The United Nations University

UNU/IIST

International Institute for Software Technology

A Relational Model for Object-Oriented Programming

He Jifeng, Liu Zhiming and Li Xiaoshan

May, 2001

UNU/IIST Report No. 231

UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2001

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

A Relational Model for Object-Oriented Programming

He Jifeng, Liu Zhiming and Li Xiaoshan

Abstract

This report presents a semantics for an object-oriented language with classes, visibility, dynamic binding, mutual recursive methods and recursion. Our semantic framework identifies both class declarations and commands as designs. All the programming constructs of our language, such as assignment, conditional, composition and recursion, are defined in the exactly same way as their counterparts in the imperative programming languages. This makes the approach more accessible to users who are already familiar with imperative program design and also enables the use of existing tools and methods of verification and refinement developed for these languages. Furthermore, the algebraic laws developed for the imperative languages remain applicable in designing object-oriented programs.

He Jifeng is a senior research-fellow of UNU/IIST, He is also a professor of computer science at East China Normal University and Shanghai Jiao Tong University. His research interests include the mathematical theory of programming and refined methods, the design techniques for the mixed software/hardware systems. Email: hjf@iist.unu.edu

Liu Zhiming is a visiting scientist at UNU/IIST, on leave from Department of Mathematics and Computer Science of the University of Leicester, England, where he is a lecturer in computer science. His research interests include theory of computing systems, formal methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems; and formal techniques for OO development. Email: Z.Liu@mcs.le.ac.uk

Li Xiaoshan is an Assistant Professor at the University of Macau. His research areas are Interval Temporal Logic, formal specification, verification and simulation of computer systems, formal methods in system design and implementation. Email: fstxsl@umac,mo

This work is partly supported by the British EPSRC Grant GR/M89447.

Contents

1 Introduction

It has been widely recognised that design and development of object-oriented programs is intricate. The need for formal basis of object-oriented development was identified by many researchers. This paper demonstrates how the programming theory of Hoare and He [9], can be used for constructing more reliable object-oriented programs. We present a semantics for an object-oriented language with classes, visibility, dynamic binding, mutually recursive methods and recursion. For simplicity, it forbids the redefinition of a variable within its own scope, and omits reference types, which have been investigated separately in [4, 10]. This language is sufficiently similar to Java and C++ to be used in meaningful case studies and to capture some of the central difficulties.

Our semantic framework identifies both class declarations and commands as designs. All the programming constructs of our language, such as assignment, conditional, composition and recursion, are defined in the exactly same way as their counterparts in the imperative programming languages. This makes the approach more accessible to users who are already familiar with imperative program design and also enables the use of existing tools and methods of verification and refinement developed for these languages. Furthermore, the algebraic laws developed for the imperative languages remain applicable in designing object-oriented programs.

In our approach, commands denote binary relations over states of variables, objects and classes. Variables of the primitive data types take the values of the corresponding type, while object states are represented by tuples of attribute values with recursive nesting, but no sharing. We treat a method as a design with a set of parameters, and class the tuple of the meanings of its methods. The state also contains necessary environmental information in support of the well-definedness check. In general, a program is modelled by a design of the form

$$WF \;\Rightarrow\; D$$

where the predicate WF establishes the well-definedness of the program, and the design D acts as its behavioural specification. This treatment integrates the dynamic type and visibility check mechanism with the traditional specification-oriented semantics.

In this paper we adopt a compositional approach to specify the properties of classes and the meanings of their methods. Like commands, class declarations denote binary relations over states, where each declaration provides contextual information for the structure of the class and its relationship with the others. The composition of class declarations simply merges its components when they are consistent, otherwise it behaves like a chaotic design. The whole class declaration sector gives rise to a design which initialises the state of variables, objects and classes.

We build a logic of object-oriented programs as a conservative extension of standard predicate logic, in the style of [9]. An alternative approach is undertaken by Cavalcanti and Naumann in [6, 14], where they define the semantics of an Oberon-like programming language based on predicate transformer, and employ the standard technique in type theory and denotational

semantics; the semantics is defined in terms of typing derivations. Abadi and Leino [2] develop a logic of object-oriented programs in the style of Hoare, prove its soundness, and discuss completeness issues. Sekerinski [16] defines a rich object-oriented programming by using a type system with subtyping and also using predicate transformer. Mikhajlova uses sum type in [13] instead, as suggested by Back and Butler in [5] to avoid the complications in the typing and the logic when reasoning about the record types. Rustan and Leino [15] have extended existing refinement calculi with object-oriented features, but restricting inheritance or not dealing with class and visibility. However, those works deal with sharing and concurrency.

This paper is organised as follows. Section 2 provides a computation model for object-oriented programs, and explores the structure and context of states. Section 3 deals with expressions, and examines their well-definedness. We devote Section 4 to the formalisation of commands. Section 5 investigates class declarations, and shows how the parallel merge operator in [9] can be employed in defining the meaning of composite declarations. We discuss the algebraic laws of the language in Section 6. The paper concludes with a short discussion.

2 Computation model

In [9], an imperative program is identified as a binary relation $(\alpha,\ P)$, where

- α denotes the set of program variables known to the program.
- P is a predicate which relates the initial values of program variables to their final values, and takes the form of $p(x) \vdash R(x,\ x')$ (called a *design* in [9])

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

  where
  - the *pre-condition* $p(x)$ must be true before the program starts.
  - the *post-condition* $R(x,\ x')$ holds when the program terminates.

  where x and x' represent the initial and final values of program variables x respectively.

That simplified model makes the assumption that there is a universal data type, and ignores the issues of dynamic typing and data accessibility. Clearly, it can not cope with many characteristic features of object-oriented program development.

To formalising the behaviour of object-oriented programs, the computation model, we adopt in this paper, takes into account the following features:

1. An object-oriented program operates not only on (global and local) variables but objects. To ensure legal access to variable, the model is equipped with the set of visible attributes in addition to the set of global and local variables known to the program.

2. Due to the subclass mechanism, an object can lie in any subclass of its original declared one. As a result, the behaviour of its methods will depend on its current type. To support such a dynamic binding mechanism for method call, our model keeps track of the dynamic type for each object. This enables us to validate expressions and commands in a framework where the type of each variable is properly recorded.

3. As in the imperative languages, the state of a variable is simply the value it takes at the moment. Because an object may contain attributes of object type, its value will oftenly be represented by a finite tuple , which records the current type of the object, and the values of its attributes (probably with recursive nesting).

