A WP-calculus for OO

F.S de Boer[1]

Utrecht University, The Netherlands
Email: frankb@cs.uu.nl

Abstract. A sound and complete Hoare-style proof system is presented for a sequential object-oriented language, called SPOOL. The proof system is based on a weakest precondition calculus for aliasing and object-creation.

1 Introduction

This paper introduces a Hoare-style proof system for an object-oriented language, called SPOOL. SPOOL is a sequential version of the parallel object-oriented language POOL [2].

The main aspect of SPOOL that is dealt with is the problem of how to reason about *pointer structures*. In SPOOL, objects can be created at arbitrary points in a program, references to them can be stored in variables and passed around as parameters in messages. This implies that complicated and dynamically evolving structures of references between objects can occur. We want to reason about these structures on an abstraction level that is *at least as high as that of the programming language*. In more detail, this means the following: The only operations on "pointers" (references to objects) are testing for equality and dereferencing (looking at the value of an instance variable of the referenced object). Furthermore, in a given state of the system, it is only possible to mention the objects that exist in that state. Objects that do not (yet) exist never play a role.

Strictly speaking, direct dereferencing is not even allowed in the programming language, because each object only has access to its own instance variables. However, for the time being we allow it in the assertion language. Otherwise, even more advanced techniques would be necessary to reason about the correctness of a program.

The above restrictions have quite severe consequences for the proof system. The limited set of operations on pointers implies that first-order logic is too weak to express some interesting properties of pointer structures (for example, the property, as considered in [9], that it is possible to go from one object to the other by following a finite number of x-links). Therefore we have to extend our assertion language to make it more expressive. In this paper we do so by allowing the assertion language to reason about finite sequences of objects.

The proof system itself is based on a *weakest precondition* calculus for aliasing and object-creation. This means that in the proof system aliasing and object-creation are modelled by substitutions which, when applied to a given postcondition, yield the corresponding weakest precondition.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 135-149, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Plan of the paper In the following section we introduce the programming language SPOOL. In section 3 the assertion language for describing object structures is introduced. The proof system is discussed in section 4. In the final section related work is discussed and some general conclusions are drawn.

2 The language SPOOL

The most important concept of SPOOL is the concept of an *object*. This is an entity containing data and procedures (*methods*) acting on these data. The data are stored in *variables*, which come in two kinds: *instance variables*, whose lifetime is the same as that of the object they belong to, and *temporary variables*, which are local to a method and last as long as the method is active. Variables can contain references to other objects in the system (or even the object under consideration itself). The object a variable refers to (its *value*) can be changed by an *assignment*. The value of a variable can also be nil, which means that it refers to no object at all.

The variables of an object cannot be accessed directly by other objects. The only way for objects to interact is by sending *messages* to each other. If an object sends a message, it specifies the receiver, a method name, and possibly some parameter objects. Then control is transferred from the sender object to the receiver. This receiver then executes the specified method, using the parameters in the message. Note that this method can, of course, access the instance variables of the receiver. The method returns a result, an object, which is sent back to the sender. Then control is transferred back to the sender which resumes its activities, possibly using this result object.

The sender of a message is *blocked* until the result comes back, that is, it cannot answer any message while it still has an outstanding message of its own. Therefore, when an object sends a message to itself (directly or indirectly) this will lead to abnormal termination of the program.

Objects are grouped into *classes*. Objects in one class (the *instances* of the class) share the same methods, so in a certain sense they share the same behaviour. New instances of a given class can be created at any time. There are two standard classes, Int and Bool, of integers and booleans, respectively. They differ from the other classes in that their instances already exist at the beginning of the execution of the program and no new ones can be created. Moreover, some standard operations on these classes are defined.

A program essentially consists of a number of class definitions, together with a statement to be executed by an instance of a specific class. Usually, but not necessarily, this instance is the only non-standard object that exists at the beginning of the program: the others still have to be created.

In order to describe the language SPOOL, which is strongly typed, we use *typed* versions of all variables, expressions, etc. These types however are implicitly assumed in the language description below.

We assume the following sets to be given: A set C of *class names*, with typical element c (this means that metavariables like $c, c', c_1, \ldots$ range over elements of

the set C). We assume that $\mathsf{Int}, \mathsf{Bool} \notin C$ and define the set $C^+ = C \cup \{\mathsf{Int}, \mathsf{Bool}\}$, with typical element d. For each $c \in C$, $d \in C^+$ we assume a set $IVar^c_d$, with typical element x, of *instance variables* in class c which are of type d. For each $d \in C$ we assume a set $TVar_d$ of *temporary variables* of type d, with typical element u. Finally, for each $c \in C$ and $d_0, \ldots, d_n \in C^+$ ($n \geq 0$) we assume a set $MName^c_{d_0,\ldots,d_n}$ of *method names* of class c with result type d_0 and parameter types $d_1, \ldots, d_n$. The set $MName^c_{d_0,\ldots,d_n}$ will have m as a typical element.

