REASONING ABOUT HISTORIES IN OBJECT-BASED DISTRIBUTED SYSTEMS

F.S. de Boer

Utrecht University

Utrecht, The Netherlands

Email: frankb@cs.uu.nl

Abstract: The goal of this paper is to develop a compositional proof system for reasoning about systems consisting of a dynamically evolving collection of objects, which are all executing in parallel, and which know each other by maintaining and passing around references.

The proof system formalizes reasoning about these dynamic structures on an abstraction level that is at least as high as that of the programming language. The only operations on ‘pointers’ (references to objects) are testing for equality and dereferencing. Moreover, in a given state of the system, it is only possible to mention the objects that exist, or are known to exist, in that state. Objects that have not (yet) been created do not play a role.

Compositionality is achieved by recording for each object both the history of its communications and the history of its creations.

INTRODUCTION

The goal of this paper is to develop a compositional proof system for reasoning about the correctness of a certain class of parallel programs. A program of the language to be considered describes the behaviour of a whole system in terms of its constituents, *objects*. These objects have the following important properties: First of all, each object has an independent activity of its own: a local process that proceeds in parallel with all the other objects in the system.

Second, new objects can be created at any point in the program. The identity of such a new object is at first only known to itself and its creator, but from there it can be passed on to other objects in the system. Note that this also means that the number of processes executing in parallel may increase during the evolution of the system.

Objects possess some internal data, which they store in *variables*. The value of a variable is either an element of a predefined data type (Int or Bool), or it is a *reference* to another object. The variables of one object are not accessible to other objects. The objects can interact only by sending *messages*. A message is transferred synchronously from the sender to the receiver. It contains exactly one value; this can be an integer or a boolean, or it can be a reference to an object. Thus we see that a system described by a program consists of a dynamically evolving collection of objects, which are all executing in parallel, and which know each other by maintaining and passing around references. This means that the communication structure of the processes is determined dynamically, without any regular structure imposed on it a priori. This is in contrast to the static structure (a fixed number of processes, communicating with statically determined partners) in [3] and the tree-like structure in [6].

In this paper a Hoare logic is introduced which formalizes reasoning about these dynamic structures on an abstraction level that is at least as high as that of the programming language. This means that the only operations on 'pointers' (references to objects) are testing for equality and dereferencing. Moreover, in a given state of the system, it is only possible to mention the objects that exist, or are *known* to exist, in that state. Objects that have not (yet) been created do not play a role.

The specification of an object in isolation is given in terms of its internal data and its *interface* with the environment. Such an interface consists of two components. One component, the *communication history*, records the sequence of communications of the object with its environment. The other component, the *creation history*, records the objects which it has created itself. The idea of specifying the interface of a process in terms of its communications with its environment has been applied to a CSP-like language in [6]. We generalize this idea to cope with dynamic object creation and dynamically evolving object structures. A key feature of this generalization is that a specification of an object can only refer to those other objects which are *known*. The known objects (of an object) are those objects which it has received via a communication or which itself has created.

The local specifications of the objects of a complete system can be combined to yield a global specification under the condition that the local interfaces of the existing objects are *compatible*. Compatibility of the local communication histories is defined in terms of the global communication history of the complete system by means of a simple projection mechanism. Compatibility of the local creation histories is defined in terms of a global tree structure of creation (that is, at the root of the creation-tree we have the *root-object* and each object has as its children the objects it has created).

An interesting feature of this compositional Hoare logic of object creation is that it requires no logical formalization of the phenomenon of *aliasing*. This is because the effect of the execution of a statement by an object is described only with respect to its own internal state. Properties of the topology of a complete system are derived from the local communication and creation histories.

THE PROGRAMMING LANGUAGE

In this section we give a formal definition of the programming language. We assume as given a set C of *class names*, with typical element c . By this we mean that symbols like c, c', c_1 , etc. will range over the set C of class names. The set $C \cup \{\text{Int}, \text{Bool}\}$ of *data types*, with typical element d , we denote by C^+ . Here Int and Bool denote the types of the integers and booleans, respectively. For each $d \in C^+$ we assume given a set $Pvar_d$ of program variables of type d , with typical elements x (we assume that $Pvar_d \cap Pvar_{d'} = \emptyset$, for $d \neq d'$). A variable $x \in Pvar_d$ can refer to objects of type d .