In our model, an object-oriented program denotes a binary relation (α, P), where the predicate P relates the state before the activation of the program to the state after termination, α identifies the environment in which the program executes. The environment α can be divided into three categories:

1. The first category provides the contextual information on classes and their relationships:

   - **CName**: the set of classes which are already declared,
   - **superclass**: the partial function which maps a class to its *direct* superclass, i.e. **superclass**$(M) = N$ states that N is a direct superclass of M.

2. The second category describes the structure of each class in detail: for every class name $N \in$ **CName**, it includes

   - **attribute**(N): the set of (declared or inherited) attributes of class N

     $$\{< a_1 : U_1,\ c_1 >, \ldots, < a_m : U_m,\ c_n >\}$$

     where U_i and c_i stand for the type and the initial value of attribute a_i of class N, and will be referred by $type(N.a_i)$ and $initial(N.a_i)$ in the later discussion.

   - **method**(N): the set of methods declared or inherited by N together with their signatures and specifications

     $$\begin{aligned}&\{m_1 \mapsto\ (< x_1 : T_{1,1},\ y_1 : T_{1.2},\ z_1 : T_{1.3} >,\ D_1),\\ &\qquad\qquad \ldots\ldots\\ &m_k \mapsto\ (< x_k : T_{k,1},\ y_k : T_{k.2},\ z_k : T_{k.3} >,\ D_k)\}\end{aligned}$$

     which indicates that the method m_i has x_i, y_i and z_i as its value, result and value-result parameters respectively, which will be referred by $valparameter(N.m_i)$, $resparamater(N.m_i)$ and $valresparameter(N.m_i)$ later. The behaviour of a method m_i is captured by the design D_i.

3. The third category identifies those variables that are accessible by the program:

   - **alphabet**: the set of global variables known to the program

     $$\{x_1 : T_1, \ldots, x_n : T_n\}$$

where T_i represents the type of variable x_i, and it can be either a built-in data type (such as the Boolean type $\mathcal{B}$ or the real numbers $\mathcal{R}$), or a class name, and will be referred by $type(x_i)$ in the later discussion. For notational simplicity we will abuse the notation by writing $x \in$ **alphabet** to abbreviate

$$x \in \pi_1(\textbf{alphabet})$$

where the projection π_1 maps each pair to its first element. A similar interpretation will apply to the cases when we deal with local variables and attributes.

- **locvar**: the set of local variables in scope

  $$\{v_1 : T_1, ..., v_m : T_m\}$$

- **visibleattr**: the set of attributes which are visible from inside the current class, i.e, all the declared private and the (either declared or inherited) public and protected attributes of the class.

For notational convenience, we assume the existence of four disjoint sets of names for variable names, class names, attribute names and method names.

A state of our model binds variables in **alphabet** and **locvar** to their current values. A variable of a primitive data type can take any value of that type. If a variable is an object, its value will be a tuple of the values of its attributes together with its current type:

$$\{myclass \mapsto M\} \cup \{a \mapsto value \mid a \in \textbf{attribute}(M)\}$$

In summary, a state of our model consists of the values of logical variables

$$< \textbf{alphabet}, \textbf{locvar}, \textbf{CName}, \textbf{superclass}, \textbf{attribute}, \textbf{visibleattr}, \textbf{method} >$$

and program variables in the sets **alphabet** and **locvar**.

Notations

Let V be a set of variables. A design with the frame V has the following form

$$V : (p \vdash R)$$

which denotes the predicate

$$(p \vdash (R \wedge \underline{w}' = \underline{w}))$$

where the list $\underline{w}$ contains all (logical and program) variables but those in the set V.

N is said to be a superclass of M, if there exists a finite family $\{N_i \mid 0 \leq i \leq n\}$ of class names such that $M = N_0$ and $N = N_n$ and

$$\textbf{superclass}(N_i) = N_{i+1} \text{ for all } 0 \leq i < n.$$

We introduce a binary relation $\leq$ over types as follows. $T1 \leq T2$ if either $T2$ is a superclass of $T1$ or $T1 = T2$. In general, let $\underline{T}$ and $\underline{U}$ be list of types, we define

$$\underline{T} \leq \underline{U} =_{df} \#\underline{T} = \#\underline{U} \wedge \forall i : 1..n \bullet (T_i \leq U_i)$$

where $\#s$ denotes the length of s. and $n =_{df}$ **min**$(\#\underline{T}, \#\underline{U})$

3 Expression

This section deals with expressions, and illustrates how to validate an expression and determine its value. An expression can be in one of the following forms:

$$e ::= x \mid \mathbf{null} \mid \mathbf{new}\, N \mid e\ \mathbf{is}\ N \mid (N)e \mid e.a \mid (e;\, a : f) \mid f(e) \mid \mathbf{self}$$

To validate an expression e, we introduce a predicate $\mathcal{D}(e)$, which is true just in those circumstances in which e can be successfully evaluated.

A variable x is well-defined if it is known to the program.

$$\mathcal{D}(x) =_{df} x \in (\mathbf{alphabet} \cup \mathbf{locvar})$$

The notation **null** represents a well-defined object.

$$\mathcal{D}(\mathbf{null}) =_{df} true$$

$$type(\mathbf{null}) =_{df} NULL$$

where $NULL$ is a reserved class name. We adopt the convention that

$$NULL \leq N \text{ for all } N \in \mathbf{CName}.$$

The expression $\mathbf{new}\, N$ is well-defined if the class N is already declared.

$$\mathcal{D}(\mathbf{new}\, N) =_{df} N \in \mathbf{CName}$$

It denotes a newly created object of class N.

$$type(\mathbf{new}\, N) =_{df} N$$

$$\mathbf{new}\, N =_{df} \{myclass \mapsto N\} \cup \bigcup_{a \in \mathbf{attribute}(N)} \{a \mapsto initial(N.a)\}$$

The type test $e\ \mathbf{is}\ N$ is a well-defined (boolean) expression when N is an already declared class, and e is a well-defined object.

$$\mathcal{D}(e\ \mathbf{is}\ N) =_{df} (N \in \mathbf{CName}) \wedge \mathcal{D}(e) \wedge (type(e) \in \mathbf{CName})$$

$$type(e\ \mathbf{is}\ N) =_{df} \mathcal{B}$$

The value of the expression is determined by whether the value of e is an object of class N.

$$(e\ \mathbf{is}\ N) =_{df} (e \neq \mathbf{null}) \wedge (e(myclass) \leq N)$$

The type cast $(N)e$ is equal to e if it is a non-null object of a subclass of N.

$$\mathcal{D}((N)e) =_{df} \mathcal{D}(e\,\mathbf{is}\,N) \wedge (e \neq \mathbf{null}) \wedge (e(myclass) \leq N)$$

$$(N)e =_{df} e$$

The type of $(N)e$ is the class N.

