,56,45,12,1,8]. The work there concerns programming languages. A large amount of work on operational semantics [56,12] supports methods of simulation and model checking. Our calculus is based on a relational model that supports state-based reasoning and stepwise (or incremental) refinement in system development.

There are a number of recent articles on Hoare logics for object-oriented programming (see, e.g. [46,55,30,47,36,13]). The normal form of a program in our article is similarly to that of [13,46]. However, one major difference of our work is that we also provide a formal characterization and refinement of the contextual (or structural) features, i.e. the declaration section, of an object program. This is motivated by our work on the formalization of UML models [40,41]. This characterization has been proven to be very useful in defining semantics for integrated specification languages in general.

Class or object refinements are studied in [5,36]. A refinement object-oriented language (ROOL) and some general notions of refinement are defined in [13] using predicate transformers without treating reference types. The work in [10], also without treatment of reference types, describes a set of algebraic laws for ROOL, that can be used to derive refactorings [18,19]. Our initial version of rCOS (called OOL) with a relational semantics and the idea object-oriented refinement were presented in [24]. OOL does not have references types or nested variable declarations. In this article, we have revised OOL and its semantics. We have also provided refinement laws that reflect the characteristic aspects, functionality delegation, data encapsulation and class decomposition for high cohesion, of object-oriented design and the ideas of design patterns [21,35]. We also take a *weak semantic* approach meaning that when the precondition of a contract is not satisfied, the program will behave as *chaos*; any program modification made, such as adding exception handling, is a refinement. We also describe static well-formedness conditions in the precondition so that any correction of any static inconsistency in a program, such as static-type mismatching, missing variables, missing methods, etc. can be treated as refinements too. This allows us to treat *refactoring* [18] as refinement and to combine it with *behavioural refinement*. This combination is important for composing different UML models and reasoning about their consistency [40,41,37].

Our work on formal support for object-oriented design using UML [40,41,37] has provided us with the insight of functional decomposition in the object-oriented setting and its relation with data encapsulation and class decomposition. The main ideas of those article are summarized in the following subsection.

7.2. Support UML-like software development

Consider the incremental and iterative rational unified process (RUP) [33] and the use-case driven approach [31]. System requirements capture and analysis starts by identifying domain (or business) services and the domain structure that consists of the domain *classes* (or *concepts*) and their *associations*. Business services can be described by a UML use-case model and the domain structure is represented as a UML class diagram. The UML class diagram can be formally specified as a rCOS class declaration section, and each use case is declared as a set of methods of a *use-case controller* class. Then the application program is specified as a main method that uses the services, i.e. calls to the methods, provided in the use-case controller classes. Therefore, the normal requirement specification is of the form

$$(CM; Controller_1; \cdots; Controller_n) \bullet Main,$$

where CM is a sequence of class declarations obtained from the class diagram (an association is also declared as a class). Each $Controller_i$ is a use-case controller class (following the facade controller pattern [21,35]) that

contains the functional specifications (in terms of designs in rCOS) and formalizes the system sequence diagram of the corresponding use case. The *consistency* of the class diagram and the use cases (their sequence diagrams and functional specifications) has to ensure that the class diagram *fully supports* the use cases. Formally, this means that the declaration section $(CM; Controller_1; \cdots; Controller_n)$ of the program is well-formed and any invocation of a method in a use-case controller in P does not end with $chaos$. In case of any inconsistency, we can modify the class diagram or the use cases (or both) according to the refinement laws that allow us to change the UML model consistently.

We design each use case by applying Law 20 to delegate its partial responsibilities to other classes in the class diagram according to *what information a class maintains or knows via its associations with other classes*. In the mean time, we can decompose complex classes according to Law 22 and encapsulate data according to Law 21. Obviously, before applying Law 20 or 22, we have to add classes, attributes and methods. These design or refinement activities lead to incremental creation of the sequence diagrams and design class diagram of the system, and the refined laws will ensure that the design class diagram refines the requirement class diagram. For details about formalization of UML models of requirements and designs in rCOS, we refer the reader to [40,41,37]. For detailed, but informal, application of the design patterns that have been formalized as refinement laws in this article, please see Larman's book [35].

rCOS captures the commonality and difference between structured functional development and object-oriented development. In the traditional structured approach, a software project starts with the identification of data and functions. A specification of a procedure defines how the data are manipulated in terms of precondition and postcondition: $\{Pre\}F\{Post\}$. The design is to decompose the functions step by step into subfunctions by applying the decomposition rule

$$\frac{\{Pre\}F_1\{Mid\}, \{Mid\}F_2\{Post\}}{F \sqsubseteq F_1; F_2}.$$

The problem with this approach is that it is difficult to determine a suitable Mid, among many possibilities. In the object-oriented approach that we propose here, we use the *expert pattern* (Law 20) and high cohesion pattern (Law 22) to decompose a use case according to the system structure modelled by the class diagram. As in the functional approach, the decomposition has to preserve the functional specification of the use case, i.e. the pre- and postcondition relations. However, the decomposition is more pragmatic as its is supported by the known *structure*. In the structured approach, the design of the system has to be constructed by decomposition too.

