# A Relational Model for Object-Oriented Designs

He Jifeng[1], Zhiming Liu[1,2], Xiaoshan Li[3] and Shengchao Qin[4]

[1] International Institute for Software Technology, The United Nations University
{hjf,lzm}@iist.unu.edu
[2] Department of Mathematics and Computer Science, The University of Leicester
zl2@mcs.le.ac.uk
[3] Faculty of Science and Technology, The University of Macau
xsl@umac.mo
[4] Singapore-MIT Alliance & Dept. of Computer Science, National Univ. of Singapore
qinsc@comp.nus.edu.sg

**Abstract.** This paper develops a mathematical characterisation of object-oriented concepts by defining an observation-oriented semantics for an object-oriented language (OOL) with a rich variety of features including subtypes, visibility, inheritance, dynamic binding and polymorphism. The language is expressive enough for the specification of object-oriented designs and programs. We also propose a calculus based on this model to support both structural and behavioural refinement of object-oriented designs. We take the approach of the development of the design calculus based on the standard predicate logic in Hoare and He's Unifying Theories of Programming (UTP). We also consider object reference in terms of object identity as values and mutually dependent methods.

**Keywords:** *Object Orientation, Refinement, Semantics, UTP*

## 1 Introduction

Software engineering is mainly concerned with using techniques to systematically develop large and complex program suites. However, it is well known that it is hard to obtain the level of assurance of correctness for safety critical software using old fashioned programming techniques. In the search for techniques for making software cheaper and more reliable, two important but largely independent approaches have been visibly influential in recent years. They are

– object-oriented programming, and
– formal methods.

First, it becomes evident that *objects* are and will remain an important concept in software development. Experimental languages of the 1970's introduced various concepts of package, cluster, module, etc, giving concrete expression to the importance of modularity and encapsulation, the construction of software components hiding their state representations and algorithmic mechanisms from users, exporting only those features which are needed in order to use the components. This gives the software components a level of abstraction, separating the view of what a module does for the system from the details of how it does them. It is also clear that certain features of 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.

At least for highly critical systems, it seems essential to give software engineering the same basis in mathematics that is the hall mark of other important engineering disciplines. In this there has good progress, resulting in three main paradigms: model-based, algebraic and process calculi. Both practitioners of formal methods and experts in object technology have investigated how formal specification can supplement object-oriented development, e.g. [21], or how it may help to clarify the semantics of object-oriented notations and concepts, e.g. [1]. Examples of such work include formalisation of the OMG's core object model [19] using Z.

Model-based formalisms have been used extensively in conjunction with object-oriented techniques, via languages such as Object-Z [8], VDM++ [12], and methods such as Syntropy [11] which uses the Z notation and Fusion [10] that is related to VDM. Whilst these formalisms are effective at modelling data structures as sets and relations between sets, they are not ideal for capturing more sophisticated object-oriented mechanisms, such as dynamic binding and polymorphism.

Using predicate transformer, Cavalcanti and Naumann defined an object-oriented programming language with subtype and polymorphism [9, 29]. Sekerinski [33, 28] 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 tackled in those approaches. America and de Boer have given a logic for the parallel language POOL [5]. It applies to imperative programs with object sharing, but without subtyping and method overriding. Abadi and Leino defined an axiomatic semantics for an imperative, object-oriented language with object sharing [2], 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 [31]. However, as pointed by Leino in [23], instead of allowing the designer of a method defining its specification and then checking that implementation meet the specification, the specification of a method in the Poetzsch-Heffter and Müller logic is derived from the method's known implementation. Leino presented a logic in [23] with imperative features, subtyping, and recursive types. It allows the specification of methods of classes, but restricting inheritance and not dealing with visibility.

In this paper, we aim to develop a mathematical characterisation of object-oriented concepts, and provide a proper semantic basis essential for ensuring correctness and for the development of tool support for the use of formal techniques. We define an object-oriented language with subtypes, visibility, reference types, inheritance, dynamic binding and polymorphism. The language is sufficiently similar to Java and C++ and can be used in meaningful case studies and to capture some of the central difficulties in modelling object-oriented programs.

We build a logic of object-oriented programs as a conservative extension of the standard predicate logic [18]. 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. Our framework allows local variables to be redefined in its scope. Consequently, their states will usually comprise sequences of values. A variable of a primitive type stores a data of the corresponding type whereas a variable of an object type holds the identity or reference of an object as its value. We define the traditional programming constructs, such as conditional, sequential composition, and recursion in the exactly 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. For simplicity, unlike [30], we consider neither attribute domain redefinition nor attribute hiding. This assumption will be incorporated into the well-formedness condition of a declaration section in Section 3. With this assumption, the set $\mathtt{attr}(C)$ of attributes of *C* contains all the attributes declared in *C* and those inherited from its superclasses. We simplify the model this way because our focus is program requirement specification, design and verification, whilst attribute domain redefinition and attribute hiding are languages facilities for programming around defects in the requirement specification and design or for the reuse of some classes that were not originally designed for program being developed.

After this introduction, Section 2 introduces the syntax of the language. The semantics of the language is given in Section 3, with the discussion about behavioural refinement of OO designs. In Section 4, we present some initial work towards a (structural) refinement calculus for OO design and programming. We will draw some conclusions in Section 5.

## 2 Syntax

In our model, an object system (or program) $S$ is of the form *cdecls* • *P*, where *cdecls* is a *declaration* of a finite number of classes, and *P* is called the main method and is of the form $(\mathtt{glb}, c)$ consisting of a finite set `glb` of *global variables* with their types and a command *c*. *P* can be understood as the `main` method if $S$ is taken as a Java program.

### 2.1 Class declarations

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

$[\mathtt{private}]\ \mathtt{class}\ N\ \mathtt{extends}\ M\ \{$

$\mathtt{private}\ (U_i\ u_i = a_i)_{i:1..m};\ \ \mathtt{protected}\ (V_i\ v_i = b_i)_{i:1..n};\ \ \mathtt{public}\ (W_i\ w_i = c_i)_{i:1..k};$

$\mathtt{method}\quad m_1(\underline{T}_{11}\ \underline{x}_1, \underline{T}_{12}\ \underline{y}_1, \underline{T}_{13}\ \underline{z}_1)\{c_1\}; \cdots; m_\ell(\underline{T}_{\ell 1}\ \underline{x}_\ell, \underline{T}_{\ell 2}\ \underline{y}_\ell, \underline{T}_{\ell 3}\ \underline{z}_\ell)\{c_\ell\}\}$

Note that

- A class can be declared as `private` or `public`. By default, it is assumed as `public`. We use a function *anno* to extract this information from a class declaration such that *anno*(*cdecl*) is *true* if *cdecl* declares a private class and *false* otherwise.
- *N* and *M* are distinct names of classes, and *M* is called the direct superclass of *N*.
- Attributes annotated with `private` are private attributes of the class, and similarly, the `protected` and `public` declarations for the protected and public attributes. Types and initial values of attributes are also given in the declaration.

- the `method` declaration declares the methods, their value parameters ($\underline{T}_{i1}\ \underline{x}_i$), result parameters($\underline{T}_{i2}\ \underline{y}_i$), value-result parameters ($\underline{T}_{i3}\ z_i$) and bodies ($c_i$). We sometimes denote a method by $m(\underline{paras})\{c\}$, where $\underline{paras}$ is the list of parameters of $m$ and $c$ is the body command of $m$.
- The body of a method $c_i$ is a command that will be defined later.

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

### 2.2 Commands

Our language supports typical object-oriented programming constructs, but we also allow some commands for the purpose of specification and refinement:

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

where $b$ is a Boolean expression, $e$ is an expression, and $le$ is an expression which may appear on the left hand side of an assignment and is of the form $le ::= x \mid le.a$ where $x$ is a simple variable and $a$ an attribute of an object. Unlike [30] that introduces "statement expressions", we use $le.m(\underline{e}, \underline{v}, \underline{u})$ to denote a call of method $m$ of the object denoted by the left-expression $le$ with actual value parameters $\underline{e}$ for input to the method, actual result parameters $\underline{v}$ for the return values, and value-result parameters $\underline{u}$ that can be changed during the execution of the method call and with their final values as return values too; and use the command $C.\mathit{new}(x)[\underline{e}]$ to create a new object of class $C$ with the initial values of its attributes assigned to the values of the expressions in $\underline{e}$ and assign it to variable $x$. Thus, $C.\mathit{new}(x)[\underline{e}]$ uses $x$ with type $C$ to store the newly created object.