We define the set $Stat^c$ of statements in class c , with typical element S . These statements describe the behaviour of a single object of class c .

Definition 1 Basic statements can be of the following forms (we omit the typing information):

$$x := e \mid x := \text{new} \mid x!e \mid x?y \mid ?y$$

The statement $x := e$ denotes the usual assignment statement (it is implicitly assumed that x and e are of the same type). Apart from the expressions `self` and `nil` the expression e is defined as usual (that is, in terms of the program variables and the standard repertoire of boolean and integer operations). The expression `self` denotes the object itself (thus $x := \text{self} \in Stat^c$ implies that x is of type c). The expression `nil` stands for ‘uninitialized’ (it can be of any type $d \in C^+$). A statement $x!e$, with x of some type $c \in C$, is called an *output* statement and statements like $x?y$, again with x of some type $c \in C$, and $?y$ are called *input* statements. Together they are called I/O statements. The execution of an output statement $x!e$ by an object α is always synchronized with the execution of a corresponding input statement $z?y$ or $?y$ by another object β . Such a pair of input and output statements are said to *correspond* if all the following conditions are satisfied:

- The variable x of the sending object α should refer to the receiving object β (therefore necessarily the type of the variable x coincides with the class of β).
- If the input statement to be executed is of the form $z?y$, then the variable z of the receiving object β should refer to the sending object α (again, this means that the type of the variable z coincides with the class of α).
- The type d of the expression e in the output statements should coincide with the type of the destination variable y in the input statement.

If an object tries to execute a I/O statement, but no other object is trying to execute a corresponding statement yet, it must wait until such a communication partner appears. If two objects are ready to execute corresponding I/O statements, the communication may take place. This means that the value of the expression e in the sending object α is assigned to the destination variable y in the receiving object β . When an object is ready to execute an input statement $?y$ there may be several objects ready to execute a corresponding output statement. One of them is chosen non-deterministically.

Statements are built up from these atomic statements by means of sequential composition, denoted by the semicolon ‘;’, the conditional construct *if-then-else-fi* and the iterative construct *while-do-od*. The meaning of these constructs we shall assume to be known. We only remark here that only for technical convenience we restrict to *deterministic* control structures: The proof system presented below can be generalized in a straightforward manner to control structures which describe, for example, *global non-determinism*.

Definition 2 Finally we define the set $Prog^{c_n}$ of *programs*, with typical element ρ , as follows:

$$\rho ::= \langle c_1 \leftarrow S_1, \dots, c_{n-1} \leftarrow S_{n-1} : S_n \rangle$$

Here we require that all the class names $c_1, \dots, c_n$ are different and that $S_n \in Stat^{c_n}$. Furthermore we require for every variable x occurring in ρ that its type d is among $c_1, \dots, c_n, Int, Bool$ and that in every *new-statement* $x := new$ the type d of the newly created object is among $c_1, \dots, c_{n-1}$.

The first part of a program consists of a finite number of class definitions $c_i \leftarrow S_i$, which determine the local processes of the instances of the classes $c_1, \dots, c_{n-1}$. Whenever a new object of class c_i is created, it will begin to execute the corresponding statement S_i . The program variables of S_i will refer to its own private state. These variables are initialized to nil. The second part specifies the local process S_n of the *root class* c_n . The execution of a program starts with the creation of a single instance of this root class, the *root object*, which begins executing the statement S_n . This root object can create other objects in order to establish parallelism. Due to the above restriction on the types of *new-statements*, the root object will always be the only instance of its class.

An example program

We illustrate the programming language by giving a program that generates the prime numbers up to a certain constant n . The program uses the sieve method of Eratosthenes. It consists of two classes. The class G (for ‘generator’) describes the behaviour of the root object, which consists of generating the natural numbers from 2 to n and sending them to an object of the other class P. The objects of the class P (for ‘prime sieve’) essentially form a chain of filters. Each of these objects remembers the first number it is sent; this will always be

Figure 1 Objects in the sieve program in a certain stage of the execution

a prime. From the numbers it receives subsequently, it will simply discard the ones that are divisible by its local prime number, and it will send the others to the next P object in the chain.