Now we can specify the syntax of our (strongly typed) language (we omit the typing information).

Definition 1. For any $c \in C$ and $d \in C^+$ the set Exp^c_d of *expressions* of type d in class c, with typical element e, is defined as usual. We give the following base cases.

$$e ::= x \mid u \mid \mathsf{nil} \mid \mathsf{self} \cdots$$

The set $SExp^c_d$ of expressions with possible *side effect* of type d in class c, with typical element s, is defined as follows:

$$s ::= e \mid \mathsf{new} \mid e_0!m(e_1, \ldots, e_n)$$

The first kind of side effect expression is a normal expression, which has no actual side effect, of course. The second kind is the creation of a new object. This new object will also be the value of the side effect expression. The third kind of side effect expression specifies that a message is to be sent to the object that results from e_0, with method name m and with arguments (the objects resulting from) $e_1, \ldots, e_n$.

The set $Stat^c$ of *statements* in class c, with typical element S, are constructed from assignments by means of the standard sequential operations of sequential composition, (deterministic) choice and iteration.

Definition 2. The set $MethDef^c$ of *method definitions* in class c, with typical element μ, is defined by:

$$\mu ::= (u_1, \ldots, u_n : S \uparrow e)$$

Here we require that the u_i are all different and that none of them occurs at the left hand side of an assignment in $S \in Stat^c$ (and that $n \geq 0$).

When an object is sent a message, the method named in the message is invoked as follows: The variables $u_1, \ldots, u_n$ (the parameters of the method) are given the values specified in the message, all other temporary variables (i.e. the *local* variables of the method, are initialized to nil, and then the statement S is executed. After that the expression e is evaluated and its value, the result of the method, is sent back to the sender of the message, where it will be the value of the send-expression that sent the message.

Definition 3. The set $ClassDef^c$ of definitions of class c, with typical element D, is defined by:

$$D ::= c : \langle m_1 = \mu_1, \ldots,\ m_n = \mu_n \rangle$$

where we require that all the method names are different (and $n \geq 0$).

Definition 4. Finally, the set $Prog^c$ of *programs* in class c, with typical element ρ, is defined by:

$$\rho ::= \langle U | c : S \rangle$$

where U denotes a finite set of class definitions and $S \in Stat^c$. The interpretation of such a program is that the statement S is executed by some object of class c (the root object) in the context of the declarations contained in the unit U. In many cases (including the following example) we shall assume that at the beginning of the execution this root object is the only existing non-standard object.

Example 1. The following program generates prime numbers using the sieve method of Eratosthenes.

```
⟨Sieve : ⟨input ⇐ (q) : if next = nil
                        then next := new;
                             p := q
                        else  if q mod p ≠ 0
                             then next ! input(q)
                             fi
                        fi
                        ↑ self ⟩,
Driver : ⟨ ⟩
|
Driver : i := 2;
         first := new;
         while i < bound
         do first ! input(i);
            i := i + 1
         od
⟩
```

Figure 1 represents the system in a certain stage of the execution of the program.

3 The assertion language

In this section a formalism is introduced for expressing certain properties of a complete system, or configuration, of objects. Such a system consists for each class of a set of *existing* objects in that class (i.e. the objects in that class which have been created sofar) together with their internal states (i.e. an assignment of values to their own instance variables), and the currently active object together with an assignment of values to its temporary variables.

One element of this assertion language will be the introduction of *logical variables*. These variables may not occur in the program, but only in the assertion

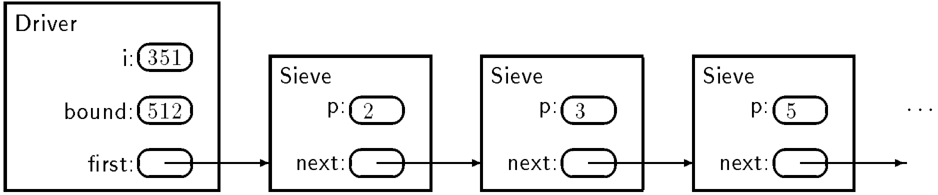

Fig. 1. Objects in the sieve program in a certain stage of the execution

language. Therefore we are always sure that the value of a logical variable can never be changed by a statement. Apart from a certain degree of cleanliness, this has the additional advantage that we can use logical variables to express the constancy of certain expressions (for example in the proof rule for message passing). Logical variables also serve as bound variables for quantifiers.

The set of expressions in the assertion language is larger than the set of programming language expressions not only because it contains logical variables, but also because by means of a dereferencing operator it is allowed to refer to instance variables of other objects. Furthermore we include conditional expressions in the assertion language. These conditional expressions will be used for the analysis of the phenomenon of aliasing which arises because of the presence of a dereferencing operator.