### 2.3 Expressions

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

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

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

## 3 Semantics

We now show how to use the basic model of the UTP to define the semantics of our language. We will adopt the convention that the semantics $[\![\mathcal{E}]\!]$ of an element $\mathcal{E}$, such as $e_1 = e_2$ or $x := e$, of the language is denoted by $\mathcal{E}$ itself in a semantic defining equation. When $\mathcal{E}$ appears on the left hand side of a defining equation, it means that its semantics is defined as the right hand side of the equation. When $\mathcal{E}$ appears on the right hand side, it denotes its *defined* semantics of $\mathcal{E}$.

## 3.1 Programs are designs

In [18], Hoare and He proposed a state-based model in which a program or a program command is identified as a *design*, represented by a pair $(\alpha, P)$, where $\alpha$ denotes the set of variables of the program, and $P$ is a predicate of the form

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

Notice that

- we call $\alpha$ the *alphabet* of the design and $P$ the *contract* of the design; $\alpha$ declares the *variables* (including logical ones) whose *values* form the *state* of the program at a moment of time, and the contract specifies the *behaviour* of the program in terms of what change in the state it may make.
- $x$ and $x'$ stand for the initial and final values of program variables $x$ in $\alpha$, respectively.
- predicate $p$, called the *precondition* of the program, characterises the initial states in which the activation of the program will lead its execution to termination.
- predicate $R$, called the *post-condition* of the program, relates the initial states of the program to its final states, and
- we describe the termination behaviour of a program by the Boolean variables $ok$ and $ok'$, where the former is true if the program is properly activated and the later becomes true if the execution of the program terminates successfully.

In what follows, we give formal definitions of sequential composition of designs and design refinement.

**Definition 1.** *For a given alphabet $\alpha$ and two contracts $P_1$ and $P_2$, the sequential composition $P_1; P_2$ is defined as the relation composition*

$$(P_1(x, x'); P_2(x, x')) =_{df} \exists m \cdot P_1(x, m) \wedge P_2(m, x')$$

*We also define the composite design $(\alpha, P_1); (\alpha, P_2)$ by $(\alpha, P_1; P_2)$.*

Within this model, the concept of refinement is defined as predicate implication.

**Definition 2.** *(***Design refinement***) Design $D_2 =_{df} (\alpha, P_2)$ is a* refinement *of design $D_1 =_{df} (\alpha, P_1)$, denoted by $D_1 \sqsubseteq D_2$, if $\forall x, x' \ldots, z, z' \cdot (P_2 \Rightarrow P_1)$, where $x, \ldots, z$ are variables contained in $\alpha$. $D_1 \equiv D_2$ if and only if $D_1 \sqsubseteq D_2$ and $D_2 \sqsubseteq D_1$.*

**Definition 3.** *(***Data refinement***) Let $\rho$ be a mapping (that can also be specified as a design) from $\alpha_2$ to $\alpha_1$. Design $D_2 =_{df} (\alpha_2, P_2)$ is a* refinement *of design $D_1 =_{df} (\alpha_1, P_1)$ under $\rho$, denoted by $D_1 \sqsubseteq_\rho D_2$, if $(\rho; P_1) \sqsubseteq (P_2; \rho)$.*

A program command usually modifies a subset of the program variables in $\alpha$. Let $V$ be a subset of $\alpha$ , the notation $V : (p \vdash R)$ denotes the (*framed*) design $p \vdash (R \wedge \underline{w}' = \underline{w})$, where $\underline{w}$ contains all variables in $\alpha$ but those in $V$. $V$ is called the frame of the design $p \vdash R$. In examples, we often omit the frames of designs by assuming that a design only changes the value of a variable $x$ if its primed version $x'$ occurs in the design.

For simplicity, the above model in [18] adopts a universal data type and allows neither reference types nor nested declaration. This assumption will not be applicable to modelling OO designs anymore. However, we can still follow this classical way of defining a state-based model for a programming language and define our OOL in terms of *values, variables, states, expressions, commands, declarations and programs*.

## 3.2 Values, Objects, Variables and States

Each program declares a set `cname` of class names, a partial function `superclass` that maps a class name in `cname` to its *direct superclass*, a function `attr` that associates each class name $C \in$ `cname` with the set $\mathtt{attr}(C)$ of its attributes, and a function `op` that associates each $C \in$ `cname` the set $\mathtt{op}(C)$ of its methods. We use $\preceq$ to denote the reflexive and transitive closure of `superclass` and $C_1 \preceq C_2$ denotes that $C_1$ is a subclass of $C_2$.

We assume a set $\mathcal{T}$ of *primitive types* and an infinite set *REF* of *object identities* (or *references*), and $null \in REF$. A value is either a member of a primitive type in $\mathcal{T}$ or an object identity in *REF*. Let the set of values be $VAL =_{df} \bigcup \mathcal{T} \cup REF$. An *object* $o$ is an entity defined by the following structure $o ::= null \mid \langle ref, \mathtt{type}, \mathtt{state} \rangle$, where $ref \in REF$, and `type` is a class name, and `state` is a mapping from $\mathtt{attr}(\mathtt{type})$ to *VAL*. Given an object $o = \langle ref, C, \sigma \rangle$, we use $identity(o)$ to denote the identity *ref* of $o$, $\mathtt{type}(o)$ the type $C$ of the object $o$, and $\mathtt{state}(o)(a)$ the value $\sigma(a)$ of an attribute $a$ of class $C$.

Let $\mathcal{O}$ be the set of all objects, including *null*. Notice that infinite recursive and looping constructions are allowed, such as $\langle r_i, C, \sigma_i \rangle$ such that $\sigma_i(a) = r_i$, where $a$ is an attribute of $C$ that is type of $C$ too.

The following notations will be employed in the semantics definitions.

- Given a non-empty sequence $\underline{s} = \langle s_1, .., s_k \rangle$, we have $head(\underline{s}) = s_1$, $tail(\underline{s}) = \langle s_2, .., s_k \rangle$. We use $|\underline{s}|$ to denote the length of $s$, and $\pi_i(\underline{s})$ the $ith$ element $s_i$, for $i : 1, .., k$.
- For two sets $S$ and $S_1$, $S_1 \trianglerighteq S$ is the set obtained by removing elements in $S_1$ from $S$. Note that $\trianglerighteq$ has higher associativity than normal set operators like $\cup$, $\cap$.
- For a mapping $F : D \longrightarrow E$, $d \in D$ and $r \in E$,

$$F \oplus \{d \mapsto r\} =_{df} F' \qquad \text{where } F'(b) =_{df} \begin{cases} r, & \text{if } b = d; \\ F(b), & \text{if } b \in \{d\} \trianglerighteq D. \end{cases}$$

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

$$ref \oplus \{a \mapsto d\} =_{df} \langle ref, C, \sigma \oplus \{a \mapsto d\} \rangle$$

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

$$S \uplus \{\langle ref, C, \sigma \rangle\} =_{df} \{o \mid identity(o) = ref\} \trianglerighteq S \;\cup\; \{\langle ref, C, \sigma \rangle\}$$
$$Ref(S) =_{df} \{ref \mid ref \text{ is the identity of an object in } S\}$$

Our model describes the behaviour of an OO program as a *design* containing the logical variables given in Fig 1 as its *free variables* that form the alphabet of the program.

The semantic model will ensure that for any $o_1$ and $o_2$ in $\Sigma$, $identity(o_1) = identity(o_2)$ implies $\mathtt{type}(o_1) = \mathtt{type}(o_2)$ and $\mathtt{state}(o_1) = \mathtt{state}(o_2)$. We therefore can use identity of an object to refer to an object in $\Sigma$. In the rest of the paper, an object $o = \langle ref, C, \sigma \rangle$ means one in $\Sigma$ if there is no confusion, and will use *ref.a* to denote the value of $\mathtt{state}(o)(a)$, and $\mathtt{type}(ref)$ to denote $\mathtt{type}(o)$ (i.e. $C$).