The class G makes use of two instance variables: f (for 'first') of type P and c (for 'count') of type Int (note that *n* is not an instance variable but an integer constant). The class P has three instance variables: m ('my prime') and b ('buffer') of type Int and l ('link') of type P. Here is the complete program:

```

⟨P ← ?m;
  if m ≠ nil
  then l := new;
    ?b;
    while b ≠ nil
    do if m / b then l ! b fi;
      ?b
    od;
    l ! b
  fi,
G ← f := new; c := 2;
  while c ≤ n do f ! c; c := c + 1 od;
  f ! nil)

```

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

THE ASSERTION LANGUAGE

In this section we define two different assertion languages. An *assertion* describes the state of (a part of) the system at one specific point during its execution. The first assertion language describes the *internal state* of a single object. This is called the *local* assertion language. The other one, the *global* assertion language, describes a whole system of objects.

Definition 3 The internal state of an object will consist of the following components: (1) an assignment of values to its program variables, (2) for each class *c* a set of created objects (more precisely, created by the object itself), and (3) a *local communication history*.

The local communication history of an object records the sequence of its communications. A communication consists of the following information: the kind of communication (input or output), the communication partner and the data communicated.

Definition 4 The *known* objects of a particular object at one specific point during its execution are those objects which it has received sofar via a communication or which itself has created.

We remark that a reference via a program variable to a known object does not necessarily need to exist (anymore). However, every the known object of an object is recorded as such in the set of created objects or in the local history.

Definition 5 A complete system of objects consists of a set of *existing* objects (i.e., the objects which have been created sofar) together with their internal states (as defined above), a root object, and a global communication history which records the sequence of communications that have occurred during the evolution of the whole system.

In the sequel we will restrict to complete systems of objects which are *compatible* in the following sense.

Definition 6 The objects of a complete system are compatible if the following holds:

Compatibility of the local communication histories: For every existing object its sequence of communications as recorded by the global history differs from its local communication history only with respect to possible occurrences of unknown senders in the local history (stemming from input statements $?y$).

Compatibility of the local creation histories: The local creation histories form an unordered tree with root the root-object and for each existing object a node with its children the objects it has created.

Restricting to complete systems of objects which form a tree of creation ensures compatibility of the local creation histories. It is worthwhile to observe that it is not sufficient to require the local creation histories of any two distinct existing objects to be disjoint. This requirement guarantees that each object has an unique creator but it does not exclude cycles in the creation ordering (for example, two objects creating each other).

In order to describe and reason about the internal state of an object and a complete system of objects we introduce the following sets of variables: For every type d we introduce infinite sets of *logical* variables $Lvar_d$ and $Lvar_{d^*}$, with typical element z . A variable $z \in Lvar_d$ ranges over objects of type d , whereas a variable $z \in Lvar_{d^*}$ ranges over (finite) sequences of objects of type d . In order to describe and reason about (finite sequences of) communications we introduce an infinite set of *history* variables $hvar$, with typical element

h. The type of a history variable we denote by r^* , where r is the type of a communication. The set of all types

$$C^+ \cup \{d^* \mid d \in C^+\} \cup \{r, r^*\}$$

with typical element t we denote by T .

Furthermore, we assume given a set F of operators, with typical element f . To each such an operator corresponds a type $t_1 \times \dots \times t_n \rightarrow t$, where n is the arity of f (a predicate like identity on integers is represented by a function $\text{Int} \times \text{Int} \rightarrow \text{Bool}$). The set of F will contain apart from the standard repertoire of arithmetical and boolean operations, operations on sequences like concatenation, append, etc.

The local assertion language

We define the set $LExp^c$ of expressions of the local assertion language in class c , with typical element l , as follows.