$$type((N)e) =_{df} N$$

The attribute selection $e.a$ is well-defined when e is a non-null object, and its a attribute is visible from inside the current class.

$$\mathcal{D}(e.a) =_{df} \mathcal{D}(e) \wedge (type(e) \in \mathbf{CName}) \wedge (e \neq \mathbf{null}) \wedge e(myclass).a \in \mathbf{visibleattr}$$

$$e.a \ =_{df} \ e(a)$$
$$type(e.a) \ =_{df} \ type(e(myclass).a)$$

The attribute update $(e;\ a : f)$ is well-defined provided that $e.a$ is visible, and the type of f is a subtype of $e.a$.

$$\mathcal{D}(e;\ a : f) \ =_{df} \ \mathcal{D}(e.a) \ \wedge \ \mathcal{D}(f) \ \wedge \ (type(f) \leq type(e(myclass).a))$$

The type of $(e;\ a : f)$ is the class of e, and its value can be obtained from the value of e by changing the value of its a attribute to the value of f.

$$type(e;\ a : f) \ =_{df} \ type(e)$$
$$(e;\ a : f) \ =_{df} \ e \oplus \{a \mapsto f\}$$

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

$$\mathcal{D}(e/f) \ =_{df} \ \mathcal{D}(e) \ \wedge \ (type(e) = \mathcal{R}) \ \wedge \mathcal{D}(f) \ \wedge \ (type(f) = \mathcal{R}) \ \wedge \ (f \neq 0)$$
$$\mathcal{D}(e \, and \, f) \ =_{df} \ \mathcal{D}(e) \ \wedge \ (type(e) = \mathcal{B}) \ \wedge \mathcal{D}(f) \ \wedge \ (type(f) = \mathcal{B})$$

The variable **self** can only be referred in the definition of methods, and is treated as a local variable in our model.

$$\mathcal{D}(\mathbf{self}) \ =_{df} \ \mathbf{self} \in \mathbf{locvar}$$

Let $\underline{e}$ be a list of expressions. Define

$$\mathcal{D}(\underline{e}) \ =_{df} \ \forall i : 1..\#\underline{e} \bullet \mathcal{D}(\underline{e}[i])$$

4 Commands

This section deals with commands. Our language supports typical object-oriented programming constructs:

$c ::=$	$skip$	termination
	$\mid chaos$	abort
	$\mid c \lhd b \rhd c$	conditional
	$\mid c \sqcap c$	non-determinism
	$\mid c; c$	sequence
	$\mid b * c$	iteration
	$\mid \mathbf{var}\ x : T$	declaration
	$\mid \mathbf{end}\ x$	undeclaration
	$\mid le := e$	assignmant
	$\mid o.m(e)$	method call

In our framework, each command denotes a design of the form

$$WF \Rightarrow D$$

where the predicate WF is true only when the command is well-defined in the initial state, while the design D is used to capture the dynamic behaviour of the command. We will demonstrate how all the programming operators can be defined in the exactly same way as their counterparts of the imperative languages. This makes the approach more comprehensible to users who are already familiar with the existing languages.

4.1 skip

The program *skip* does nothing, and terminates successfully. It is always well-defined.

$$skip =_{df} \emptyset : (true \vdash true)$$

4.2 chaos

chaos is the worst program, and its behaviour is unpredictable.

$$chaos =_{df} \emptyset : (false \vdash true)$$

4.3 Conditional

Let P and Q be designs. The notation

$$P \lhd b \rhd Q$$

describes a program which behaves like P if the initial value of b is true, or like Q if the initial value of b is false.

$$P \lhd b \rhd Q \quad =_{df} \quad (\mathcal{D}(b) \ \wedge \ type(b) = \mathcal{B}) \ \Rightarrow (P \wedge b \ \vee \ Q \ \wedge \neg b)$$

We use the conditional

$$b_\perp \ =_{df} \ (skip \lhd b \rhd choas)$$

to represent Floyd assertion [8], which fails if the initial value of b is false, otherwise it has no effect.

Let $\{P_i \mid 1 \leq i \leq n\}$ be a family of designs. The alternation

$$\textbf{if } \{(b_i \rightarrow P_i) \mid 1 \leq i \leq n\} \textbf{ fi}$$

selects P_i to execute if its guard b_i is true. When all the guards are false it behaves chaotically.

$$\textbf{if } \{(b_i \rightarrow \ P_i) \mid 1 \leq i \leq n\} \textbf{ fi } =_{df}$$

$$\bigwedge_{i:1..n} (\mathcal{D}(b_i) \wedge type(b_i) = \mathcal{B}) \;\Rightarrow\; (\bigvee_{i:1..n} (b_i \wedge P_i) \;\vee\; \neg(\bigvee_{i:1..n} b_i) \wedge chaos)$$

4.4 Non-determinism

Let P and Q be designs. The notation

$$P \sqcap Q$$

stands for a program which is executed by executing either P or Q, but with no indication which one will be chosen.

$$P \sqcap Q \;=_{df}\; P \vee Q$$

4.5 Composition

Let P and Q be designs. Their composition

$$P; Q$$

describes a program which may be executed by first executing P, and when P terminates then Q is started. The final state of P is passed on as the initial state of Q.

$$P(s,\, s'); Q(s,\, s') \;=_{df}\; \exists m \bullet P(s,\, m) \wedge Q(m,\, s')$$

4.6 Iteration

Let P be a design. If b is a condition, the notation

$$b * P$$

repeats the program P as long as b is true before each iteration.

$$b * P \;=_{df}\; \mu X \bullet (P; X) \lhd b \rhd skip$$

where $\mu X \bullet F(X)$ stands for the weakest fixed point [17] of the recursive equation $X = F(X)$.

4.7 Assignment

The assignment command is of the form $le := e$, where le lies in one of the following categories:

1. x, a program variable.
2. $le.a$, the a attribute of object le.

3. $(N)le$, the object le with type casting.

4.7.1 Simple assignment

The execution of $x := e$ assigns the value of expression e to variable x. It is well-defined if x and e are both well-defined, and the type of e matches that of x.

$(x := e) =_{df}$
$(\mathcal{D}(x) \wedge \mathcal{D}(e) \wedge (type(e) \leq type(x))) \Rightarrow \{x\} : (true \vdash (x' = e))$

4.7.2 Update of attribute

The execution of $o.a := e$ assigns the value of e to the a attribute of object o

$$o.a := e =_{df} o := (o; a : e)$$

Define

$$(N)o.a := e =_{df} o := ((N)o; a : e)$$

Notice that the assignments $o.a := e$ and $(N)o.a := e$ are defined by simple assignments, and their well-definedess is thus defined in terms of the well-definedess of their defining assignments $o := (o; a : e)$ and $o := ((N)o; a : e)$ respectively.