## 3.3 Evaluation of expressions

The evaluation of an expression $e$ determines its type $\mathtt{type}(e)$ and its value that is a member of $\mathtt{type}(e)$ if this type is primitive, and an object of the current type that is

| *variable* | *representation* | *description* |
|---|---|---|
| `cname` | | the set of classes declared so far |
| `pricname` | | the set of private class names |
| $\mathtt{attr}(C)$ | $\{\langle a_i : T_i, d_i\rangle\}_{i=1}^{m}$ | $T_i$ and $d_i$ are the type and initial value of attribute $a_i$, and will be referred by $\mathtt{decltype}(C.a_i)$ and $\mathtt{initial}(C.a_i)$ respectively. We also abuse the notation $a \in \mathtt{attr}(C)$ and use it to denote $\exists T, d \cdot (\langle a : T, d\rangle \in \mathtt{attr}(C))$. Again, we do not allow attribute hiding (or redefinition) in a subclass. We also use an attribute name to represent its value and a type name to denote the set of its legal values. |
| $\mathtt{op}(C)$ | $\{m_1 \mapsto (\underline{x}_1{:}\underline{T}_{11}, \underline{y}_1{:}\underline{T}_{12}, \underline{z}_1{:}\underline{T}_{13}, D_1),$ $\ldots,$ $m_k \mapsto (\underline{x}_k{:}\underline{T}_{k1}, \underline{y}_k{:}\underline{T}_{k2}, \underline{z}_k{:}\underline{T}_{k3}, D_k)\}$ | each method $m_i$ has $\underline{x}_i$, $\underline{y}_i$ and $\underline{z}_i$ as its value, result and value-result parameters respectively, that are denoted by $\mathtt{val}(C.m_i)$, $\mathtt{res}(C.m_i)$, and $\mathtt{valres}(C.m_i)$, and the behaviour of $m_i$ is defined by the design $D_i$ referred by $\mathtt{Def}(C.m_i)$. Sometimes we simply denote each element in $\mathtt{op}(C)$ as $m_i \mapsto (\underline{paras}_i, D_i)$. We also sometimes abuse the notation $m \in \mathtt{op}(C)$ and use it to denote $\exists \underline{paras}, D \cdot (m \mapsto (\underline{paras}, D) \in \mathtt{op}(C))$ |
| $\Sigma(C)$, $\Sigma$ | $\Sigma =_{df} \bigcup_{C \in \mathtt{cname}} \Sigma(C)$ | $\Sigma(C)$:the set of objects of class $C$ that currently exist in the execution of the program. $\Sigma$: system state, also called current configuration [30] |
| `superclass` | $\{N \mapsto M, \ldots\}$ | a partial function mapping a class ($N$) to its *direct* superclass ($M$). |
| `glb` | | the set of global variables declared at the beginning of the main program |
| `locvar` | $\{(x_1, \langle T_{11}, .., T_{1m}\rangle),$ $\ldots,$ $(x_n, \langle T_{n1}, .., T_{nk}\rangle)\}$ | the set of local variables which are known to the current scope of the program. $T_{i1}$, for $i = 1, .., n$ is the most recently declared type of $x_i$ |
| `var` | $\mathtt{var} = \mathtt{glb} \cup \mathtt{locvar}$ | |
| `visibleattr` | | the set of attributes which are visible from inside the current class, i.e. all its declared attributes plus the protected attributes of its superclasses and all public attributes. Every time before a method of an object is executed, this set is set to the attributes of the class of the object, and it will be reset after the execution of the method. |
| $\overline{x}$ | | the state of variable $x \in \mathtt{var}$. Since a local variable can be redeclared, its state usually comprises a nonempty finite sequence of values, whose first (head) element represents the current value of the variable. $\overline{x}$ for $x \in \mathtt{glb}$ contains at most one value and thus we can simply use $x$ to denote it. A primitive variable takes values of primitive type, while an object variable can store an object *name* or *identity* as its value. |

**Fig. 1.** The Alphabet: Logical Variables

attached to $e$. The evaluation makes use of the state of $\Sigma$. However, an expression can only be evaluated when it is well-defined. Some well-definedness conditions are static that can be checked at compiling time, but some are dynamic. The evaluation results of expressions are given in Fig. 2.

| Expression | Evaluation |
|---|---|
| *null* | $\mathcal{D}(null) =_{df} true$, $\mathtt{type}(null) =_{df} NULL$, $\mathtt{value}(null) =_{df} null$ |
| $x$ | $\mathcal{D}(x) =_{df} x \in \mathtt{var} \wedge (\mathtt{decltype}(x) \in \mathcal{T} \vee \mathtt{decltype}(x) \in \mathtt{cname})$ (Static)<br>$\wedge\ \mathtt{decltype}(x) \in \mathcal{T} \Rightarrow head(\overline{x}) \in \mathtt{decltype}(x)$ (Dynamic)<br>$\wedge\ \mathtt{decltype}(x) \in \mathtt{cname} \Rightarrow$<br>$head(\overline{x}) \in Ref(\Sigma(\mathtt{decltype}(x)))$ (Dynamic)<br>$\mathtt{type}(x) =_{df} \begin{cases} \mathtt{decltype}(x) & \mathtt{decltype}(x) \in \mathcal{T} \\ \mathtt{type}(head(\overline{x})) & \text{otherwise} \end{cases}$<br>$\mathtt{value}(x) =_{df} head(\overline{x})$ |
| *self* | $\mathcal{D}(self) =_{df} self \in \mathtt{locvar} \wedge \mathtt{decltype}(self) \in \mathtt{cname}$ (Static)<br>$\wedge\ head(\overline{self}) \in Ref(\Sigma(\mathtt{decltype}(self)))$ (Dynamic)<br>$\mathtt{type}(self) =_{df} \mathtt{type}(head(\overline{self}))$<br>$\mathtt{value}(self) =_{df} head(\overline{self})$ |
| $x.a$ | $\mathcal{D}(x.a) =_{df} \mathcal{D}(x)$<br>$\wedge\ \mathtt{decltype}(x) \in \mathtt{cname} \wedge \mathtt{type}(x).a \in \mathtt{visibleattr}$ (Static)<br>$\wedge\ head(\overline{x}) \neq null$<br>$\mathtt{type}(x.a) =_{df} \mathtt{type}(head(\overline{x}).a)$<br>$\mathtt{value}(x.a) =_{df} head(\overline{x}).a$ |
| $le.a$ | $\mathtt{D}(le.a) =_{df} \mathtt{D}(le) \wedge \mathtt{type}(le).a \in \mathbf{visibleattr}$<br>$\mathtt{value}(le.a) =_{df} \mathtt{value}(le).a$<br>$\mathtt{type}(le.a) =_{df} \mathtt{type}(\mathtt{value}(le).a)$ |
| $(e\ \mathbf{is}\ C)$ | $\mathcal{D}(e\ \mathbf{is}\ C) =_{df} \mathcal{D}(e) \wedge (\mathtt{type}(e) \in \mathtt{cname}) \wedge (C \in \mathtt{cname})$<br>$\mathtt{type}(e\ \mathbf{is}\ C) =_{df} Bool$<br>$\mathtt{value}(e\ \mathbf{is}\ C) =_{df} \mathtt{value}(e) \neq null \wedge \mathtt{type}(e) \preceq C$ |
| $(C)e$ | $\mathcal{D}((C)e) =_{df} \mathcal{D}(e\ \mathbf{is}\ C) \wedge \mathtt{value}(e\ \mathbf{is}\ C)$<br>$\mathtt{type}((C)e) =_{df} \mathtt{type}(e)$<br>$\mathtt{value}((C)e) =_{df} \mathtt{value}(e)$ |
| $e/f$ | $\mathcal{D}(e/f) =_{df} \mathcal{D}(e) \wedge \mathcal{D}(f) \wedge \mathtt{decltype}(e) = Real$<br>$\wedge\ \mathtt{decltype}(f) = Real \wedge \mathtt{value}(f) \neq 0$<br>$\mathtt{value}(e/f) =_{df} \mathtt{value}(e)/\mathtt{value}(f)$ |

**Fig. 2.** Evaluation of Expressions

## 3.4 Semantics of commands

A typical aspect of an execution of an OO program is about how objects are to be attached to program variables (or entities [27]). An attachment is made by an assignment, the object creation or parameter passing in a method invocation. With the approach of UTP, these different cases are unified as an assignment of a value to a program variable. We shall only present the semantic definitions for assignment, object creation and method calls, due to page limit. All other programming constructs will be defined in exactly the same way as their counter-parts in a procedural language, thus are omitted here. We also present some basic refinement laws for commands.