Definition 7 We omit the typing information.

$$\begin{array}{lcl} l & ::= & \text{self} \mid \text{nil} \mid \theta \mid d \ (d \in C) \\ & & \mid x \mid z \mid h \\ & & \mid \langle l_1, \text{self}, l_2 \rangle \mid \langle \text{self}, l_1, l_2 \rangle \\ & & \mid f(l_1, \dots, l_n) \ (f \in F) \end{array}$$

A local expression $l \in LExp^c$ is evaluated with respect to the internal state of an object in class c . The expression **self** denotes the object itself. The expression **nil** stands for ‘uninitialized’. The local communication history of the object is denoted by the (distinguished) history variable θ . The class name $d \in C$ denotes the sequence of objects in class d which have been created (by the given object). A program variable x will, as usual, denote the value assigned to it. A logical variable z involving a type $d \in C$ will denote a (sequence of) known object(s) of type d . A history variable h will denote a subsequence of the local history of the object (i.e. a subsequence of θ). A communication record $\langle l_1, \text{self}, l_2 \rangle$ encodes the information that the object l_2 has been received from l_1 (so l_1 is of some type $c \in C$). Analogously, the expression $\langle \text{self}, l_1, l_2 \rangle$ encodes the information that the object l_2 has been sent to l_1 . As a special case a communication record of the form $\langle \text{nil}, \text{self}, l \rangle$ will indicate that the identity of the sender is unknown (which is the case when an input statement $?y$ is executed).

In the sequel we will make use of the append operation $\cdot : r^* \times r \rightarrow r^*$: The expression $l \cdot \langle \text{nil}, \text{self}, l \rangle$ denotes the result of appending the communication record $\langle \text{nil}, \text{self}, l \rangle$ to the communication history l . The operation of concatenation (of sequences) we denote by $\circ$.

Some other examples of operations on histories which provide useful abstractions in practice are the operations $\text{out}_c(l)$ and $\text{in}_c(l)$, where l denotes a communication history. The operation out_c (in_c) yields the sequence of communicated (received) objects of type c . For example, the following local assertion

(these assertions are formally introduced below) of the internal state of a prime object of our prime generator expresses that every integer received which is not divisible by m has been sent (let $d = \text{Int}$):

$$\forall n(n \in \text{in}_d(\theta) \wedge m \nmid n \rightarrow n \in \text{out}_d(\theta))$$

(The relation $\in$ indicates whether an element occurs in a given sequence.)

Definition 8 The set $LAss^c$ of *local assertions* in class c , with typical element p , is defined as follows (again, we omit the typing information):

$$p ::= l \mid \neg p \mid p_1 \wedge p_2 \mid \exists z p \mid \exists z \in l(p) \mid \exists z \subseteq l(p) \mid \exists h \subseteq l(p)$$

Basic local assertions are boolean local expressions l (in class c). Quantification can be applied only to logical variables and history variables. Unrestricted quantification is only allowed in case of integer and boolean logical variables. Thus in an assertion $\exists z p$ the variable z is required to be of type Int or Bool . In a restricted quantification $\exists z \in l(p)$ and $\exists z \subseteq l(p)$ the variable z is of some type d and d^* , respectively, and l is of type d^* . In case z is of type d , $\exists z \in l(p)$ amounts to stating the existence of an object in the sequence denoted by l for which p holds. Similarly, if z is of type d^* , $\exists z \subseteq l(p)$ amounts to stating the existence of a subsequence of l for which p holds. In an assertion $\exists h \subseteq l(p)$, which involves a restricted quantification of a history variable h , l and h are of the same type, i.e. l and h are of type r^* . Such an assertion states the existence of a subsequence of the history denoted by l for which p holds.

Local assertions are evaluated with respect to the internal state of an object of class c and a logical environment. A logical environment assigns a value to each logical variable and each history variable. We require that every logical variable z of some type c (c^*) is assigned a (sequence of) known object(s) and that each history variable denotes a subsequence of the local history of the object. Given an object the history variable θ is interpreted as its local communication history and the local expression c refers to the sequence of objects of class c which it has created.