4.7.3 Assignment with type casting

The command $(N)x := e$ behaves the same as the simple assignment $x := e$ if N is a subtype of the declared type of x, but a supertype of the type of e.

$$(N)x := e =_{df} (type(e) \leq N \leq type(x)) \Rightarrow (x := e)$$

4.7.4 Simultaneous assignment

Let $\underline{x}$ be a list of variables, and $\underline{e}$ a list of expressions. The execution of $\underline{x} := \underline{e}$ assigns the values of the members of list $\underline{e}$ to the corresponding members of list $\underline{x}$ simultaneously.

$\underline{x} := \underline{e} =_{df}$
$(\mathcal{D}(\underline{x}) \wedge \mathcal{D}(\underline{e}) \wedge distinct(\underline{x}) \wedge type(\underline{e}) \leq type(\underline{x})) \Rightarrow \underline{x} : (true \vdash (\underline{x}' = \underline{e}))$

where the predicate $distinct(s)$ returns the value $true$ if none of members of s occurs twice or more.

$$distinct(s) \ =_{df} \ \forall i, j : 1..\#s \bullet (i \neq j) \Rightarrow (s[i] \not\equiv s[j])$$

where $x \not\equiv y$ indicates that x and y are not the same variable.

4.8 Variable declaration

To introduce a new program variable v we use the form of *declaration*

$$\textbf{var } v : T$$

which permits the variable v of type T to be used in the portion of the program that follows it. The complementary operation (called *undeclaration*) takes the form

$$\textbf{end } v$$

and terminates the region of permitted use of v. The portion of program P which the variable v may be used is called its *scope*. In this case, v is called a *local variable*. Formally, the declaration and undeclaration are defined by

$$\begin{aligned} \textbf{var } v : T \ =_{df} \ & v \notin (\textbf{alphabet} \cup \textbf{locvar}) \Rightarrow \\ & \textbf{locvar} : (true \vdash \textbf{locvar}' = \textbf{locvar} \cup \{v : T\}) \\ \textbf{end } v =_{df} \ & v \in \textbf{locvar} \Rightarrow \\ & \textbf{locvar} : (true \vdash \textbf{locvar}' = \{v\} \trianglelefteq \textbf{locvar}) \end{aligned}$$

where $\{v\} \trianglelefteq \textbf{locvar}$ represents the set **locvar** after removal of variable v. Note that the constraints on **locvar** forbid the redefinition of a variable within its own scope.

For convenience we allow variables to be declared together in a list provided that they are all distinct.

$$\begin{aligned} & \textbf{var } v_1 : T_1, \ .. \ , v_k : T_k \ =_{df} \\ & (distinct(< v_1, .., v_k >) \ \wedge \ \{v_1, ..., v_k\} \cap \pi_1(\textbf{alphabet} \cup \textbf{locvar}) = \emptyset) \Rightarrow \\ & \textbf{locvar} : (true \vdash \textbf{locvar}' = \textbf{locvar} \cup \{v_1 : T_1, .., v_k : T_k\}) \end{aligned}$$

4.9 Method call

Let *vale*, *rese* and *valrese* be lists of expressions. The program $o.m(vale, rese, valrese)$ assigns the values of the actual parameters *vale* and *valrese* to the formal value and value-result parameters of the method m of object o, and then executes the body of m. After it terminates, the values of the result and value-result parameters of m are passed back to the actual parameters *rese* and *valrese*.

$$\begin{aligned} & o.m(vale, \ rese, \ valrese) \ =_{df} \\ & (\mathcal{D}(o) \ \wedge \ type(o) \in \textbf{CName} \ \wedge \ m \in \textbf{method}(o(myclass))) \ \Rightarrow \end{aligned}$$

$$\textbf{if } \{(o(myclass) = N \rightarrow \left(\begin{array}{l} \textbf{var self} : N,\ x : T_1,\ y : T_2,\ z : T_3\,; \\ \textbf{self},\ x,\ z := o,\ vale,\ valrese\,; \\ N.m\,; \\ o,\ rese,\ valrese := \textbf{self},\ y,\ z\,; \\ \textbf{end self},\ x,\ y,\ z \end{array}\right)$$
$$\mid N \leq type(o) \wedge m \in \textbf{method}(N)\} \textbf{ fi}$$

where

- x, y and z are the value, result and value-result parameters of the method m of class $o(myclass)$, and T_1, T_2 and T_3 are their types and $N.m$ stands for the design associated with the method m of class N.
- **self**, a specific value-result parameter used to refer to the object which currently activates the method, is binded to the object o

.

5 Class Declarations

This section deals with class declarations and their well-definedness. We present a formal definition of individual class declaration in the first subsection. The second subsection discusses the composition of class declarations. We formalise the whole class declaration section as a single semantic entity in the final section.

5.1 Class declaration

A class declaration *cdecl* has the following form

$$\begin{array}{rl} \textbf{class } N & \textbf{extends } M \\ \textbf{pri} & u_1 : U_1 = c_1, ..., u_i : U_i = c_i \\ \textbf{pro} & v_1 : V_1 = d_1,\ ...,\ v_j : V_j = d_j \\ \textbf{pub} & w_1 : W_1 = e_1,\ ...,\ w_k : W_k = e_k \\ \textbf{meth} & m_1 =_{df} \textbf{ val } \underline{x}_{1,1} : \underline{T}_{1,1},\ \textbf{res } \underline{y}_{1,1} : \underline{T}_{1,2},\ \textbf{valres } \underline{z}_{1,3} : \underline{T}_{1,3} \bullet p_1 \\ & ..... \\ & m_l =_{df} \textbf{ val } \underline{x}_{l,1} : \underline{T}_{l,1},\ \textbf{res } \underline{y}_{l,2} : \underline{T}_{l,2},\ \textbf{valres } \underline{z}_{l,3} : \underline{T}_{l,3} \bullet p_l \\ \textbf{end} & \end{array}$$

where

- N is the name of this declared class, and M is its direct superclass.
- The section **pri** $u_1 : U_1 = c_1, ..., u_i : U_i = c_i$ declares the variables $u_1, ..., u_i$ as the private attributes of the class N, while $U_1, ..., U_i$ represent their types, and constants $c_1, ..., c_i$ their initial values respectively. This section is optional and not contained in the class declaration if N does not have private attributes.
- The section **pro** $v_1 : V_1 = d_1$, ..., $v_j : V_j = d_j$ defines the protected attributes of N. This section is also optional as not every class has to have protected attributes.
- The section **pub** $w_1 : W_1 = e_1$, ..., $w_k : V_k = d_k$ defines the public attributes of N. This section is again optional.
- m_1, ... , m_l are methods defined in N, where $\underline{x}_{i,1} : \underline{T}_{i,1}$, $\underline{y}_{i,2} : T_{i,2}$, $\underline{z}_{i,3} : T_{i,3}$ and p_i represent the value, result and value-result parameters and the body of the method m_i. This method section is also optional.