In two respects our assertion language differs from the usual first-order predicate logic: Firstly, the range of quantifiers is limited to the *existing* objects in the current state of the system. For the classes different from Int and Bool this restriction means that we cannot talk about objects that have not yet been created, even if they could be created in the future. This is done in order to satisfy the requirements on the proof system stated in the introduction. Because of this the range of the quantifiers can be different for different states. More in particular, a statement can change the truth of an assertion even if none of the program variables accessed by the statement occurs in the assertion, simply by creating an object and thereby changing the range of a quantifier. (The idea of restricting the range of quantifiers was inspired by [11].)

Secondly, in order to strengthen the expressiveness of the logic, it is augmented with quantification over finite sequences of objects. It is quite clear that this is necessary, because simple first-order logic is not able to express certain interesting properties.

Definition 5. For each $d \in C^+$ we introduce the symbol d^* for the type of all finite sequences with elements from d, we let C^* stand for the set $\{d^* | d \in C^+\}$, and we use $C^\dagger$, with typical element a, for the union $C^+ \cup C^*$. We assume that for every a in $C^\dagger$ we have a set $LVar_a$ of logical variables of type a, with typical element z.

Definition 6. We give the following typical elements of the set $LExp_a^c$ of logical expressions of type a in class c (we omit the typing information):

$$l ::= e \mid z \mid l.x \mid \mathsf{if}\ l_0\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2\ \mathsf{fi}$$

An expression e is evaluated in the internal state of the currently active object which is denoted by self. The difference with the set Exp_d^c of expressions in the programming language is that in logical expressions we can use logical variables, refer to the instance variables of other objects (the expression $l.x$ refers to the local value of the instance variable x of the object denoted by l), and write conditional expressions. Furthermore, we extended the domain of discourse by means of logical variables ranging over sequences. In order to reason about sequences we assume the presence of notations to express, for example, the length of a sequence (denoted by $|l|$) and the selection of an element of a sequence (denoted by $l(n)$, where n is an integer expression).

Definition 7. The set Ass^c of assertions in class c, with typical elements P and Q, is defined by:

$$P ::= l \mid P \wedge Q \mid \neg P \mid \exists z P$$

Here l denotes a boolean expression (i.e. $l \in Exp_{\mathsf{Bool}}^c$).

As already explained above, a formula $\exists z P$, with z of some type $c \in C$ states that P holds for some *existing* object in class c. A formula $\exists z P$, with z of a sequence type c^*, states the existence of a sequence of existing objects in class c.

Example 2. The formula $\exists z\ \mathsf{true}$, where z is of some type $c \in C$, thus states the existence of an object in class c. As such this formula is false in case no such objects exist. As another example, the following formula states the existence of a sequence of objects in class Sieve (of the example program in the previous section) such that the value of p of the nth element in this sequence is the nth prime number and next refers to the next element, i.e. the $n+1$th element, in the sequence.

$$\exists z \forall n \begin{pmatrix} (0 < n \wedge n \leq |z| \rightarrow z(n).\mathsf{p} = prime(n)) \\ \wedge \\ (0 < n \wedge n < |z| \rightarrow z(n).\mathsf{next} = z(n+1)) \end{pmatrix}$$

Here n denotes a logical variable ranging over integers and z ranges over sequences of objects in class Sieve. The predicate $prime(n)$ holds if n is a prime.

Definition 8. A *correctness formula* in class c is a Hoare triple of the form $\{P\}\rho\{Q\}$, where $P, Q \in Ass^c$ and $\rho \in Prog^c$.

A Hoare-triple $\{P\}\rho\{Q\}$ expresses a *partial correctness* property of the program ρ: It holds if every *successfully terminating* execution of the program ρ in a system of objects which satisfies the precondition P results in a final configuration which satisfies the postcondition Q.

4 The proof system

In this section we present a Hoare-style proof system which provides a view of programs in SPOOL as *predicate-transformers.*

Simple assignments We shall call a statement a *simple assignment* if it is of the form $x := e$ or $u := e$ (that is, it uses the first form of a side effect expression: the one without a side effect). For the axiomatization of simple assignments to temporary variables the standard assignment axiom suffices because objects are only allowed to refer to the instance variables of other objects and therefore *aliasing*, i.e. the situation that different expressions refer to the same variable, does not arise in case of temporary variables.

In the case that the target variable of an assignment statement is an instance variable, we use the following axiom:

$$\Big\{P[e/x]\Big\}\ \langle U|c : x := e\rangle\ \Big\{P\Big\}$$

The substitution operation $[e/x]$ has to account for possible aliases of the variables x, namely, expressions of the form $l.x$: It is possible that, after substitution, l refers to the currently active object (i.e. the object denoted by self), so that $l.x$ is the same variable as x and should be substituted by e. It is also possible that, after substitution, l does not refer to the currently executing object, and in this case no substitution should take place. Since we cannot decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides "dynamically".

Definition 9. We have the following main cases of the substitution operation $[e/x]$:

$$l\,.\,x\ [e/x] = \mathsf{if}\ (l[e/x]) = \mathsf{self\ then}\ e\ \mathsf{else}\ (l[e/x])\,.\,x\ \mathsf{fi}$$
$$l\,.\,x'\,[e/x] = (l[e/x])\,.\,x' \qquad \text{if } x' \neq x$$

The definition is extended to assertions other than logical expressions in the standard way.