The research of formal support for UML modelling is currently very active [7,22,48]. However, there is a large body of work in formalizing UML and providing tool support for UML focuses on models for a particular view (e.g. a class models, statecharts, and sequence diagrams), and the translation of them into an existing formal formalism (e.g. Z, VDM, B, and CSP). Very little work has been conducted as to how UML models can be *refined consistently*. In contrast, we are concerned with combinations of different UML models, the most *imprecise* part of UML. Our methodology is directed towards improved support for requirement analysis and transition from requirements to design models in RUP. Our choice of a Java-like syntax for the specification language is a pragmatic solution to the problems of representing name spaces and (the consequences of) inheritance in a notation such as CSP.

7.3. Limitation and future work

rCOS can be extended to deal with features of communication, interaction, real-time and resources. If we add variables for traces, refusals and divergence into the alphabet, the different kinds of semantics of communicating processes can be defined as designs [28]. By introducing clock variables in the alphabet [32,28,57,52], we can define real-time programs as designs and further extend our approach to support other aspects of object-oriented programming. Alternatively, one can also use temporal logic, such as [2], for the specification and verification of multi-threading Java-like programs. However, we would like to deal with concurrency at a higher level [25,23,38].

In [11], Broy argued that the property of object identities is too low level and implementation oriented. The use of references does cause side-effects, making the semantics more complex. A preliminary version of the model without references can be found in [24]. This simplification is not significant. The complexity mainly affects reasoning about low-level designs and implementations. With our approach, we can describe change of system state in terms of what objects are created or deleted, what modifications are made to an object and what links between objects are formed or broken. Low-level features such as method overriding and attribute hiding are only useful to program around the requirement and design defects detected at the coding stage or even later when one tries to reuse a class with a similar

template in a program that the class was not originally designed. These features cause problems in programming verification and the smooth application of the notion of program refinements.

Future work includes the study of the completeness of the refinement calculus and the applications of the method to more realistic case studies. We will also extend this work to deal with component systems [38,25,23]. Further challenges for formal object-oriented methods include the formal treatment of *patterns* [21] in general. We are also interested in studying the difference and relationship between our model and separation logic [49,14], that can be used for extending the calculus to multi-thread programming.

Acknowledgements

We would like to thank the referees for their thorough reviews. The detailed and constructive comments have helped us to bring the paper to the current form. We thank Shengchao Qin at National University of Singapore for his comments and LaTex improvement on an earlier version of the paper. We also thank Dines Bjorner at Technical University of Denmark, Anders Ravn from Aalborg University of Denmark and Uday Reddy from Birmingham University of the UK for their helpful comments and discussions at and after the seminars on parts of the works that the third author gave when he visited them. Our UNU-IIST fellows Xin Chen, Jing Liu, Xiaojian Liu, Quan Long, Leila Silva, Bhim Upadhyaya, Jing Yang and Liang Zhao also read and gave useful comments on earlier versions of the article. The third author would also like to thank the students at the University of Leicester and those participants of the UNU-IIST training schools and courses who took his course on Software Engineering and System Development for their feedback on the understanding of the use-case driven, incremental and iterative object-oriented development and the design patterns.