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

$$\mathcal{D}(le := e) =_{df} \mathcal{D}(le) \wedge \mathcal{D}(e) \wedge \mathtt{type}(e) \preceq \mathtt{decltype}(le)$$

Notice that this requires dynamic type matching. However, it is *safe* to replace the condition $\mathtt{type}(e) \preceq \mathtt{decltype}(le)$ with $\mathtt{decltype}(e) \preceq \mathtt{decltype}(le)$, as the semantics will ensure the later implies the former. With the use of type test e **is** $C$ and type casting $(C)e$, changing the dynamic type matching to the static matching will not lose expressive power either.

There are two cases of assignment. The first is to (re-)attach a value to a variable (i.e. change the current value of the variable), but this can be done only when the type of the object is consistent with the declared type of the variable. The attachment of values to other variables are not changed.

$$x := e =_{df} \{x\} : \mathcal{D}(x := e) \vdash (\overline{x}' = \langle \mathtt{value}(e) \rangle \cdot \mathit{tail}(\overline{x}))$$

As we do not allow attribute hiding/redefinition in subclasses and semantics of assignment, the assignment to a simple variable has not side-effect, and thus the Hoare triple $\{o_2.a = 3\}\ o_1 := o_2\ \{o_1.a = 3\}$ is valid in our model for variables $o_1$ of class $C_1$ and $o_2$ of $C_2$, where $C_2 \preceq C_1$ and $C_1$ has $a$ as protected attribute of integer type. This has made the theory much simpler than the Haore-logic based semantics for OO programming in [30].

The second case is to modify the value of an attribute of an object attached to an expression. This is done by finding the attached object in the system state $\Sigma$ and modifying its state accordingly. Thus, all variables that point to the identity of this object will be updated.

$$le.a{:=}e{=}_{df}\ \{\Sigma(\mathtt{decltype}(le))\}\colon\ \mathcal{D}(le.a{:=}e){\vdash}\begin{pmatrix}\Sigma(\mathtt{decltype}(le))'{=}\Sigma(\mathtt{decltype}(le))\\ \uplus\ (\{\mathtt{value}(le)\}{\oplus}\{a{\mapsto}\mathtt{value}(e)\})\end{pmatrix}$$

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. $x.d.a := 4$ will change state of $x = \langle 1, C, \{d \mapsto 2\}\rangle$, where reference $2$ is the identity of $\langle 2, D, \{a \mapsto 3\}\rangle$ to $x = \langle 1, C, \{d \mapsto 2\}\rangle$, but the $2$ is now the identity of the object $\langle 2, D, \{a \mapsto 4\}\rangle$.

This semantic definition shows the side-effect of an assignment and does reflect the OO feature pointed out by Broy in [7] that an invocation to a method of an object which contains such an assignment or an instance creation defined later on, changes the state $\Sigma$ of the system.

**Law 1** $(le := e; le := f(le)) \equiv (le := f(e))$

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

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

$$x.a := 1;\ y.a := 2\ \equiv\ y.a = 2;\ x.a = 1$$

**Object creation** The execution of $C.new(x)[\underline{e}]$ is well-defined if $C \in \texttt{cname}$, the length of the list $\underline{e}$ of the expressions is the same as the number of attributes of $C$ and the types of the expressions match those of the corresponding attributes of $C$, i.e.

$$\mathcal{D}(C.new(x)[\underline{e}]) =_{df} C{\in}\texttt{cname} \wedge |\underline{e}|{=}size(\texttt{attr}(C)) \wedge \forall i{\cdot}\texttt{type}(e_i){\preceq}\texttt{decltype}(C.a_i)$$

The command (re-)declares variable $x$, creates a new object, attaches the object to $x$ and attaches the initial values of the attributes to the attributes of $x$ too.

$$\begin{array}{l}
C.new(x)[\underline{e}] =_{df} \{\texttt{var}, x, \Sigma(C)\} : \\
\mathcal{D}(C.new(x)[\underline{e}]) \vdash \exists ref \notin Ref(\Sigma)\cdot \\
\left(\begin{array}{l}
(\Sigma(C)' = \Sigma(C) \cup \{\langle ref, C, \{a_i \mapsto \texttt{value}(e_i)\}\rangle \mid a_i \in \texttt{attr}(C)\})\wedge \\
((x \in \texttt{glb} \wedge (x' = ref) \\
\vee \\
(x{\in}\texttt{locvar} \wedge (\overline{x}'{=}\langle ref\rangle{\cdot}\overline{x})) \wedge (\texttt{locvar}'{=}\{x\}{\trianglerighteq}\texttt{locvar}{\cup}\{(x, \langle C\rangle{\cdot}\texttt{locvar}(x))\}) \\
\vee \\
(x \notin \texttt{var} \wedge (\overline{x}' = \langle ref\rangle) \wedge (\texttt{locvar}' = \texttt{locvar} \cup \{(x, \langle C\rangle)\})))
\end{array}\right)
\end{array}$$

We will use $C.new(x)$ to denote the command $C.new(x)[\underline{\texttt{Initial}(C.a)}]$ that creates an instance of $C$ with the default initial values of its attributes.

**Law 3** *If $x$ and $y$ are distinct, $x$ does not appear in $\underline{f}$ and $y$ does not appear in $\underline{e}$,*

$$C_1.new(x)[\underline{e}]; C_2.new(y)[\underline{f}] \equiv C_2.new(y)[\underline{f}]; C_1.new(y)[\underline{e}]$$

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

$$C.new(x)[\underline{e}]; (P \lhd b \rhd Q) \equiv (C.new(x)[\underline{e}]; P) \lhd b \rhd (C.new(x)[\underline{e}]; Q)$$

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

$$\begin{array}{l}
le.m(v, r, vr) =_{df} (\mathcal{D}(le) \wedge \texttt{type}(le) \in \texttt{cname} \wedge m \in \texttt{op}(\texttt{type}(le)) \Rightarrow \\
\exists N \cdot (\texttt{type}(le) = N) \wedge \left(\begin{array}{l}
\texttt{var}\ N\ self = le, T_1\ x = v, T_2\ y = r, T_3\ z = vr; \\
\Psi(N.m);\ r, vr := y, z; \\
\texttt{end}\ self, x, y, z
\end{array}\right)
\end{array}$$

where $x, y, z$ are resp. value, result and value-result parameters of the method $m$ of class $\texttt{type}(le)$, and $\Psi(N.m)$ stands for the design associated with method $m$ of class $N$, that will be defined in the semantics of the whole program in Section 3.6.

## 3.5 Class declarations

A class declaration *cdecl* given in Section 2.1 is well-defined if the following conditions hold.

1. $N$ has not been declared before: $N \notin \texttt{cname}$.
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.