Object creation Next we consider the creation of objects. We will introduce two different axiomatizations of object-creation which are based on the logical formulation of the *weakest precondition* and the *strongest postcondition*, respectively. First we consider a weakest precondition axiomatization.

For an assignment of the form $u := \mathsf{new}$ we have a axiom similar to the previous two:

$$\Big\{P[\mathsf{new}/u]\Big\}\ \langle U|c : u := \mathsf{new}\rangle\ \Big\{P\Big\}$$

We have to define the substitution $[\mathsf{new}/u]$. As with the notions of substitution used in the axioms for simple assignments, we want the expression after substitution to have the same meaning in a state before the assignment as the unsubstituted expression has in the state after the assignment. However, in the case of a new-assignment, there are expressions for which this is not possible,

because they refer to the new object (in the new state) and there is no expression that could refer to that object in the old state, because it does not exist yet. Therefore the result of the substitution must be left undefined in some cases.

However we *are* able to carry out the substitution in case of assertions, assuming, without loss of expressiveness, that in the assertion language the operations on sequences are limited to $|l|$, i.e. the length of the sequence l, and $l(n)$, i.e. the operation which yields the nth element of l. The idea behind this is that in an assertion the variable u referring to the new object can essentially occur only in a context where either one of its instance variables is referenced, or it is compared for equality with another expression. In both of these cases we can predict the outcome without having to refer to the new object.

Definition 10. Here are the main cases of the formal definition of the substitution $[\mathsf{new}/u]$ for logical expressions. As already explained above the result of the substitution $[\mathsf{new}/u]$ is undefined for the expression u. Since the (instance) variables of a newly created object are initialized to nil we have

$$u.x[\mathsf{new}/u] \;=\; \mathsf{nil}$$

If neither l_1 nor l_2 is u or a conditional expression they cannot refer to the newly created object and we have

$$\left(l_1 = l_2\right)[\mathsf{new}/u] = \left(l_1[\mathsf{new}/u]\right) = \left(l_2[\mathsf{new}/u]\right)$$

If either l_1 is u and l_2 is neither u nor a conditional expression (or vice versa) we have that after the substitution operation l_1 and l_2 cannot denote the same object (because one of them refers to the newly created object while the other one refers to an already existing object):

$$\left(l_1 = l_2\right)[\mathsf{new}/u] = \mathsf{false}$$

On the other hand if both the expressions l_1 and l_2 equal u we obviously have

$$\left(l_1 = l_2\right)[\mathsf{new}/u] = \mathsf{true}$$

We have that $l[\mathsf{new}/u]$ is defined for boolean expressions l.

Definition 11. We extend the substitution operation $[\mathsf{new}/u]$ to assertions other than logical expressions as follows (we assume that the type of u is $d \in C$):

$$\begin{array}{ll}
(P \rightarrow Q)[\mathsf{new}/u] & = (P[\mathsf{new}/u]) \rightarrow (Q[\mathsf{new}/u]) \\
(\neg P) \quad [\mathsf{new}/u] & = \neg(P[\mathsf{new}/u]) \\
(\exists z\ P) \quad [\mathsf{new}/u] & = (\exists z(P[\mathsf{new}/u])) \vee (P[u/z][\mathsf{new}/u]) \\
(\exists z\ P) \quad [\mathsf{new}/u] & = \exists z\ \exists z'\ (|z| = |z'| \wedge (P[z', u/z][\mathsf{new}/u])) \\
(\exists z\ P) \quad [\mathsf{new}/u] & = (\exists z(P[\mathsf{new}/u]))
\end{array}$$

In the third and fourth clause the (bound) variable z is assumed to be of type d and d^*, respectively. The type of the variable z in the last clause is of a type

different from d and d^*. The (bound) variable z' in the fourth clause is assumed to be of type boolean (this variable is also assumed not to occur in P).

The idea of the application of $[\mathsf{new}/u]$ to $(\exists z\ P)$ (in case z is of the same type as u) is that the first disjunct $(\exists z(P[\mathsf{new}/u]))$ represents the case that the object for which P holds is an 'old' object (i.e. which exists already before the creation of the new object) whereas the second disjunct $P[u/z][\mathsf{new}/u]$ represents the case that the new object itself satisfies P.

The idea of the fourth clause is that z and z' together code a sequence of objects in the state after the new-statement. At the places where z' yields true the value of the coded sequence is the newly created object. Where z' yields false the value of the coded sequence is the same as the value of z. This encoding is described by the substitution operation $[z', u/z]$ the main characteristic cases of which are:

$$
\begin{array}{ll}
z[z', u/z] & \text{is undefined} \\
\multicolumn{2}{l}{\Big(z(l)\Big)[z', u/z] = \mathsf{if}\ z'(l')\ \mathsf{then}\ u\ \mathsf{else}\ z(l')\ \mathsf{fi},\ \text{where } l' = l[z', u/z]}
\end{array}
$$

This substitution operation $[z', u/z]$ is defined for boolean expressions.