References

[1] M. Abadi, R. Leino, A logic of object-oriented programs, in: M. Bidoit, M. Dauchet (Eds.), TAPSOFT '97: Theory and Practice of Software Development Seventh International Joint Conference, Springer, Berlin, 1997, pp. 682–696.
[2] E. Abraham-Mumm, F.S. de Boer, W.P. de Roever, M. Steffen, Verification for Java's reentrant multithreading concept, Foundations of Software Science and Computation Structures, Lecture Notes in Computer Science, Vol. 2303, Springer, Berlin, 2002, pp. 5–20.
[3] P. America, Designing an object-oriented programming language with behavioural subtyping. in: J.W. de Bakker, W.P. de Roever, G. Rozenberg (Eds.), REX Workshop, Lecture Notes in Computer Science, Vol. 489, Springer, 1991, pp. 60–90.
[4] P. America, F. de Boer, Reasoning about dynamically evolving process structures, Formal Aspects Comput. 6 (3) (1994) 269–316.
[5] R. Back, A. Mikhajlova, J. von Wright, Class refinement as semantics of correct object substitutability, Formal Aspects Comput. 2 (2000) 18–40.
[6] R. Back, J. vonWright, Refinement Calculus, Springer, Berlin, 1998.
[7] R.J.R. Back, L. Petre, I.P. Paltor, Formalizing UML use cases in the refinement calculus, in: Proc. UML'99, Springer, Berlin, 1999.
[8] M.M. Bonsangue, J.N. Kok, K. Sere, An approach to object-orientation in action systems, in: J. Jeuring (Ed.), Mathematics of Program Construction, Lecture Notes in Computer Science, Vol. 1422, Springer, Berlin, 1998, pp. 68–95.
[9] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modelling Language User Guide, Addison-Wesley, Berlin, MA, 1999.
[10] P. Borba, A. Sampaio, M. Cornélio, A refinment algebra for object-oriented programming, in: L. cardelli (Ed.), Proc. ECOOP03, Lecture Notes in Computer Science, Vol. 2743, Springer, Berlin, 2003, pp. 457–482.
[11] M. Broy, Object-oriented programming and software development—a critical assessment, in: A. McIver, C. Morgan (Eds.), Programming Methodology, Springer, Berlin, 2003.
[12] K. Bruce, J. Grabtre, G. Kanapathy, An operational semantics for TOOPLE: a statically-typed object-oriented programming language, in: S. Brooks et al. (Ed.), Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science, Vol. 802, Springer, Berlin, 1994, pp. 603–626.
[13] A. Cavalcanti, D. Naumann, A weakest precondition semantics for an object-oriented language of refinement, Lecture Notes in Computer Science, Vol. 1709, Springer, Berlin, 1999, pp. 1439–1460.
[14] Y. Chen, J. Sanders, Compositional reasoning for pointer structures, in: Eighth Internat. Conf. on Mathematics of Program Construction (MPC'06), Lecture Notes in Computer Science, Vol. 4014, Springer, Berlin, 2006, pp. 115–139.
[15] D. Coleman, et al., Object-Oriented Development: the FUSION Method, Prentice-Hall, Englewood cliffs, NJ, 1994.
[16] S. Cook, J. Daniels, Designing Object Systems: Object-Oriented Modelling with Syntropy, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[17] E. Dürr, E.M. Dusink, The role of VDM^{++} in the development of a real-time tracking and tracing system, in: J. Woodcock, P. Larsen (Eds.), Proc. of FME'93, Lecture Notes in Computer Science, Vol. 670, Springer, Berlin, 1993.
[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999.
[19] M. Fowler, Refectoring Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 2000.
[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994.