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

$$
\begin{array}{l}
cdecl =_{df} \\
\{\mathtt{cname}, \mathtt{pricname}, \mathtt{superclass}, \mathtt{pria}, \mathtt{prota}, \mathtt{puba}\} : \mathcal{D}(cdecl) \vdash \\
\left(\begin{array}{l}
\quad \mathtt{cname}' = \mathtt{cname} \cup \{N\} \\
\wedge\ \mathtt{pricname}' = (\mathtt{pricname} \cup \{N\} \lhd anno(cdecl) \rhd \mathtt{pricname}) \\
\wedge\ \mathtt{superclass}' = \mathtt{superclass} \oplus \{N \mapsto M\} \\
\wedge\ \mathtt{pria}' = \mathtt{pria} \oplus \{N \mapsto \{\langle \underline{u} : \underline{U}, \underline{a} \rangle\}\} \\
\wedge\ \mathtt{prota}' = \mathtt{prota} \oplus \{N \mapsto \{\langle \underline{v} : \underline{V}, \underline{b} \rangle\}\} \\
\wedge\ \mathtt{puba}' = \mathtt{puba} \oplus \{N \mapsto \{\langle \underline{w} : \underline{W}, \underline{c} \rangle\}\} \\
\wedge\ \mathtt{op}' = \mathtt{op} \oplus \{N \mapsto \{(m_1 \mapsto (\underline{paras})_1, c_1)), \cdots\cdots, (m_\ell \mapsto (\underline{paras}_\ell, c_\ell))\}\}
\end{array}\right)
\end{array}
$$

where the logical variables `pria`, `prota` and `puba` are introduced to record the declared attributes of $N$, from which the state `attr` can later be constructed. Similarly, the dynamic behaviour of the methods cannot be defined before the dependency relation among classes is specified. At the moment, the logical variable $\mathtt{op}(N)$ binds each method $m_i$ to code $c_i$ rather than its definition which will be calculated in the end of the declaration section.

**Example** Consider a simple bank system illustrated by the UML class diagram in Figure 3. *Account* is an abstract class[1] and has two subclasses of current accounts *CA* and saving accounts *SA*. The declaration of class *Account*, denoted by *declAccount*,

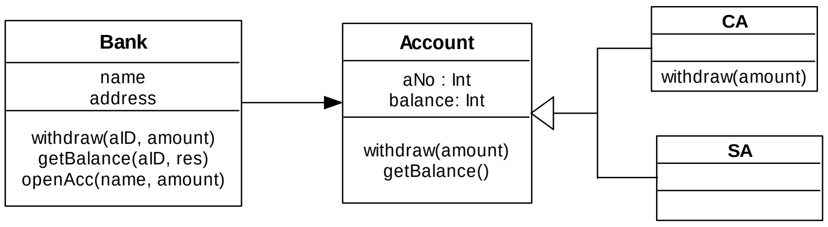

**Fig. 3.** A bank system

is written as follows. Note that we allow specification notations (designs) to appear in methods and commands.

$$
\begin{array}{l}
\textbf{class}\ Account\ \{ \\
\textbf{protected} : Int\ aNo, Int\ balance; \\
\textbf{method} : getBal(\emptyset, Int\ b, \emptyset)\ \{b := balance\}; \\
\qquad\qquad withdraw(Int\ x, \emptyset, \emptyset)\{balance \geq x \vdash balance' = balance - x\}\}
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{class}\ CA\ \textbf{extends}\ Account \\
\textbf{method} : withdraw(Int\ x, \emptyset, \emptyset)\{balance := balance - x\} \\
\}
\end{array}
$$

[1] See [25] for a formal definition of an abstract class.

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

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

$$\begin{array}{l} \mathit{declAccount} = \mathit{true} \vdash \\ \left(\begin{array}{l} \mathtt{cname}' = \mathtt{cname} \cup \{\mathit{Account}\} \wedge \mathtt{prot}' = \{\mathit{Account} \mapsto \{\langle \mathit{Int\ aNo}\rangle, \langle \mathtt{int}\ \mathit{balance}\rangle\}\} \wedge \\ \mathtt{op}' = \{\mathit{Account} \mapsto \{\mathit{getBal} \mapsto (\langle \emptyset, \mathit{Int}\ b, \emptyset\rangle, b' = \mathit{balance}), \\ \mathit{withdraw} \mapsto (\langle \mathit{Int}\ x, \emptyset, \emptyset\rangle, \mathit{balance} \geq x \vdash \mathit{balance}' = \mathit{balance} - x)\}\} \end{array}\right) \end{array}$$

The semantics of *declCA* is the following.

$$\mathit{declCA} = \mathit{true} \vdash \left(\begin{array}{l} \mathtt{cname}' = \mathtt{cname} \cup \{\mathit{CA}\} \wedge \\ \mathtt{superclass}' = \{\mathit{CA} \mapsto \mathit{Account}\} \wedge \\ \mathtt{op}' = \{\mathit{Account} \mapsto \{\mathit{withdraw} \mapsto \\ (\langle \mathit{Int}\ x, \emptyset, \emptyset\rangle, \mathit{balance}' = \mathit{balance} - x)\}\} \end{array}\right)$$

The semantics of *declSA* and *declBank* for classes *SA* and *Bank* can be defined in the same way.

A class declaration section *cdecls* comprises a sequence of class declarations. Its semantics is defined from the semantics of a single class declaration given above, and the semantics of sequential composition. However, the following well-definedness conditions need to be enforced onto a declaration section:

1. All class names used must be declared in the declaration section;
2. Any superclass of a declared class is declared too;
3. The function `superclass` does not induce circularity;
4. No attributes of a class can be redefined in its subclasses;
5. No method is allowed to redefine its signature in its subclass.

The formal definitions for these conditions are omitted here due to page limitation. In what follows we denote them as $\mathcal{D}_1, \ldots, \mathcal{D}_5$, respectively.

### 3.6 The semantics of a program

Let *cdecls* be a class declaration section and *P* main method of the form $(\mathtt{glb}, c)$, the meaning of a program $(\mathit{cdecls} \bullet P)$ is defined as the composition of the meaning of class declarations *cdecls* (defined in Section 3.5), the design *init*, and the meaning of command *P*:

$$\mathit{cdecls} \bullet P =_{df} \mathit{cdecls};\ \mathit{init};\ \mathtt{cname}' = \mathtt{pricname} \trianglerighteq \mathtt{cname};\ c$$

where the design *init* performs the following tasks

1. to check the well-definedness of the declaration section,
2. to decide the values of `attr` and `visibleattr` from those of `pria`, `prota` and `puba`,
3. to define the meaning of every method body $c$,

4. to check the well-definedness of glb, i.e. its consistency with the class declarations:

$$\mathcal{D}(\texttt{glb}) =_{df} \forall (x:T) \in \texttt{glb} \cdot T \in (\mathcal{T} \cup (\texttt{pricname} \trianglerighteq \texttt{cname}))$$

The design *init* is formalised as:

$$\begin{array}{l} \mathit{init} =_{df} \{\texttt{visibleattr}, \texttt{attr}, \texttt{op}\} : \\ \mathcal{D}_1 \wedge \mathcal{D}_2 \wedge \mathcal{D}_3 \wedge \mathcal{D}_4 \wedge \mathcal{D}_5 \wedge \mathcal{D}(\texttt{glb}) \vdash \\ \left( \begin{array}{l} \quad \texttt{visibleattr}' = \bigcup\limits_{N \in \texttt{cname}} \{N.a \mid a \in \texttt{puba}(N)\} \\ \wedge\ \forall N \in \texttt{cname} \cdot \texttt{attr}'(N) = \texttt{pria}(N) \cup \bigcup \{\texttt{prota}(M) \cup \texttt{puba}(M) \mid N \preceq M\} \\ \wedge\ \texttt{op}'(N) = \{m \mapsto (\underline{paras}, \Psi(N.m)) \mid (m \mapsto (\underline{paras}, c)) \in \texttt{op}(M) \wedge N \preceq M\} \end{array} \right) \end{array}$$

where the family of designs $\Psi(N.m)$ is defined in the rest of this section.