Example 3. Let z be a logical variable of the same type as u. We have

$$
\begin{array}{l}
\Big(\exists z(u = z)\Big)[\mathsf{new}/u] \qquad\qquad\qquad\quad \equiv \\
\Big(\exists z(u = z)[\mathsf{new}/u]\Big) \vee (u = u)[\mathsf{new}/u] \equiv \\
\exists z\ \mathsf{false} \vee \mathsf{true}
\end{array}
$$

where the last assertion obviously reduces to true, which indeed is the weakest precondition of $\exists z(u = z)$ with respect to $u := \mathsf{new}$.

Note that we cannot apply the substitution operation $[\mathsf{new}/u]$ directly to assertions involving more high-level operations on sequences. For example, an assertion like $l_1 \leq l_2$, which expresses that the sequence l_1 is a prefix of l_2, we have first to reformulate into a logically equivalent one which uses only the sequence operations $|l|$ and $l(n)$. Thus, $l_1 \leq l_2$ should be first translated into

$$\forall n(0 < n \wedge n \leq |l_1| \rightarrow l_1(n) = l_2(n))$$

If our assignment is of the form $x := \mathsf{new}$ we have the following axiom:

$$\Big\{P[\mathsf{new}/x]\Big\}\ \langle U|c : x := \mathsf{new}\rangle\ \Big\{P^c\Big\}$$

The substitution operation $[\mathsf{new}/x]$ is defined by: $P[u/x]\ [\mathsf{new}/u]$, where u is a temporary variable that does not occur in P. (It is easy to see that this definition does not depend on the actual u used.)

Thus we see that we are able to compute the weakest precondition of a new-statement despite the fact that we cannot refer to the newly created object in the state prior to its creation. Alternatively, we have the following strongest postcondition axiomatization of object-creation. Let u be a temporary variable

of type c, and the logical variables z and z' be of type c^* and c, respectively. Moreover, let V be a finite set of instance variables in class c. For an assignment of the form $u := \mathsf{new}$ we have the following axiom.

$$\Big\{P\Big\}\ \langle U|c : u := \mathsf{new}\rangle\ \Big\{\exists z\Big(P' \downarrow z \wedge Q(V, z)\Big)\Big\}$$

where $P' = \exists z'(P[z'/u])$ and $Q(V, z)$ denotes the following assertion

$$u \notin z \wedge \forall z'(z' \in z \vee z' = u) \wedge \bigwedge_{x \in V} u.x = \mathsf{nil}$$

The operation $\downarrow z$ applied to an assertion R restricts all quantifications in R to z. It is described in more detail below. Let us first explain the role of the logical variables z and z' (which are assumed not to occur in P). The logical variable z in the postcondition is intended to the store all the objects in class c which exist in the state prior to the creation of the new object. The logical variable z' is intended to represent the old value of u. Given that z' denotes the old value of u, that P holds for the old value of u then can be expressed in the postcondition simply by $P[z'/u]$. However the quantification $\exists z'(P[z'/u])$ in the postcondition will also include the newly created object. In general we thus have to take into account the changing scope of the quantifiers. For example, consider $P = \forall z''.\mathsf{false}$ (with z'' of type c). Obviously P, which states that there do not exist objects in class c, does not hold anymore after the creation of a new object in class c. Our solution to this problem is to restrict the scope of all quantifications involving objects in class c to the old objects in class c, which are given by z. This restriction operation is denoted by $R \downarrow z$. Its main characteristic defining clauses are the following two:

$$(\exists z'' R) \downarrow z = \exists z''(z'' \in z \wedge R \downarrow z)$$
$$(\exists z'' R) \downarrow z = \exists z''(z'' \subseteq z \wedge R \downarrow z)$$

where in the first clause z'' is of type c while in the second clause z'' is of type c^* (for convenience we assume the presence of the relation 'is an element of the sequence', denoted by $\in$, and the containment-relation $\subseteq$, which holds whenever all the elements of its first argument occur in its second argument). Finally, the assertion $Q(V, z)$ in the postcondition of axiom above expresses that u denotes the newly created object and specifies the initial values of the variables in V (of the newly created object).

For new-statements involving instance variables we have a similar axiom characterizing its strongest postcondition semantics.

It is of interest to observe here that the strongest postcondition axiomatization does not require a restricted repetoire of primitive sequence operations.

Method calls Next we present proof rules for verifying the third kind of assignments: the ones where a message is sent and the result stored in the variable on the left hand side. We present here a rule for non-recursive methods (recursion is handled by a straightforward adaptation of the classical recursion rule, see for example [3]).