Restricted quantification of history variables and logical variables ranging over user-defined classes ensures that the evaluation of a local assertion indeed only depends on the internal state of an object (this is formally expressed by proposition 15).

The global assertion language

The set $GExp_d$ of *global expressions* of type d , with typical element g is defined as follows.

Definition 9 We omit the typing information.

$$\begin{aligned} g &::= \text{nil} \mid \theta \mid z \mid h \\ &\mid g.x \mid g.\theta \mid g.c \\ &\mid \langle g_1, g_2, g_3 \rangle \\ &\mid f(g_1, \dots, g_n) \ (f \in F) \end{aligned}$$

A global expression is evaluated with respect to a complete system of objects. The (distinguished) history variable θ denotes the *global* communication history. A logical variable z denotes a (sequence of) existing object(s). A history variable h denotes a subsequence of the global communication history. The expression $g.x$ (assuming g to be of some type c) denotes the value of the variable x of the object denoted by g . Similarly, the expressions $g.\theta$ and $g.c$ denote the *local* communication history of g and the sequence of objects created by g (of type c), respectively. Note that in this global assertion language we must explicitly specify the object of which we want to access the internal data. A communication record $\langle g_1, g_2, g_3 \rangle$ indicates that g_1 has sent g_2 the object g_3 .

We observe that, contrary to local expressions, global expressions give rise to the phenomenon of *aliasing*. For example, if g and g' refer to the same object then $g.x$ and $g'.x$ are aliases.

In the sequel we will assume for every $c \in C$ the presence of the restriction operator $|_c : r^* \times c \rightarrow r^*$. The history expression $g_1|_c g_2$, where g_1 is of type r^* and g_2 is of type c , denotes the subsequence of g_1 consisting of all the communication records involving g_2 as a communication partner. Moreover, we will assume an operator $\preceq$ of type $r^* \times r^* \rightarrow \text{Bool}$ with the following interpretation: $g_1 \preceq g_2$ if the histories denoted by g_1 and g_2 differ only with respect to the *unknown senders* of communications of g_1 , which are indicated as such by nil . As such g_2 contains more information. For example, in general it is the case that $g.\theta \preceq \theta|_c g$ (g being of type c), i.e. the local history of the object denoted by g provides less information than the sequence of communications involving g obtained from the global history θ because *locally* the communication partner is not always known (namely, in case of an input statement $?y$). More formally, the relation $\preceq$ is a pointwise extension of the corresponding relation on communication records defined by: $\langle g_1, g_2, g_3 \rangle \preceq \langle g'_1, g'_2, g'_3 \rangle$ if and only if $g_1 \preceq_{\text{nil}} g'_1$ and $g_2 = g'_2$ and $g_3 = g'_3$, where $g \preceq_{\text{nil}} g'$ if $g = \text{nil}$ or $g = g'$.

Definition 10 The set $G\text{Ass}$ of *global assertions*, with typical element P , is defined as follows:

$$P ::= g \mid \neg P \mid P_1 \wedge P_2 \mid \exists z P \mid \exists h P$$

Here g denotes a boolean global expression.

A global assertion is evaluated with respect to a complete system of objects and a logical environment which assigns (sequences of) existing objects to the logical variables and to each history variable a subsequence of the global communication history. The history variable θ is interpreted as the global history.

Quantification over (sequences of) integers and booleans is interpreted as usual. However, quantification over (sequences of) objects of some class c is interpreted as ranging only over the *existing* objects of that class, i.e., the objects that have been created up to the current point in the execution of the program. For example, the assertion $\exists z \text{ true}$ (z of some type $c \in C$) is false in some state iff there are no objects of class c in this state. Quantification over history variables is limited to the subsequences of the given global history.

Correctness formulas

In this section we define how we specify an object and a complete system of objects, using the well-known formalism of Hoare triples. We start with the specification of an object.

Definition 11 We define a *local correctness formula* to be of the following form: $\{p\}S\{q\}$. Here the assertion p is called the *precondition* and the assertion q is called the *postcondition*.