The class declaration *cdecl* is well-defined when it meets the following conditions:

1. N and M are distinct.
2. The attribute names are distinct
3. The initial values of the attributes lie in their corresponding types accordingly.
4. The method names are distinct.
5. The parameters of every method are distinct.

The well-definedeness of the method bodies can not be determined by *individual* class declaration itself, and it will be defined in the end of the declaration section of a program. Formally, the well-definedness of the above class declaration given for N is defined by the following predicate $\mathcal{WD}$.

$$
\begin{aligned}
&\mathcal{WD} =_{df} \\
&N \not\equiv M \ \wedge \ distinct(< u_1, .., u_i > \cdot < v_1, ..., v_j > \cdot < w_1, ..., w_k > \cdot < m_1, .., m_l >) \ \wedge \\
&\forall n : 1..i \bullet type(c_n) = U_n \ \wedge \ \forall n : 1..j \bullet type(d_n) = V_n \ \wedge \ \forall n : 1..k \bullet type(e_n) = W_n \ \wedge \\
&\forall n : 1..l \bullet \ (\ distinct(\underline{x}_{n,1} \cdot \underline{y}_{n,2} \cdot \underline{z}_{n,3}) \ \wedge \\
&\#\underline{x}_{n,1} = \#\underline{T}_{n,1} \ \wedge \ \#\underline{y}_{n,2} = \#\underline{T}_{n,2} \ \wedge \ \#\underline{z}_{n,3} = \#\underline{T}_{n,3})
\end{aligned}
$$

The class declaration *cdecl* adds the structural information of class N to the state of the follow-up program, and its role is captured by the design

$$
\begin{array}{l}
cdecl \ =_{df} \\
\qquad \mathcal{WD} \ \vdash \left(\begin{array}{l}
\mathbf{alphabet}' = \mathbf{alphabet} \ \wedge \\
\mathbf{CName}' = \{N\} \ \wedge \ \mathbf{superclass}' = \{N \mapsto M\} \wedge \\
\mathbf{pri}' =_{df} \ \{N \mapsto \{< u_1 : U_1,\, c_1 >, \ldots, < u_i : U_i,\, c_i >\} \ \wedge \\
\mathbf{prot}' = \{N \mapsto \{< v_1 : V_1,\, d_1 >, \ldots < v_j : V_j,\, d_j >\} \wedge \\
\mathbf{pub}' = \{N \mapsto \{< w_1 : W_1,\, e_1 >, \ldots < w_k : W_k,\, e_k >\} \wedge \\
\mathbf{method}' = \{N \mapsto \{(m_1 \mapsto (< \underline{x}_{1,1} : T_{1,1},\, \underline{y}_{1,2} : T_{1,2},\, \underline{z}_{1.3} : T_{1,3} >,\, p_1), \ldots \\
\qquad\qquad\qquad\qquad m_l \mapsto (< \underline{x}_{l,1} : T_{l,1},\, \underline{y}_{l.2} : T_{l,2},\, \underline{z}_{l,3} : T_{l,3} >,\, p_l)\})
\end{array} \right)
\end{array}
$$

where the logical variables **pri**, **prot** and **pub** are introduced to record the declared attributes of N, from which the value of **attribute** will be derived later after a complete superclass relation among the declared classes is defined. Similarly, the dynamic behaviour of the methods can not be formalised before the dependency relation among classes is specified. As a result, the logic variable **method**(N) binds each method m_i to its body p_i rather than its meaning, which will be calculated in the end of the declaration section.

5.2 Composing class declarations

Composing class declarations simply adds up the contents of the current environment α generated by the component class declarations provided that there is no redefinition of a class in its scope. It can be defined by the parallel merge [9]:

$$(cdecl1\,;\, cdecl2) \ =_{df} \ cdecl1 \parallel_M cdecl2$$

where $X(m,\, m') \|_M Y(m,\, m')$ is formulated by

$$(X(m,\, m'_1) \wedge Y(m,\, m'_2))\,;\, M(m_1,\, m_2,\, m')$$

and the design M, which merges the outputs m_1 and m_2 of X and Y into the output m of the parallel construct, is defined by

$$
(\mathbf{CName}_1 \cap \mathbf{CName}_2 = \emptyset) \ \vdash \left(\begin{array}{l}
\mathbf{alphabet}' = \mathbf{alphabet}_1 \cup \mathbf{alphabet}_2 \wedge \\
\mathbf{CName}' = \mathbf{CName}_1 \cup \mathbf{CName}_2 \wedge \\
\mathbf{superclass}' = \mathbf{superclass}_1 \cup \mathbf{superclass}_2 \wedge \\
\mathbf{pri}' = \mathbf{pri}_1 \cup \mathbf{pri}_2 \wedge \\
\mathbf{prot}' = \mathbf{prot}_1 \cup \mathbf{prot}_2 \wedge \\
\mathbf{pub}' = \mathbf{pub}_1 \cup \mathbf{pub}_2 \wedge \\
\mathbf{method}' = \mathbf{method}_1 \cup \mathbf{method}2
\end{array} \right)
$$

5.3 Well-definedness of a declaration section

A class declaration section is a composition of a sequnce of class declarations, and it is well-defined if the contents of α it has generated meet the following conditions:

1. All the class names used as the types of global variables, attributes and parameters are declared in the section:

   $$\mathcal{WD}1 \ =_{df} \ \bigwedge_{x:T\in S} T \in \mathbf{CName} \ \vee \ T \text{ is primitive}$$

   where S stands for the set of global variables, attributes of the declared classes and formal parameters of the declared methods:

   $$S \ =_{df} \ \mathbf{alphabet} \ \cup \ \bigcup_{N\in\mathbf{CName}} \pi_1(range(\mathbf{method}(N)) \ \cup \ \bigcup_{N\in\mathbf{CName}} \mathbf{attr}(N)$$

   and

   $$\mathbf{attr}(N) \ =_{df} \ \{x : T \mid \exists c \bullet < x : T, \, c > \in \mathbf{pri}(N) \cup \mathbf{prot}(N) \cup \mathbf{pub}(N)\}$$