For the statement $x := e_0!m(e_1, \ldots, e_n)$, we have the following proof rule (for the statement $u := e_0!m(e_1, \ldots, e_n)$ we have a similar rule):

$$\frac{\Big\{P \wedge \bigwedge_{i=1}^{k} v_i = \mathsf{nil} \wedge \mathsf{self} \notin \delta_{c'}\Big\}\langle U|c' : S\rangle\Big\{Q[e/r]\Big\}, \quad Q'[\bar{f}/\bar{z}] \rightarrow R[r/x]}{\Big\{P'[\bar{f}/\bar{z}]\Big\}\langle U|c : x := e_0!m(e_1, \ldots, e_n)\rangle\Big\{R\Big\}}$$

where $S \in Stat^{c'}$ and $e \in Exp_{d_0}^{c'}$ are the statement and expression occurring in the definition of the method m in the unit U, $u_1, \ldots, u_n$ are its formal parameters, $v_1, \ldots, v_k$ is a row of temporary variables that are *not* formal parameters ($k \geq 0$), r is a logical variable of type of the result of the method m (it is assumed that r does not occur in R), $\bar{f}$ is an arbitrary row of expressions (*not* logical expressions) in class c, and $\bar{z}$ is a row of logical variables, mutually different and different from r, such that the type of each z_i is the same as the type of the corresponding f_i. Furthermore, we assume given for each $d \in C$, a logical variable δ_d of type d^*. These variables will store the objects in class d that are blocked (as will be explained below). Finally, P' and Q' denote the result of applying to P and Q a *simultaneous* substitution having the "components" $[e_0/\mathsf{self}], [\delta_c \cdot \mathsf{self}/\delta_c], [e_1/u_1], \ldots, [e_n/u_n]$ (a formal definition will follow). We require that no temporary variables other than the formal parameters $u_1, \ldots, u_n$ occur in P or Q.

Before explaining the above rule let us first summerize the execution of a method call: First, control is transferred from the sender of the message to the receiver (*context switching*). The formal parameters of the receiver are initialized with the values of the expressions that form the actual parameters of the message and the other temporary variables are initialized to nil. Then the body S of the method is executed. After that the result expression e is evaluated, control is returned to the sender, the temporary variables are restored, and the result object is assigned to the variable x.

The first thing, the context switching, is represented by the substitutions $[e_0/\mathsf{self}]$, $[\delta_c \cdot \mathsf{self}/\delta_c]$ (the append operation is denoted by $\cdot$), and $[\bar{e}/\bar{u}]$ (where $\bar{e} = e_1, \ldots, e_n$ and $\bar{u} = u_1, \ldots, u_n$).

The transfer of control itself corresponds with a 'virtual' statement $\mathsf{self} := e_0$. Thus we see that if $P[e_0/\mathsf{self}]$ holds from the viewpoint of the sender then P holds from the viewpoint of the receiver after the transfer of control (i.e. after $\mathsf{self} := e_0$). Or, in other words, an assertion P as seen from the receiver's viewpoint is equivalent to $P[e_0/\mathsf{self}]$ from the viewpoint of the sender.

Definition 12. We have the following main cases of the substitution operation $[e/\mathsf{self}]$: $x[e/\mathsf{self}] = e\,.\,x$ and $\mathsf{self}[e/\mathsf{self}] = e$.

Note that this substitution changes the class of the assertion: $P[e_0/\mathsf{self}] \in Ass^c$ whereas $P \in Ass^{c'}$.

The (standard) substitution $[\delta_c \cdot \mathsf{self}/\delta_c]$ models the other aspect of the context switch, namely that the sender of the message is blocked when the receiver

is active. This aspect of the control switch thus corresponds with a virtual statement $\delta_c := \delta_c \cdot \mathsf{self}$. Moreover, the implicit check that the receiver itself is not blocked is expressed by the additional information $\mathsf{self} \notin \delta_{c'}$ in the precondition of the receiver (below is given an example of how this information can be used).

Now the passing of the parameters is simply represented by the *simultaneous* substitution $[\bar{e}/\bar{u}]$. (Note that we really need *simultaneous* substitution here, because u_i might occur in an e_j with $j < i$, but it should not be substituted again.) In reasoning about the body of the method we may also use the information that temporary variables that are not parameters are initialized to nil.

The second thing to note is the way the result is passed back. Here the logical variable r plays an important role. This is best understood by imagining after the body S of the method the statement $r := e$ (which is syntactically illegal, however, because r is a *logical* variable). In the sending object one could imagine the (equally illegal) statement $x := r$. Now if the body S terminates in a state where $Q[e/r]$ holds (a premiss of the rule) then after this "virtual" statement $r := e$ we would have a situation in which Q holds. Otherwise stated, the assertion Q describes the situation after executing the method body, in which the result is represented by the logical variable r, everything seen from the viewpoint of the receiver. Now if we context-switch this Q to the sender's side, and if it implies $R[r/x]$, then we know that after assigning the result to the variable x (our second imaginary assignment $x := r$), the assertion R will hold.