The meaning of such a correctness formula is formally described in the full paper in terms of a semantic function $M^c(S) \in \Sigma^c \rightarrow \mathcal{P}(\Sigma^c)$, where Σ^c , with typical element s , denotes the set of internal states of objects in class c (as described above). The set of all possible final outcomes of (terminating) computations of S in the initial state s is given by $M(S)(s)$. Non-determinism arises because of object-creation and input-statements. As a characteristic example we give the semantics of an input statement $x?y$, with y of type c . Let O_d denote the set of objects of type d (for $c \in C$, we assume this set of all possible objects of type c to be infinite). We define $s' \in M^c(x?y)(s)$ if s' results from s by assigning to y an *arbitrary* object $\alpha \in O_c$ and appending the corresponding communication record $\langle s(x), s(\text{self}), \alpha \rangle$ to the local communication history $s(\theta)$ ($s(x)$ denotes the value of x in s , $s(\text{self})$ denotes the 'owner' of the internal state s , and $s(\theta)$ denotes the local communication history of s). (In case of an input-statement $?y$ the communication record $\langle \text{nil}, s(\text{self}), \alpha \rangle$ is appended.)

The meaning of a local correctness formula is described informally as follows:

Every terminating execution of S by an object of class c starting from an internal state satisfying p will end in a state satisfying q .

Global correctness formulas describe the behaviour of a complete system.

Definition 12 A global correctness formula is of the form $\{p\}\rho\{Q\}$. The precondition p describes the initial internal state of the root object. Initially this root object is the only existing object, so it is sufficient for the precondition of a complete system to describe only its local state. On the other hand the final state of an execution of a complete system is described by a global assertion Q . The meaning of the global correctness formula $\{p\}\rho\{Q\}$ can be rendered informally as follows:

If the execution of the program ρ starts with a root object that satisfies the local assertion p and no other objects exist, then upon termination the final state will satisfy the global assertion Q .

The meaning of global correctness formulas is described formally in terms of a semantic function $M(\rho) \in \Sigma^{c_n} \rightarrow \mathcal{P}(\Sigma)$, where c_n is the root-class of ρ (Σ^{c_n} denotes the set of internal states of the root-object) and Σ , with typical element σ , denotes the set of complete systems which are compatible (see definition 6). We have the following *compositional* characterization of the initial/final state semantics of programs.

Definition 13 Let $\rho = \langle c_1 \leftarrow S_1, \dots, c_{n-1} \leftarrow S_{n-1} : S_n \rangle$, with $S_n \in \text{Stat}^{c_n}$. We have $\sigma \in M(\rho)(s)$ if for all objects α that do exist in σ it is the case that $\sigma(\alpha) \in M^{c_i}(S_i)(s_0)$ (assuming that α is in class c_i), where $\sigma(\alpha)$ denotes the internal state of α in σ and s_0 denotes the initial state of newly created objects in class c_i (i.e. all the variables of the object are initialized to nil). For the root-object s_0 equals the given initial state s .

In the full paper this compositional semantics is shown to be correct with respect to an operational semantics along the lines of [2].

We remark that the notion of correctness introduced above corresponds with the usual notion of *partial correctness*. However, the proof system presented below can be easily extended to the correctness of non-terminating executions by the inclusion of *invariants*.

THE PROOF SYSTEM

We start with the *local* proof system which allows us to reason about the correctness of a single object. For technical convenience only we restrict to *well-structured* programs: A program ρ is well-structured if the following holds: For every class c defined in ρ there exists a variable x_c of type c such that x_c occurs only in *new*-statements and all the creation-statements of objects in class c are of the form $x_c := \text{new}$; and for every type d , there exists a variable y_d of type d such that y_d occurs only in input-statements and all the input-statements of ρ are of the form $?y_d$ or $x?y_d$, for some program variable x .

The local proof system

We have the following standard axiomatization of an assignment statement. This axiomatization is based on the logical characterization of the *weakest precondition*.

Assignment

$$\{p[e/x]\}x := e\{p\}$$

Here $p[e/x]$ denotes the result of substituting every occurrence of x in p by the expression e . Note that aliasing does not arise here (because we are reasoning about the effect of an assignment with respect to the internal state of an object). Also it is worthwhile to notice that a simple assignment does not affect the scope of the quantifiers since it does not affect the set of known objects.