2. The function **superclass** does NOT contain circularity.

   $$\mathcal{WD}2 \ =_{df} \ \mathbf{superclass}^+ \cap Id = \emptyset$$

   where we abuse the notation by treating **superclass** as a binary relation, and defining

   $$\begin{aligned} \mathbf{superclass}^+ &=_{df} \bigcup_{n\geq 1} \mathbf{superclass}^n \\ \mathbf{superclass}^1 &=_{df} \mathbf{superclass} \\ \mathbf{superclass}^{n+1} &=_{df} \mathbf{superclass}^n ; \mathbf{superclass} \end{aligned}$$

3. No attribute of a class can be redefined in its subclass.

   $$\begin{aligned} &\mathcal{WD}3 \ =_{df} \\ &\forall N, M : \mathbf{CName} \bullet (N, M) \in \mathbf{superclass}^+ \Rightarrow \mathbf{attrname}(N)) \cap \mathbf{attrname}(M) = \emptyset \end{aligned}$$

   where

   $$\mathbf{attrname}(N) \ =_{df} \ \{x \mid \exists T, \, c \bullet < x : T, \, c > \in \mathbf{pri}(N) \cup \mathbf{prot}(N) \cup \mathbf{pub}(N)\}$$

4. No method is allowed to change its signature given in its subclass.

   $$\begin{aligned} &\mathcal{WD}4 \ =_{df} \\ &\forall N, M : \mathbf{CName}, \forall m : dom(\mathbf{method}(N)) \cap dom(\mathbf{method}(M)\bullet \\ &(N, M) \in \mathbf{superclass}^+ \ \Rightarrow \ \pi_1(\mathbf{method}(N)(m)) = \pi_1(\mathbf{method}(M)(m)) \end{aligned}$$

Let *cdecls* be a class declaration section, and P be a command, the meaning of a program $(cdecls \bullet P)$ is defined as the composition of the meaning of class declarations *cdecls* (defined in Section 5.2) and the design *init* and the meaning of P (defined in Section 4):

$$cdecls \bullet P \ =_{df} \ (cdecls\,;\, init\,;\, P)$$

where the design $init$, that can be seen as the semantics of $\bullet$, performs the following tasks

1. to check the well-definedness of the declaration section,
2. to derive the values of the logical variables **attribute** and **visibleattr** from those of **pri**, **prot** and **pub**.
3. to define the dynamic behaviour of every method

$$
\begin{array}{l}
init \ =_{df} \\
(\mathcal{WD}1 \ \wedge \ \mathcal{WD}2 \ \wedge \ \mathcal{WD}3 \ \wedge \ \mathcal{WD}4) \ \vdash \\
\left(\begin{array}{l}
\mathbf{alphabet}' = \mathbf{alphabet} \ \wedge \ \mathbf{locvar}' = \emptyset \ \wedge \\
\mathbf{superclass}' = \mathbf{superclass} \ \wedge \ \mathbf{CName}' = \mathbf{CName} \wedge \\
\mathbf{visibleattr}' = \bigcup_{N \in \mathbf{CName}} \{N.a \mid \exists T,\, c \bullet < a : T,\, c > \in \mathbf{pub}(N)\} \wedge \\
\forall N : \mathbf{CName} \bullet \\
\mathbf{attribute}'(N) = \bigcup \{\mathbf{pri}(\mathbf{M}) \cup \mathbf{prot}(M) \cup \mathbf{pub}(M) \mid N \leq M\} \wedge \\
\mathbf{method}'(N) = \{(m \mapsto (< x : T_1,\, y : T_2,\, z : T_3 >, \mathbf{D}(N.m))) \mid \\
\qquad\qquad (\exists p \bullet \{m \mapsto (< x : T,\, y : T_2,\, z : T_3 >, p)\} \in \mathbf{method}(M) \wedge N \leq M\}
\end{array}\right)
\end{array}
$$

where the family of designs $\mathbf{D}$ is in sequel.

5.4 Formalisation of methods

The family of designs $\mathbf{D}$ is defined as the least fixed point of a set of recursive equations, which captures the dependency relation among the declared classes, and contains for each class $N \in \mathbf{CName}$ and every method $m \in \{\mathbf{method}(M) \mid N \leq M\}$ an equation

$$\mathbf{D}(N.m) \ = \ f(\mathbf{D})$$

that is described as follows.

Case (1): m is not defined in N, but in a superclass of N, i.e.

$$m \notin \mathbf{method}(N) \ \wedge \ m \in \bigcup \{\mathbf{method}(M) \mid N \leq M\}$$

The defining equation for $\mathbf{D}(N.m)$ is simply

$$\mathbf{D}(N.m) \ = \ \mathbf{D}(M.m) \qquad\qquad \text{where } \mathbf{superclass}(N) = M$$

Case (2): m is a method defined in class N. In this case, its dynamic behaviour can be captured

by its body program and the environment in which it is executed. The design $\mathbf{D}(N.m)$ is thus governed by the following defining equation

$$\mathbf{D}(N.m) \quad = \quad \phi_N(body(N.m))$$

where the function ϕ_N revises the body program of the method m by performing three tasks

1. To decide the set of attributes which are visible from inside the class N
2. To replace the reference to the attributes of class N in the body text by the reference to the attributes of object **self**.
3. To treat the method calls in the body text in a similar way as in Section 4.9 by passing the actual parameters to their corresponding formal parameters

The function ϕ_N distributes over programming operators, and can be inductively defined as follows:

$$\begin{aligned}
\phi_N(p_1; p_2) &=_{df} \phi_N(p_1); \phi_N(p_2) \\
\phi_N(P_1 \lhd b \rhd P_2) &=_{df} setvisibattr(N); (\phi_N(p_1) \lhd \phi_N(b) \rhd \phi_N(p_2)); resetvisibattr \\
\phi_N(b * p) &=_{df} setvisibattr(N); (\phi_N(b) * \phi_N(p)); resetvisibattr \\
\phi_N(l := e) &=_{df} setvisibattr(N); \phi_N(l) := \phi_N(e); resetvisibsttr
\end{aligned}$$

where the design $setvisibattr(N)$ generates the set of attributes which are accessible from inside the class N.

$$setvisinbattr(N) =_{df}$$

$$\{\mathbf{visibattr}\} : true \vdash \mathbf{visibattr}' = \left(\begin{array}{l} \{N.a \mid \exists T,\, c \bullet < a : T,\, c > \in \mathbf{pri}(N)\} \cup \\ \bigcup_{N \leq M} \{M.a \mid \exists T,\, c \bullet < a : T,\, c > \in \mathbf{prot}(M)\} \cup \\ \bigcup_{M \in \mathbf{CName}} \{M.a \mid \exists T,\, c \bullet < a : T,\, c > \in \mathbf{pub}(M)\} \end{array} \right)$$