The family of designs $\Psi(N.m)$ captures the dynamic binding and is defined by a set of recursive equations, which contains for each class $N \in \texttt{cname}$, each class $M$ such that $N \preceq M$, and every method $m \in \texttt{op}(M)$ and equation

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

where $F$ is constructed according to the following rules:

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

$$F_{N.m}(\Psi) =_{df} \mathit{Set}(N); \phi_N(\mathbf{body}(M.m)); \mathit{Reset}$$

where the design $\mathit{Set}(N)$ finds out all attributes visible to class $N$ in order for the invocation of method $m$ of $N$ to be executed properly, whereas *Reset* resets the environment to be the set of variables that are accessible to the main program only:

$$\begin{array}{l} \mathit{Set}(N) =_{df} \{\texttt{visibleattr}\} : \mathit{true} \vdash \\ \texttt{visibleattr}' = \left( \begin{array}{l} \{N.a \mid a \in \texttt{pria}(N)\} \cup \bigcup_{N \preceq M} \{M.a \mid a \in \texttt{prota}(M)\} \cup \\ \bigcup_{M \in \texttt{cname}} \{M.a \mid a \in \texttt{puba}(M)\} \end{array} \right) \end{array}$$

$$\mathit{Reset} =_{df} \{\texttt{visibleattr}\} : \mathit{true} \vdash \texttt{visibleattr}' = \bigcup_{M \in \texttt{cname}} \{M.a \mid a \in \texttt{puba}(M)\}$$

The function $\phi_N$ renames the attributes and methods of class $N$ in the code $\mathbf{body}(N.m)$ by adding object reference *self* that represents the *active* object that is executing its method. The definition of $\phi_N$ is given in Fig. 4. Note that *Set* and *Reset* are used to ensure data encapsulation that is controlled by visibleattr and the well-formedness condition of an expression.

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

$$F_{N.m}(\Psi) =_{df} \mathit{Set}(N); \phi_N(\mathbf{body}(N.m)); \mathit{Reset}$$

| $P$ | $\phi_N(P)$ | $P$ | $\phi_N(P)$ |
|---|---|---|---|
| *skip* | *skip* | *chaos* | *chaos* |
| $P_1 \lhd b \rhd P_2$ | $\phi_N(P_1) \lhd \phi_N(b) \rhd \phi_N(P_2)$ | $P_1; P_2$ | $\phi_N(P_1); Set(N); \phi(P_2)$ |
| $P_1 \sqcap P_2$ | $\phi_N(P_1) \sqcap \phi_N(P_2)$ | $b * P$ | $\phi_N(b) * (\phi_N(P); Set(N))$ |
| $\texttt{var}\ x : T = e$ | $\texttt{var}\ x : T = \phi_N(e)$ | $\texttt{end}\ x$ | $\texttt{end}\ x$ |
| $C.new(x)$ | $C.new(\phi_N(x))$ | $le := e$ | $\phi_N(le) := \phi_N(e)$ |
| $le.m(v, r, vr)$ | $\phi_N(le).m(\phi_N(v), \phi_N(r), \phi_N(vr))$ | $le.a$ | $\phi_N(le).a$ |
| $m(v, r, vr)$ | $self.m(\phi_N(v), \phi_N(r), \phi_N(vr))$ | *null* | *null* |
| *self* | *self* | $f(e)$ | $f(\phi_N(e))$ |
| $x$ | $\begin{cases} self.x, & x \in \bigcup_{N \preceq M} \texttt{attrname}(M) \\ x, & otherwise \end{cases}$ | | |

**Fig. 4.** The Definition of $\phi_N$

## 4 Refinement

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

1. Refinement relation between object systems.
2. Refinement relation between declaration sections.
3. Refinement relation between commands.

From now on, we assume the main method of each program does not use direct field access, that is, expressions of the form *le.a*. This assumption actually does not reduce the expressiveness of the language, as we can always use *getField* and *setField* methods to replace direct field access where necessary. In what follows, we give formal definitions for the above-mentioned refinement relations.

**Definition 4.** *Let $S_1$ and $S_2$ are object programs which have the same set of global variables* $\texttt{glb}$*, let $Ivar_{S_i}$, $i = 1, 2$, be the set of all other variables that are free in $P_i$ and $Ivar'_{S_i}$ be the set of their primed versions. $S_1$ is a* refinement *$S_2$, denoted by $S_1 \sqsupseteq_{sys} S_2$, if its behaviour is more controllable and predictable than that of $S_2$:*

$$S_1 \sqsupseteq_{sys} S_2 =_{df} \forall \texttt{glb}, \texttt{glb}' \cdot (\exists Ivar_{S_1}, Ivar'_{S_1} \cdot S_1) \Rightarrow (\exists Ivar_{S_2}, Ivar'_{S_2} \cdot S_2)$$

This indicates the external behaviour of $S_1$, that is, the pairs of pre- and post global states, is a subset of that of $S_2$.

**Definition 5.** *Let* $\text{cdecls}_1$ *and* $\text{cdecls}_2$ *be two declaration sections. We say* $\text{cdecls}_1$ *is a* refinement *of* $\text{cdecls}_2$*, denoted by* $\text{cdecls}_1 \sqsupseteq_{class} \text{cdecls}_2$*, if the former can replace the later in any object system:*

$$\text{cdecls}_1 \sqsupseteq_{class} \text{cdecls}_2 =_{df} \forall P \cdot (\text{cdecls}_1 \bullet P \sqsupseteq_{sys} \text{cdecls}_2 \bullet P)$$

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

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

**Definition 6.** *Let $P_1$ and $P_2$ be main methods with the same global variables, and $c_1$ and $c_2$ be commands. We define*

$$P_1 \sqsupseteq_{cmd} P_2 =_{df} \forall \text{cdecls} \cdot (\text{cdecls} \bullet P_1) \sqsupseteq_{sys} (\text{cdecls} \bullet P_2)$$
$$c_1 \sqsupseteq c_2 =_{df} \forall \underline{v}, \underline{v}' \cdot (c_1 \Rightarrow c_2)$$

*where* cdecls *is a declaration section, and $\underline{v}$ and $\underline{v}'$ are free variables in $c_1$ and $c_2$.*

Intuitively, it denotes that $P_1$ does better than $P_2$, i.e. ensures a stronger postcondition with a weaker precondition, under the same environment.

We have already given some refinement laws for refining program commands in Section 3.4, which are to ensure the correctness of the semantic model. In what follows, we first give a group of refinement laws that in fact formalize principles of refactoring [13]. After that, we will present three refinement laws which capture three key principles and patterns in object-oriented design, that are well known as the *Expert Pattern, High Cohesion Pattern* and *Low Coupling Pattern* [22, 24].

We first introduce some notations. We use $N[supc, pri, prot, pub, ops]$ to denote a well-formed class declaration that declares the class $N$ that has $supc$ as its direct superclass; $pri$, $prot$ and $pub$ as its sets of private, protected and public attributes; and $ops$ as its set of methods. $supc$ 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 also only refer to some, or even none of $M$, $pri$, $prot$, $pub$, $ops$ 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 5** *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, but it is not so obvious for a semantic definition of a class declaration to guarantee this law. For example, if the the pre-condition of the class declaration requires that the direct superclass has been declared, this law would not hold.

The next law says that more services may come from more classes.

**Law 6** *If a class name $N$ is not in* cdecls, cdecls $\sqsubseteq N[M, pri, prot, pub, ops]$; cdecls.

Introducing a private attribute has no effect.

**Law 7** *If neither $N$ nor any of its superclasses and subclasses in* cdecls *has $x$ as an attribute* $N[pri]$; cdecls $\sqsubseteq N[pri \cup \{T\ x = d\}]$; cdecls.

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

**Law 8** $N[pri \cup \{T\ x = d\}, prot]$; cdecls $\sqsubseteq N[pri, prot \cup \{T\ x = d\}]$; cdecls.

Similarly, changing a protected attribute to a public attribute refines the declaration too.

Adding a new method can refine a declaration.

**Law 9** *If $m$ is not defined in $N$, let $m(paras)\{c\}$ be a method with distinct parameters $paras$ and a command $c$. Then*

$$N[ops]; \text{cdecls} \sqsubseteq N[ops \cup \{m(paras)\{c\}\}]; \text{cdecls}$$

*provided that there is no superclass of $N$ in* cdecls*.*

**Law 10** *We can refine a method to refine a declaration. If $c_1 \sqsubseteq c_2$,*

$$N[ops \cup \{m(paras)\{c_1\}\}]; \text{cdecls} \sqsubseteq N[ops \cup \{m(paras)\{c_2\}\}]; \text{cdecls}$$

Inheritance introduces refinement.

**Law 11** *If none of the attributes of $N$ is defined in $M$ or any superclass of $M$ in* cdecls*,*

$$N[\emptyset, pri, prot, pub, ops]; \text{cdecls} \sqsubseteq N[M, pri, prot, pub, ops]; \text{cdecls}$$

We can introduce a superclass as given in the following law.