The axiomatization of object-creation will be based on the logical characterization of the *strongest postcondition*. In this way we overcome the problem with a weakest precondition axiomatization that the newly created object cannot be referred to in the state prior to its creation.

Creation The execution of a creation statement $x_c := \text{new}$ by some object α , with x of type c , will affect the value of x_c and the sequence of objects in class c created so far by α . This sequence of objects is represented in the local assertion language by c itself. To describe the strongest postcondition of such a creation statement $x := \text{new}$ with respect to a given precondition p we therefore

introduce a fresh variable z of type c^* . This variable z will represent in the new state the old value of c . Note that since we restrict to well-structured programs the old value of x_c is then given, in the postcondition, by $last(z)$, i.e. the last element of z . The local assertion $p[z/c, last(z)/x_c]$ (i.e. the result of replacing simultaneously c and x_c in p by z and $last(z)$) then expresses that p holds for the old values of c and x_c . The new values of c and x_c are described by $c = z \cdot x_c \wedge x_c \notin z$, i.e. c is obtained by appending the value of x to z and the value of x does not appear in z .

We thus arrive at the following axiomatization of a **new**-statement.

$$\{p\}x_c := \text{new}\{\exists z \subseteq c \ (q \wedge c = z \cdot x_c \wedge x_c \notin z)\}$$

where $q = p[z/c, last(z)/x_c]$. Note that the quantification of z restricted to c is in fact semantically redundant because this information is already implied by $c = z \cdot x_c$.

An output statement is simply axiomatized as follows.

Output

$$\{p[\theta \cdot \langle \text{self}, x, e \rangle / \theta]\}x!e\{p\}$$

The execution of an output statement $x!e$ only affects the local history θ and is modeled by the substitution operation $p[\theta \cdot \langle \text{self}, x, e \rangle / \theta]$ which simply replaces (every occurrence of) the initial local history θ in p by the resulting history $\theta \cdot \langle \text{self}, x, e \rangle$.

In general the object received by (the execution of) an input-statement may be an object unknown so-far, and as such we cannot refer to it in the state prior to its reception. This phenomenon complicates an axiomatization of an input-statement by means of a logical characterization of the weakest precondition. Therefore we axiomatize input-statements, like creation-statements, by means of a logical characterization of the strongest postcondition.

Input1 Let $q = p[l/y_d]$, where $l = last(in_d(\theta))$.

$$\{p\}x?y_d\{\exists h \subseteq \theta \ (q[h/\theta] \wedge \theta = h \cdot \langle x, \text{self}, y_d \rangle)\}$$

Since we restrict to well-structured programs the old value of y_d is given by $last(in_d(\theta))$, i.e. the last element of the sequence of objects (of type d) which have been received via some communication (as recorded by θ). The quantified history variable h in the postcondition plays the role of the initial local communication history. We observe that the postcondition does not constrain (the value of) y_d . Thus it can be either an already known object or any object unknown hitherto.

Input2 Let $q = p[l/y_d]$, where $l = last(in_d(\theta))$.

$$\{p\}x?y_d\{\exists h \subseteq \theta \ (q[h/\theta] \wedge \theta = h \cdot \langle \text{nil}, \text{self}, y \rangle)\}$$

The fact that the sender is not known is represented by nil .

The rules for the other constructs are as usual.

The global proof system

In this section we show how to specify the global behaviour of a complete system in terms of the local behaviour of its objects.

First we define a transformation of a local assertion to a global one. This transformation will be used to specify the global behaviour of a program in terms of the local behaviour of its objects.

Definition 14 Given a local assertion p in class c and a global expression g of type c we define a substitution operation $p[g/\text{self}]$ which results in a global assertion. This assertion denotes the result of evaluating the local assertion p in the object denoted by the global expression g . The definition proceeds by a straightforward induction on the complexity of the local assertion l . The following are the main cases: $\text{self}[g/\text{self}] = g$, $x[g/\text{self}] = g.x$, $\theta[g/\text{self}] = g.\theta$, $c[g/\text{self}] = g.c$.