[21] E. Gamma, et al., Design Patterns, Addison-Wesley, Reading, MA, 1995.
[22] D. Harel, B. Rumpe, Modeling languages: syntax, semantics and all that stuff—part I: the basic stuff, Technical Report MCS00-16, The Weizmann Institute of Science, Israel, September 2000.
[23] J. He, X. Li, Z. Liu, Component-based software engineering, in: Proc. Second Internat. Colloq. on Theoretical Aspects of Computing (ICTAC05), Lecture Notes in Computer Science, Vol. 3722, Springer, Berlin, 2005, pp. 70–95.
[24] J. He, Z. Liu, X. Li, Towards a refinement calculus for object-oriented systems (invited talk), in: Proc. ICCI02, Alberta, Canada, IEEE Computer Society, Silverspring, MD, 2002.
[25] J. He, Z. Liu, X. Li, A theories of reactive contracts, Electronic Notes of Theoretical Computer Science, Vol. 160, 2006, pp. 173–195.
[26] J. He, Z. Liu, X. Li, S. Qin, A relational model of object oriented programs, in: Proc. of the Second ASIAN Symp. on Programming Languages and Systems (APLAS04), Lecture Notes in Computer Science, Vol. 3302, Taiwan, March 2004, Springer, Berlin, pp. 415–436.
[27] C.A.R. Hoare, Laws for programming, Comm. ACM 30 (1987) 672–686.
[28] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1998.
[29] I. Houston, Formal specification of the OMG core object model, Technical Report, IMB, UK, Hursely Park, 1994.
[30] M. Huisman, B. Jacobs, Java program verification via a Hoare logic with abrupt termination, in: T. Maibaum (Ed.), FASE 2000, Lecture Notes in Computer Science, Vol. 1783, Springer, Berlin, 2000, pp. 284–303.
[31] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999.
[32] N. Jin, J. He, Resource models and pre-compiler specification for hardware/software, in: J.R. Cuellar, Z. Liu (Eds.), Proc. Second Internat. Conf. on Software Engineering and Formal Methods SEFM04, Beijing, China, IEEE Computer Society, Sliverspring, MD, 2004, pp. 28–30.
[33] P. Kruchten, The Rational Unified Process—An Introduction, Second ed., Addison-Wesly, Reading, MA, 2000.
[34] K. Lano, H. Haughton, Object-oriented specification case studies, Prentice-Hall, New York, 1994.
[35] C. Larman, Applying UML and Patterns, Prentice-Hall International, Englewood Cliffs, NJ, 2001.
[36] K.R.M. Leino, Recursive object types in a logic of object-oriented programming, Lecture Notes in Computer Science, Vol. 1381, Springer, Berlin, 1998.
[37] X. Li, Z. Liu, J. He, Q. Long, Generating prototypes from a UML model of requirements, in: Internat. Conf. on Distributed Computing and Internet Technology (ICDIT2004), Lecture Notes in Computer Science, Vol. 3347, Bhubaneswar, India, Springer, Berlin, 2004.
[38] Z. Liu, J. He, X. Li, Contract-oriented development of component systems, in: Proc. of IFIP WCC-TCS2004, Toulouse, France, Kluwer Academic Publishers, Dordrecht, 2004, pp. 349–366.
[39] Z. Liu, J. He, X. Li, rCOS: refinement of component and object systems, in: Proc. Third Internat. Symp. on Formal Methods for Components and Objects (FMCO04), Lecture Notes in Computer Science, Vol. 3657, Springer, Berlin, 2005, pp. 222–250.
[40] Z. Liu, J. He, X. Li, Y. Chen, A relational model for formal requirements analysis in UML, in: J.S. Dong, J. Woodcock (Eds.), Formal Methods and Software Engineering, ICFEM03, Lecture Notes in Computer Science, Vol. 2885, Springer, Berlin, 2003, pp. 641–664.
[41] Z. Liu, J. He, X. Li, J. Liu, Unifying views of UML, Electronic Notes Theoret. Comput. Sci. (ENTCS) 101 (2004) 95–127.
[42] B. Meyer, From structured programming to object-oriented design: the road to Eiffel, Structured Programming 10 (1) (1989) 19–39.
[43] A. Mikhajlova, E. Sekerinski, Class refinement and interface refinement in object-orient programs, in: Proc of FME'97, Lecture Notes in Computer Science, Springer, Berlin, 1997.
[44] C.C. Morgan, Programming from Specifications, Second ed., Prentice-Hall, Englewood Cliffs, NJ, 1994.
[45] D. Naumann, Predicate transformer semantics of an Oberon-like language, in: E.-R. Olerog (Ed.), Proc. of PROCOMET'94, North-Holland, Amsterdam, 1994.
[46] C. Pierik, 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.
[47] A. Poetzsch-Heffter, P. Muller, A programming logic for sequential Java, in: S.D. Swierstra (Ed.), Proc. Programming Languages and Systems (ESOP'99), Lecture Notes in Computer Science, Vol. 1576, Springer, Berlin, 1999, pp. 162–176.
[48] G. Reggio, et al., Towards a rigorous semantics of UML supporting its multiview approach, in: H. Hussmann (Ed.), Proc. FASE 2001, Lecture Notes in Computer Science, Vol. 2029, Springer, Berlin, 2001.
[49] J. Reynolds, Separation logic: a logic for a shared mutable data structure, in: Proc. of IEEE Symp. Logic in Computer Science (LICS'02), IEEE Computer Society, Sliverspring, MD, 2002.
[50] D.B. Roberts, Practical Analysis for Refactoring, Ph.D. Thesis, University of Illinois, Urbana Champain, 1999.
[51] E. Sekerinski, A type-theoretical basis for an object-oriented refinement calculus, in: Proc. of Formal Methods and Object Technology, Springer, Berlin, 1996.
[52] A. Sherif, J. He, A. Cavalcanti, A. Sampaio, A framework for specification and validation of real-time systems using Circus actions, in: Proc. First Internat. Colloq. on Theoretical Aspects of Computing (ICTAC04), Lecture Notes in Computer Science, Vol. 3407, Springer, Berlin, 2005, pp. 478–494.
[53] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, Dordrecht, 2000.
[54] L.A. Tokuda, Evolving Object-Oriented Designs with Refactoring, Ph.D. Thesis, University of Texas Austin, 1999.
[55] D. von Oheimb, Hoare logic for Java in Isabelle/HOL, Concurrency Comput: Practice Experience 13 (13) (2001) 1173–1214.
[56] D. Walker, β-calculus semantics of object-oriented programming languages, in: Proc. TACAS'91, Lecture Notes in Computer Science, Vol. 526, Springer, Berlin, 1991, pp. 532–547.
[57] J.C.P. Woodcock, A.L.C. Cavalcanti, A semantics of Circus, in: ZB 2002, Lecture Notes in Computer Science, Vol. 2272, Springer, Berlin, 2002.
[58] J. Yang, Q. Long, Z. Liu, X. Li, A predicative semantic model for integrating UML models, in: Proc. First Internat. Colloq. on Theoretical Aspects of Computing (ICTAC04), Lecture Notes in Computer Science, Vol. 3407, Springer, Berlin, 2005, pp. 170–186.