**Law 12** *Let $C_1 = N[\emptyset, pri \cup S, prot, pub, ops]$, $C_2 = N[\{M\}, pri, prot, pub, ops]$. Assume $M$ is not declared in* cdecls*,*

$$C_1; \text{cdecls} \sqsubseteq C_2; M[\emptyset, \emptyset, S, \emptyset, \emptyset]; \text{cdecls}$$

We can move some attributes of a class to its superclass.

**Law 13** *If all the subclasses of $M$ but $N$ do not have attributes in $S$, then*

$$M[prot_1]; N[\{M\}, prot \cup S]; \text{cdecls} \sqsubseteq M[prot_1 \cup S]; N[\{M\}, prot]; \text{cdecls}$$

We can move the common attributes of the direct subclasses of a class to the class itself.

**Law 14** *If $M$ has $N_1, \ldots, N_k$ as its direct subclasses,*

$$\begin{aligned} & M[prot]; N_1[prot_i \cup S]; \ldots; N_k[prot_k \cup S]; \text{cdecls} \\ \sqsubseteq\ & M[prot \cup S]; N_1[prot_1]; \ldots; N_k[prot_k]; \text{cdecls} \end{aligned}$$

We can move some methods of a class to its superclass.

**Law 15** *Let $m(paras)\{c\}$ be a methods of $N$, but not a method of its direct superclass $M$. Assume that $c$ only involves the protected attributes of $M$, then*

$$\begin{aligned} & M[ops]; N[\{M\}, ops_1 \cup \{m(\underline{paras})\{c\}\}]; \text{cdecls} \\ \sqsubseteq\ & M[ops \cup \{m(\underline{paras})\{c\}\}]; N[\{M\}, ops_1]; \text{cdecls} \end{aligned}$$

We can remove a redundant method from a subclass.

**Law 16** *Assume that $N$ has $M$ as its direct superclass and $m(paras)\{c\} \in ops \cap ops_1$, and $c$ only involves the protected attributes of $M$,*

$$M[ops]; N[\{M\}, ops_1]; \text{cdecls} \sqsubseteq M[ops]; N[\{M\}, ops_1 \backslash \{m(\underline{paras})\{c\}\}]; \text{cdecls}$$

We can remove any unused private attributes.

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

$$N[pri]; \text{cdecls} \sqsubseteq N[pri \setminus \{T\ x = d\}]; \text{cdecls}$$

We can remove any unused protected attributes.

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

$$N[prot]; \text{cdecls} \sqsubseteq N[prot \setminus \{T\ x = d\}]; \text{cdecls}$$

The expert patterns says that a class is allowed to delegate some tasks to its associated classes that contain the information for the tasks.

**Law 19** (**Expert Pattern for Responsibility Assignment**) *Suppose $M[ops_1]$ is defined in cdecls, $m(paras_1)\{c_1\} \in ops_1$, and $(M\ o)$ be an attribute of $N$, then*

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

*Here $c_1$ is obtained from $\tilde{c_1}$ by replacing $o.x$ with $x$, that is, $c_1 = \tilde{c_1}[x/o.x]$. Notice 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 $\tilde{c_1}$. Note also that $paras_1 \subseteq paras$.*

This law is illustrated by the UML class diagram in Figure 5. It will become an equation if $x$ is a public attribute $M$.

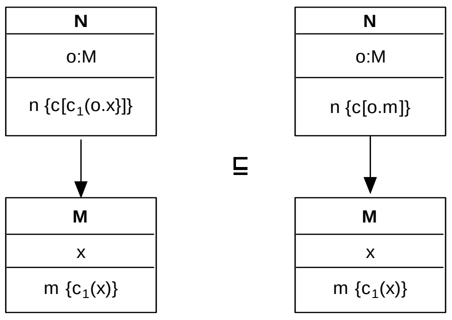

**Fig. 5.** Object-oriented Functional decomposition

To understand the above law, let us consider a simple example from the aforementioned bank system in Section 3.5.

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

$$getBalance(Int\ aID, Int\ res, \emptyset) =_{df}$$
$$\exists a \in \Sigma(Account) \cdot a.aNo = aID \vdash \exists a \in \Sigma(Account) \cdot a.aNo = aID \Rightarrow res' = a.balance$$

Note that it requires the attributes of class *Account* to be visible (public) to other classes (like *Bank*). Applying Law 19 to it, we can get the following design:

$$\begin{array}{l} \mathit{getBalance}(\mathit{Int\ aID}, \mathit{Int\ res}, \emptyset) =_{df} \\ \exists a \in \Sigma(\mathit{Account}) \cdot a.aNo{=}aID \vdash \exists a \in \Sigma(\mathit{Account}) \cdot a.aNo{=}aID \Rightarrow res'{=}a.getBalance() \end{array}$$

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

It is important to note that method invocation, or in another term, object interaction takes time. Therefore, this object-oriented refinement (and the one described in Law 21 later) usually exchanges efficiency for "simplicity", 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. This is captured in the following law.

**Law 20** (**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*

$$M[pri, prot, pub]; cdecls \sqsubseteq M[pri, prot \cup \{T_1\ a_1 = d_1\}, pub \backslash \{T_1\ a_1 = d_1\}]; cdecls$$

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

$$M[pri, prot, pub]; cdecls \sqsubseteq M[pri \cup \{T_2\ a_2 = d_2\}, prot \backslash \{T_2\ a_2 = d_2\}, pub]; cdecls$$

After applying Law 19 exhaustively (i.e. the expert pattern) to the class *Bank* for method *getBalance*, we can then apply Law 20 to the class diagram on the right hand side of Figure 5 to 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 [22] by the following refinement rule.

**Law 21** (**High Cohesion Pattern**) *Assume* $M[pri, op]$ *is a well-formed class declaration,* $pri = \{x, y\}$ *are (or 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 though it does not have to change it at all). Then*

1. $M \sqsubseteq M[pri_{new}, op_{new}]; M_1[pri_1, op_1]; M_2[pri_2, op_2]$,
*where*

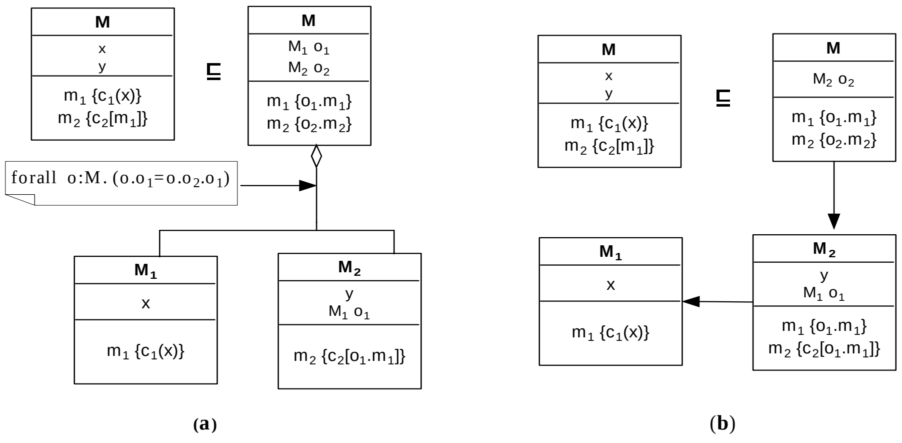

**Fig. 6.** Class Decomposition

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

*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 Figure 6(a).*

2. $M \sqsubseteq M[pri_{new}, op_{new}]; M_1[pri_1, op_1]; M_2[pri_2, op_2]$,
*where*

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

*such that* $p(o_1.x, y)$ *is an invariant of* $M_2$.

*This refinement is illustrated by the diagram in Figure 6(b).*

Notice that the first refinement in Law 21 requires that *M* to 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 $M_1$ are 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 [22]. The use of the patterns for responsibility assignment in object-oriented software development is clearly demonstrated in Larman's book [22] and in the lecture notes of Liu in [24].