Now we come to the role of $\bar{f}$ and $\bar{z}$. We know that during the evaluation of the method the sending object becomes blocked, that is, it cannot answer any incoming messages. Therefore its instance variables will not change in the meantime. The temporary variables will be restored after the method is executed, so these will also be unchanged and finally the symbol self will retain its meaning over the call. All the expressions in class c (and in particular the f_i) are built from these expressions plus some inherently constant expressions and therefore their value will not change during the call. However, the method can change the variables of other objects and new objects can be created, so that the properties of these unchanged expressions *can* change. In order to be able to make use of the fact that the expressions $\bar{f}$ are constant during the call, the rule offers the possibility to replace them temporarily by the logical variables $\bar{z}$, which are automatically constant. So, in reasoning from the receiver's viewpoint (in the rule this applies to the assertions P and Q) the value of the expression f_i is represented by z_i, and in context switching f_i comes in again by the substitution $[\bar{f}/\bar{z}]$. Note that the constancy of $\bar{f}$ is guaranteed up to the point where the result of the method is assigned to x, and that x may occur in f_i, so that it is possible to make use of the fact that x remains unchanged right up to the assignment of the result.

Example 4. Let us illustrate the use of the above rule by a small example. Consider the unit $U = c : \langle m \Leftarrow (u_0) : x_1 := u_0 \uparrow x_2 \rangle$ and the program $\rho = \langle U | c : x_1 := u_1!m(x_2) \rangle$. We want to show

$$\Big\{u_1 \,.\, x_1 = x_1 \wedge \neg u_1 = \mathsf{self}\Big\}\rho\Big\{u_1 \,.\, x_1 = x_2 \wedge x_1 = u_1 \,.\, x_2\Big\}.$$

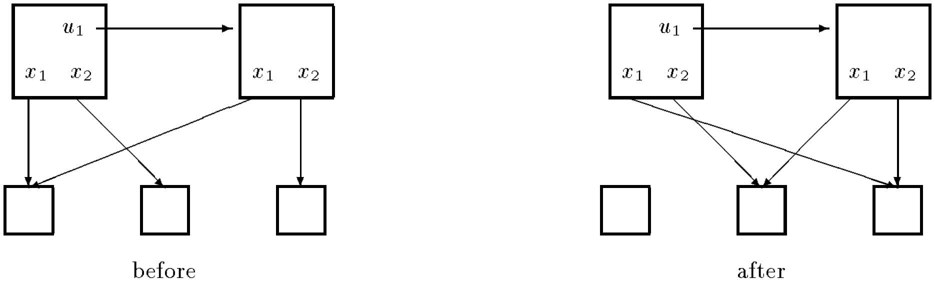

Fig. 2. The situation before and after sending the message (example 4)

So let us apply the rule (MI) with the following choices:

$$\begin{array}{lll} P & \equiv x_1 = z_1 \wedge \neg \mathsf{self} = z_2 & \\ Q & \equiv x_1 = u_0 \wedge r = x_2 \wedge \neg \mathsf{self} = z_2 & \\ R & \equiv u_1 \,.\, x_1 = x_2 \wedge x_1 = u_1 \,.\, x_2 & \\ k & = 0 & \text{(we shall use no } v_i\text{)} \\ f_1 & \equiv x_1 & \text{(represented by } z_1 \text{ in } P \text{ and } Q\text{)} \\ f_2 & \equiv \mathsf{self} & \text{(represented by } z_2 \text{ in } P \text{ and } Q\text{)} \end{array}$$

First notice that $P[u_1, x_2/\mathsf{self}, u_0][x_1, \mathsf{self}/z_1, z_2] \equiv u_1 \,.\, x_1 = x_1 \wedge \neg u_1 = \mathsf{self}$ so that the result of the rule is precisely what we want.

For the first premiss we have to prove

$$\Big\{x_1 = z_1 \wedge \neg \mathsf{self} = z_2\Big\} \langle U | c : x_1 := u_0 \rangle \Big\{x_1 = u_0 \wedge x_2 = x_2 \wedge \neg \mathsf{self} = z_2\Big\}.$$

This is easily done with the appropriate assignment axiom and the rule of consequence.

With respect to the second premiss, we have

$$\begin{array}{ll} Q[u_1, x_2/\mathsf{self}, u_0][x_1, \mathsf{self}/z_1, z_2] & \equiv\ u_1 \,.\, x_1 = x_2 \wedge r = u_1 \,.\, x_2 \wedge \neg u_1 = \mathsf{self} \\ R[r/x_1] & \equiv\ \mathsf{if}\ u_1 = \mathsf{self}\ \mathsf{then}\ r\ \mathsf{else}\ u_1 \,.\, x_1\ \mathsf{fi} = x_2 \wedge r = u_1 \,.\, x_2 \end{array}$$

It is quite clear that the first implies the second, and we can use this implication as an axiom.

In the above example we did not need to use the information represented by the logical variables δ_c. The following example illustrates the use of these variables in reasoning about deadlock.