The design $resetvisibattr$ delivers the set of attributes visible to the main program.

$$resetvisibattr =_{df}$$
$$\{\mathbf{visibattr}\} : true \vdash \mathbf{visibattr}' = \bigcup_{M \in \mathbf{CName}} \{M.a \mid \exists T,\, c \bullet < a : T,\, c > \in \mathbf{pub}(M)\}$$

The effect of ϕ_N on expression is defined as follows:

$$\phi_N(\mathbf{self}) \quad =_{df} \quad \mathbf{self}$$

$$
\begin{array}{rcll}
\phi_N(\mathbf{null}) & =_{df} & \mathbf{null} & \\
\phi_N(\mathbf{new}\, M) & =_{df} & \mathbf{new}\, M & \\
\phi_N(x) & =_{df} & \mathbf{self}.x & x \in \mathbf{attribute}(N) \\
 & & x & \text{otherwise} \\
\phi_N(f(e)) & =_{df} & f(\phi_N(e)) & \\
\phi_N(e\,\mathbf{is}\,M) & =_{df} & \phi_N(e)\,\mathbf{is}\,M & \\
\phi_N((M)e) & =_{df} & (M)\phi_N(e) & \\
\phi_N(e.x) & =_{df} & \phi_N(e).x & \\
\phi_N(e; x : f) & =_{df} & (\phi_N(e)\,;\, x : \phi_N(f)) &
\end{array}
$$

And a method call is mapped by ϕ_N to

$$
\begin{array}{l}
\phi_N(l.\hat{m}(val,\, res,\, valres)) \;=_{df} \\
\mathbf{if}\ \{\phi_N(l)(myclass) = M \wedge \hat{m} \in \mathbf{method}(M) \rightarrow c^M_{\hat{m}} \mid M \in \mathbf{CName}\}\ \mathbf{fi}
\end{array}
$$

where $c^N_{\hat{m}}$ has a similar form as a method call in Section 4.9, except that the formal parameters of $\hat{m}$ are renamed to resolve the name conflict.

$$
\begin{array}{l}
c^M_{\hat{m}} \;=_{df} \\
\mathbf{var}\ localself : M,\ localval : T1,\ localres : T2,\ localvalres : T3; \\
localself,\ localval,\ localvalres := \phi_N(l),\ \phi_N(val),\ \phi_N(valres); \\
\mathbf{D}(M.\hat{m})[localself,\ localval,\ localres,\ localvalres/\mathbf{self},\ \underline{x}_{M.\hat{m}},\ \underline{y}_{M.\hat{m}},\ \underline{z}_{M.\hat{m}}]; \\
\phi_N(l),\ \phi_N(res),\ \phi_N(valres) := localself,\ localres,\ localvalres; \\
\mathbf{end}\ localself,\ localval,\ localres,\ localvalres
\end{array}
$$

where $\underline{x}_{M.\hat{m}}$, $\underline{y}_{M.\hat{m}}$ and $\underline{z}_{M.\hat{m}}$ are the value parameters, the result parameters and the value-result parameters of the method $\hat{m}$ of class M, and $T1$, $T2$ and $T3$ are their types.

6 Conclusion

This paper presents a relational semantics to an object-oriented programming language. Our treatment integrates the dynamic type and visibility check mechanism with the traditional specification-oriented semantics. We have treated the programming operators of our object-oriented language in the exactly same way as their counterparts in the imperative languages. Consequently, most of the algebraic laws developed for the imperative languages remain applicable in designing object-oriented programs. For example, conditional, non-determinism and sequence of our language are subject to the same basic laws designed for the imperative lan-

guages as shown below.

Conditional is idempotent, skew symmetric and associative.

cond-1 $P \lhd b \rhd P \;=\; P$ provided that the Boolean expression b is always well-defined.

cond-2 $P \lhd b \rhd Q \;=\; Q \lhd \neg b \rhd P$

cond-3 $(P \lhd b \rhd Q) \lhd c \rhd R \;=\; P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$

Any conditional choice operator $\lhd b \rhd$ distributes through the conditional $\lhd c \rhd$, for any condition c.

cond-4 $P \lhd b \rhd (Q \lhd c \rhd R) \;=\; (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$

The following law expresses the criterion for making a choice between two alternatives of a conditional based on the value of the condition.

cond-5 $P \lhd true \rhd Q \;=\; P$

The non-deterministic choice satisfies a number of laws common to other forms of choice, including the conditional

choice-1 $P \sqcap Q \;=\; Q \sqcap P$

choice-2 $P \sqcap (Q \sqcap R) \;=\; (P \sqcap Q) \sqcap R$

choice-3 $P \sqcap P \;=\; P$

Both the conditional and sequence combinators distribute through $\sqcap$.

choice-4 $P \lhd b \rhd (Q \sqcap R) \;=\; (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$

choice-5 $(P \sqcap Q); R \;=\; (P; R) \sqcap (Q; R)$

choice-6 $P; (Q \sqcap R) \;=\; (P; Q) \sqcap (P; R)$

$\sqcap$ distributes through the conditional, and has *chaos* as its zero.

choice-7 $P \sqcap (Q \lhd b \rhd R) \;=\; (P \sqcap Q) \lhd b \rhd (P \sqcap R)$

choice-8 $P \sqcap chaos \;=\; chaos$

Sequential composition obeys simple and obvious laws. It is associative and distributes leftward through the conditional.

seq-1 $(P; Q); R \;=\; P; (Q; R)$

seq-2 $(P \lhd b \rhd Q); R \;=\; (P; R) \lhd b \rhd (Q; R)$

It has **skip** as the unit, and *chaos* as the left zero.

seq-3 $(skip; P) \;=\; P \;=\; (P; skip)$

seq-4 $(chaos; P) \;= chaos$

Assignment is subject to the following laws

assign-1 $(x,\ y,\ z := e,\ f,\ g)\ =\ (y,\ x,\ z := f,\ e,\ g)$

assign-2 $(x := e\,;\ x := f(x)\ =\ (x := f(e))$

assign-3 $x := e; (P \lhd b \rhd Q)\ =\ (x := e; P) \lhd b(e) \rhd (x := e; Q)$

assign-4 $(y \in (\mathbf{alphabet} \cup \mathbf{locvar}))_\perp; (x := e)\ =\ (x,\ y := e,\ y)$

Both declaration and undeclaration are commutative.

vardecl-1 $(\mathbf{var}\, x1 : T1\,;\ \mathbf{var}\, x2 : T2)\ =\ (\mathbf{var}\, x2 : T2\,;\ \mathbf{var}\, x1 : T1)\ =\ \mathbf{var}\, x1 : T1,\ x2 : T2$

vardecl-2 $(\mathbf{end}\, x1\,;\ \mathbf{end}\, x2)\ =\ (\mathbf{end}\, x2\,;\ \mathbf{end}\, x1)\ =\ \mathbf{end}\, x1,\ x2$

vardecl-3 $(\mathbf{var}\, x : T\,;\ \mathbf{end}\, y)\ =\ (\mathbf{end}\, y\,;\ \mathbf{var}\, x : T)$ provided x and y are distinct.