For each of the laws, except for Law 13 in the Appendix, let *LHS* and *RHS* denote the declarations on the left and right hand sides, respectively. For any main program *P*, each refinement law becomes an equational law: $LHS \bullet P \equiv RHS \bullet P$, provided $LHS \bullet P$ is well-defined.

## 5 Conclusion

We have shown how Hoare and He's design calculus [18] is used to define an OO language. A program is represented as a predicate called a *design*, and the refinement relation between programs is defined as implication between predicates.

In [7], Broy gave an assessment of object-orientation. Our model reflects most of the features, no matter good or bad, of object-oriented designs. For example, the model does show that inheritance with attribute hiding and method overriding makes it difficult to analyse the system behaviour, and method invocation on an object may indeed change the system's global states.

Nevertheless, formal techniques for object-orientation have achieved significant advance in areas of both formal methods and object technology, e.g. [1, 2, 6, 4, 8, 29]. There are a number of recent articles on Hoare Logics for object-oriented programming (see, e.g. [30, 35, 20, 31, 23, 9]). The normal form of a program in our paper is similarly to that of [9, 30]. However, one major difference of our work is that we also provide a formal characterisation and refinement of the contextual/structural features, i.e. the declaration section, of an object program. This is motivated by our work on the formalisation and combinations of UML models [25, 26] to deal with consistency problems of different UML models. This characterisation has been proven to be very useful in defining semantics for integrated specification languages in general. For example, [32] uses this characterisation in defining a semantics of TCOZ.

The notions of different kinds of refinements in our model are very close to those in [9], though the semantics in [9] is defined in terms of the weakest precondition predicate transformer and does not deal with reference types. We take a *weak semantic* approach meaning that when the pre-condition of a contact is not satisfied in a state, the program will then behave as *chaos*, and any modification to the program, such as adding exceptional handling, will be a refinement to the program. We also describe static well-formedness conditions in the pre-condition so that any correction of any static inconsistency in a program, such as static type mismatching, missing variables, missing methods, etc. will be refinement too. This decision is required for *structural refinement calculus* of OO designs in order to treat *refactoring* [13] as refinement and properly combine it with *functional/behavioural refinement*. This combination is important for the application of the model to composing different UML models and to reasoning about their consistency [25, 26] and in giving semantics for integrated language [32]. Also our work on formal object-oriented design with UML [25, 26] has provided us with the insight of functional decomposition in the object-oriented setting and its relation with data encapsulation. The functional decomposition and data encapsulation are characterised by the refinement **laws** 19 - 20. They reflect the essential principle of object-oriented design.

The power of UTP[18] for describing different features of computing, such as state-based properties, communication, timing, higher-order computing [18, 36, 34], makes our approach ready for an extension to cope with these different aspects of object-oriented designs. Alternatively, one can also use temporal logic, such as [3], for the specification and verification of multithreading Java-like programs. However, we would like to deal with concurrency at a higher level when we extend this model for component-based software development [17, 16].

In [7], Broy also argued that the property of object identities is of too low level and implementation oriented. This is true to some extent and the use of references does cause some side-effects, making the semantics a bit more difficult. A preliminary version of the model without references can be found in [15]. However, that version is only slight simpler than this version. On the other hand, the complexity in fact mainly affects reasoning about low level design and implementation. At high level requirement analysis and design, we can simply use the identities as the objects they refer to or just talk about objects in an abstract way. In our approach for analysis of use cases [25], we mainly describe the change of system states 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. We think that features like method overriding and attribute hiding are only useful to program around the requirement and design defects detected at the coding stage or even after, or 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 program verification and the smooth application of the notion of program refinements.

Future work includes the study of the issue of completeness of the refinement calculus, applications to more realistic case studies, and formal treatment of *patterns* [14].

## References

1. M. Abadi and L. Cardeli. *A Theory of Objects*. Springer-Verlag, 1996.
2. M. Abadi and R. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference*, pages 682–696. Springer-Verlag, 1997.
3. E. Abraham-Mumm, F.S. de Boer, W.P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *Foundations of Software Science and Computation Structures, LNCS 2303*, pages 5–20. Springer, 2002.
4. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, illem P. de Roever, and G. Rozenberg, editors, *REX Workshop*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer, 1991.
5. P. America and F. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.
6. M.M. Bonsangue, J.N. Kok, and K. Sere. An approach to object-orientation in action systems. In J. Jeuring, editor, *Mathematics of Program Construction,* LNCS 1422, pages 68–95. Springer, 1998.
7. M. Broy. Object-oriented programming and software development - a critical assessment. In A. McIver and C. Morgan, editors, *Programming Methodology*. Springer, 2003.
8. D. Carrington, *et al*. *Object-Z: an Object-Oriented Extension to Z.* North-Halland, 1989.
9. A. Cavalcanti and D. Naumann. A weakest precondition semantics for an object-oriented language of refinement. In *LNCS 1709*, pages 1439–1460. Springer, 1999.
10. D. Coleman, *et al*. *Object-Oriented Development: the FUSION Method*. Prentice-Hall, 1994.
11. S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall, 1994.
12. E. Dürr and E.M. Dusink. The role of $VDM^{++}$ in the development of a real-time tracking and tracing system. In J. Woodcock and P. Larsen, editors, *Proc. of FME'93, LNCS 670*. Springer-Verlag, 1993.
13. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

14. E. Gamma, *et al*. *Design Patterns*. Addison-Wesley, 1995.
15. J. He, Z. Liu, and X. Li. Towards a refinement calculus for object-oriented systems (invited talk). In *Proc. ICCI02, Alberta, Canada*. IEEE Computer Society, 2002.
16. J. He, Z. Liu, and X. Li. A component calculus. In H.D. Van and Z. Liu, editors, *Proc. Of FME03 Workshop on Formal Aspects of Component Software (FACS03), UNU/IIST Technical Report 284, UNU/IIST, P.O. Box 3058, Macao,* Pisa, Italy, 2003.
17. J. He, Z Liu, and X. Li. Contract-oriented component software development. Technical Report 276, UNU/IIST, P.O. Box 3058, Macao SAR China, 2003.
18. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
19. I. Houston. Formal specification of the OMG core object model. Technical report, IMB, UK, Hursely Park, 1994.
20. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *FASE 2000, LNCS 1783*, pages 284–303. Springer, 2000.
21. K. Lano and H. Haughton. *Object-oriented specification case studies*. Prentice Hall, New York, 1994.
22. C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
23. K. Rustan M. Leino. Recursive object types in a logic of object-oriented programming. In *LNCS 1381*. Springer, 1998.
24. Z. Liu. Object-oriented software development in UML. Technical Report UNU/IIST Report No. 228, UNU/IIST, P.O. Box 3058, Macau, SAR, P.R. China, March 2001.
25. Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, LNCS 2885*, pages 641–664. Springer, 2003.
26. Z. Liu, J. He, X. Li, and J. Liu. Unifying views of UML. Research Report 288, UNU/IIST, P.O. Box 3058, Macao, 2003. Presented at UML03 Workshop on Compostional Verification of UML and submitted for the inclusion in the final proceedings.
27. B. Meyer. From structured programming to object-oriented design: the road to Eiffel. *Structured Programming*, 10(1):19–39, 1989.
28. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-orient programs. In *Proc of FME'97, LNCS*. Springer, 1997.
29. D. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olerog, editor, *Proc. of PROCOMET'94*. North-Holland, 1994.
30. C. Pierik and F.S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute of Information and Computing Science, Utrecht University, 2003.
31. A. Poetzsch-Heffter and P. Muller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Proc. Programming Languages and Systems (ESOP'99), LNCS 1576*, pages 162–176. Springer, 1999.
32. S. Qin, J.S. Dong, and W.N. Chin. A semantics foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods, LNCS 2805*, pages 321–340. Springer, 2003.
33. E. Sekerinski. A type-theoretical basis for an object-oriented refinement calculus. In *Proc. of Formal Methods and Object Technology*. Springer-Verlag, 1996.
34. A. Sherif and J. He. Towards a time model for Circus. In *ICFEM02, LNCS 2495*. Springer, 2002.
35. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.
36. J.C.P. Woodcock and A.L.C. Cavalcanti. A semantics of Circus. In *ZB 2002, LNCS 2272*. Springer, 2002.