Example 5. Consider the program $\rho = \langle U \mid c : x := \mathsf{self}!m \rangle$, where m is defined in U without parameters. Since this program obviously deadlocks (in general we will have to deal with longer cycles in the calling chain) we have the validity of $\Big\{\mathsf{true}\Big\}\rho\Big\{\mathsf{false}\Big\}$. This can be proved simply by observing that true is equivalent to $\mathsf{self} \in \delta_c \cdot \mathsf{self}$ and that the latter assertion can be obtained by applying the substitution $[\delta_c {\cdot} \mathsf{self}/\delta_c]$ to the assertion $\mathsf{self} \in \delta_c$. But this latter assertion, which

by the above we can use as the part P of the precondition of the receiver in the rule (MI), obviously contradicts the additional assumption that $\mathsf{self} \notin \delta_c$. Thus the entire precondition of the receiver reduces to false from which we can derive false as the postcondition of the body of the method m. From which in turn we can derive easily by rule (MI) the correctness formula above.

5 Conclusions

In this paper we have given a proof system for a sequential object-oriented programming language, called SPOOL, that fulfills the requirements we have listed in the introduction.

In [6] detailed proofs are given of both *soundness* (i.e. every derivable correctness formula is valid) and *(relative) completeness* (every valid correctness formula is derivable, assuming, as additional axioms, all the valid assertions). These proofs are considerable elaborations of the corresponding proofs of the soundness and completeness of a simple sequential programming language with recursive procedures (as described in, for example, [3] and [5]).

Related work To the best of our knowledge the proof system presented is the first sound and complete proof system for a sequential object-oriented language. In [1] and [10] different Hoare-style proof systems for sequential object-oriented languages are given which are based on the *global store* model as it has been developed for the semantics of Algol-like languages. This model however introduces a difference between the abstraction level of the assertion language and that of the programming language itself. Moerover, as observed in [1], the global store model gives rise to incompleteness.

Future research The proof rule for message passing, incorporating the passing of parameters and result, context switching, and the constancy of the variables of the sending object, is rather complex. It seems to work fine for our proof system, but its properties have not yet been studied extensively enough. It would be interesting to see whether the several things that are handled in one rule could be dealt with by a number of different, simpler rules.

We have considered in this paper only partial correctness. But we are currently working on extensions which allow one to prove absence of deadlock and termination.

In the present proof system the protection properties of objects are not reflected very well. While in the programming language it is not possible for one object to access the internal details (variables) of another one, in the assertion language this *is* allowed. In order to improve this it might be necessary to develop a system in which an object presents some abstract view of its behaviour to the outside world. Such an abstract view of an object we expect to consist of a specification of the *interface* of an object as it is used in [4, 6, 7, 8] for reasoning about systems composed of objects which execute in parallel.

Related to the above is the problem of a formal justification of the appropriateness of the abstraction level of a formalism for describing properties of

dynamically evolving object structures. We expect that such a formal justification involves a fully abstract semantics of the notion of on object. A related question, as already described above, is to what extent the problems with the incompleteness of the global store model are due to the particular choice of the abstraction level.

In any case, we expect that our approach provides an appropriate basis for specifying such high-level object-oriented programming mechanisms like subtyping, abstract types and inheritance.

Acknowledgement

This paper reports on joint work with P. America.

References

1. M. Abadi and K.R. M. Leino: A logic of object-oriented programs. Proceedings of the 7th International Joint Conference CAAP/FASE, vol. 1214 of Lecture Notes in Computer Science, April 1997.
2. P. America: Definition of the programming language POOL-T. ESPRIT project 415A, Doc. No. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, September 1985.
3. K.R. Apt: Ten years of Hoare logic: a survey — part I. ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp. 431–483.
4. P. America and F.S. de Boer: Reasoning about dynamically evolving process structures. Formal Aspects of Computing. Vol. 6, No. 3, 1994.
5. J.W. de Bakker: Mathematical Theory of Program Correctness. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.
6. F.S. de Boer: Reasoning about dynamically evolving process structures (A proof theory of the parallel object-oriented language POOL). PhD. Thesis. Free University, Amsterdam, 1991.
7. F.S. de Boer: A proof system for the parallel object-oriented laguage POOL. Proceedings of the seventeenth International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science, Vol. 443, Warwick, England, 1990.
8. F.S. de Boer: A compositional proof system for dynamic process creation. Proceedings of the sixth annual IEEE symposium on Logics in Computer Science (LICS), IEEE Computer Society Press, Amsterdam, The Netherlands, 1991.
9. J.M. Morris: Assignment and linked data structures. Manfred Broy, Gunther Schmidt (eds.): Theoretical Foundations of Programming Methodology. Reidel, 1982, pp. 35–41.
10. A. Poetzsch and P. Mueller: Logical foundations for typed object-oriented languages. Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET98).
11. D. S. Scott: Identity and existence in intuitionistic logic. M.P. Fourman, C.J. Mulvey, D.S. Scott (eds.): Applications of Sheaves. Proceedings, Durham 1977, Springer-Verlag, 1979, pp. 660–696 (Lecture Notes in Mathematics 753).