The redefinition of a variable in its scope leads to chaos.

vardecl-4 $\mathbf{var}\, x : T1\,;\ \mathbf{var}\, x : T2\ =\ \mathit{chaos}$

vardecl-5 $\mathbf{end}\, x\,;\ \mathbf{end}\, x\ =\ \mathit{chaos}$

Declaration and undeclaration distribute through a conditional as long as no interference occurs with the condition.

vardecl-6 If x is not free in b, then

$$\mathbf{var}\, x : T; (P \lhd b \rhd Q)\ =\ (\mathbf{var}\, x : T\,;\ P) \lhd b \rhd (\mathbf{var}\, x : T\,;\ Q)$$
$$\mathbf{var}\, x : T; (P \lhd b \rhd Q)\ =\ (\mathbf{var}\, x : T\,;\ P) \lhd b \rhd (\mathbf{var}\, x : T\,;\ Q)$$

$\mathbf{var}\, x : T$ followed by $\mathbf{end}\, x$ has no effect provided that x is not known.

vardecl-7 $(\mathbf{var}\, x : T\,;\ \mathbf{end}\, x)\ =\ (x \notin (\mathbf{alphabet} \cup \mathbf{locvar}))_\perp$

The next law describes a similar property of $\mathbf{end}\, x; \mathbf{var}\, x : T$.

vardecl-8 If x is not free in e, then

$$((x : T \in \mathbf{locvar})_\perp; \mathbf{end}\, x\,;\ \mathbf{var}\, x : T\,;\ x := e)\ =\ ((x : T \in \mathbf{locvar})_\perp; x := e)$$

Nevertheless, our model does not support some useful algebraic laws for class declarations. For example, let

$$\mathit{decls}1\ =\ \left(\begin{array}{ll} \mathbf{class}\ N & \\ \quad \mathbf{pri} & a : U_1 = c1 \\ \quad \mathbf{meth} & m =_{df} \mathbf{val}\ \underline{x} : \underline{T}_1,\ \mathbf{res}\ \underline{y} : \underline{T}_2,\ \mathbf{valres}\ \underline{z} : \underline{T}_3\ \bullet\ p(a) \\ \quad \mathbf{end} & \end{array}\right)$$

and

$$decls2 \;=\; \left(\begin{array}{rl} \textbf{class}\ N & \\ \quad \textbf{pri} & a : U_1 = c1,\ b : U_2 = c_2 \\ \quad \textbf{meth} & m =_{df} \textbf{val}\ \underline{x} : \underline{T}_1,\ \textbf{res}\ \underline{y} : \underline{T}_2,\ \textbf{valres}\ \underline{z} : \underline{T}_3 \bullet p(a) \\ \quad \textbf{end} & \end{array}\right)$$

Clearly, the state space of $cdcel1$ differs from that of $cdecls2$ since the latter has added a new private attribute b. However, for any command P, $cdecls1 \bullet P$ and $cdecls \bullet P$ behave the same from the external viewpoint. To model the interactions between classes and main programs we will enrich our state-based formalism by identifying each class as a communicating sequential process. In addition, we will provide a proper equivalence relation for class declarations, and explore a comprehensive set of algebraic laws in support of normal form reduction for our language.

References

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

Springer-Verlag, (1996).

[2] M. Abadi and K.R.M. Leino. *A logic of object-oriented programs.*

Lecture Notes in Computer Science 1214, 682–696, Springer, (1997)

[3] P. America. *Designing an object-oriented programming language with behavioural subtyping.*

Lecture Notes in Computer Science 489, 60–90, (1991).

[4] P. America and F. de Boer. *Reasoning about dynamically evolving process structures.* Formal Aspect of Computing, 6:269–316, (1994).

[5] R.J.R. Back and M.J. Butler. *Exploring summation and product operators in the refinement calculus.*

Lecture Notes in Computer Science 947, (1995).

[6] A. Cavalcanti and D. Naumann. *A weakest Precondition Semantics for an Object-Oriented Language of Refinement.*

Lecture Notes in Computer Science 1708, 1439–1460, (1998)

[7] J. Dong, R. Duke and G. Rose. *An object-oriented approach to the semantics of programming languages.*

In G. Gupta (ed): 17th Annual Computer Science Conference (ACSC'17), 765–775, (1994).

[8] R.W. Floyd. *Assigning meanings to programs.*

Proceedings of Symposia in Applied Mathematics, Vol 19: 19–32, (1967)

[9] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming.*

Prentice Hall, (1998)

[10] C.A.R. Hoare and He Jifeng. *A trace model for pointers and objects.*
Lecture Notes of International Summer School at Marktoberdoff. Germany, (1998)

[11] K. Lano and H. Haughton. *Reasoning and refinement in object-oriented specification language.*
Lecture Notes in Computer Science 615, Springer-Verlag, (1992).

[12] M. Marcello et al. *An approach to object-orientation inaction systems.*
Lecture Notes in Computer Science 1422, 68-95, (1998)

[13] A. Mikhajlova and E. Sekerinski. *Class refinement and interface refinement in object-oriented programming.* Proceedings of FME'97, Springer, (1997).

[14] D.A. Naumann. *Predicate transformer semantics of an Oberon-like language.* In E.-R. Olderog (ed) Programming Concepts, Methods and Calculi, 460–480, (1994).

[15] K. Rustan and M. Leino. *Recursive object types in a logic of object-oriented programming.* Lecture Notes in Computer Science 1381, (1998).

[16] E. Sekerinski. *A type-theoretical basis for an object-oriented refinement calculus.*
Proceedings of Formal Methods and Object Technology, Springer-Verlag, (1996)

[17] A. Tarski. *A lattice-theoretical fixpoint theorem and its applications.*
Pacific Journal of Mathematics, Vol 5: 285–309, (1955).