We have the following crucial property of the above transformation.

Proposition 15 Let s be an internal state of an object α (in class c) and ω be a logical environment such that ω assigns to each logical variable a (sequence of) known (with respect to s) object(s); to each history variable it assigns a subsequence of the local communication history of s ; and to the history variable θ itself it assigns the local communication history (of s). For any local assertion p (in class c) we have

$$s, \omega \models p \text{ iff } \sigma, \omega\{\alpha/z\} \models p[z/\text{self}]$$

for any (global) state σ such that the internal state of α in σ equals s . Here z is a logical variable of type c and $\omega\{\alpha/z\}$ denotes the result of assigning α to z in ω .

This property thus states that the truth of a local assertion indeed only depends on the internal state. Crucial for this property to hold is the restricted quantification of history variables and logical variables ranging over user-defined objects. For example, unrestricted quantification in an assertion like $\forall z. \text{false}$, where z is of type c , which thus states that there does not exist an object in class c , clearly does not satisfy the above property.

Definition 16 In the following definition, let $\rho = \langle c_1 \leftarrow S_1, \dots, c_{n-1} \leftarrow S_{n-1} : S_n \rangle$, with $S_n \in \text{Stat}^{c_n}$. The program rule of the global proof system has the following form:

$$\frac{\{Init_1\}S_1\{q_1\}, \dots, \{Init_{n-1}\}S_{n-1}\{q_{n-1}\}, \{p \wedge Init_n\}S_n\{q_n\}}{\{p\}\rho\{\bigwedge_i \forall z_i (q_i[z_i/\text{self}])\}}$$

(The index-variable i ranges over $\{1, \dots, n\}$.) The premises of this rule should be interpreted as being derivable from the local proof system. Here $Init_k$, for $1 \leq k < n$, denotes the local assertion $\bigwedge_{x \in Pvar(S_k)} (x = \text{nil}) \wedge \theta = \text{nil} \wedge \bigwedge_i c_i = \text{nil}$.

This assertion describes the initial local state of newly created objects of class c_k (all its variables including its history variables are set to nil). On the other hand, $Init_n$ denotes the local assertion $\bigwedge_c \bigwedge_{x \in Pvar_c(S_n)} (x = \text{nil}) \wedge \theta = \text{nil} \wedge \bigwedge_i c_i = \text{nil}$, which describes the initial local state of the root object. So, initially all the variables of the root object of a type different from `Int` or `Bool` are set to nil (including the history variables). This reflects the fact that initially only the root object exists. In the conclusion of the program rule we take as precondition the precondition of the local process of the root object because initially only this object exists. The postcondition expresses that the final internal state of every existing object of class c_i is characterized by the local assertion q_i .

Definition 17 We have the following consequence rule for programs:

$$\frac{p \rightarrow p', \quad \{p'\}\rho\{Q'\}, \quad Q' \rightarrow Q}{\{p\}\rho\{Q\}}$$

The validity of the logical implication $Q' \rightarrow Q$ in the above consequence rule for programs is defined with respect to complete systems consisting of a collection of objects which are compatible, as defined in Definition 6.

The compatibility of the local communication histories (of objects in class c) can be logically expressed by the global assertion $\forall z(z.\theta \preceq \theta|_c z)$, where z is a logical variable of type c . This assertion expresses that for every existing object in class c its sequence of communications as recorded by the global history (denoted by the expression $\theta|_c z$) differs from its local history $z.\theta$ only with respect to the possible occurrences of unknown senders in the local history.

Compatibility of the local creation histories can be expressed logically in terms of a global creation history. A creation history denotes a sequence of creation records described by global expressions of the form $\langle g, g' \rangle$ which indicate that the object g has created g' . The tree structure of the creation ordering then can be imposed by requiring, among others, that the global history satisfies the property that *no object occurs in the global creation history before its creation*. This property can be easily expressed in the global assertion language (assuming the presence of history variables ranging over sequences of creation records).

CONCLUSION

In this paper a compositional proof system for object-based distributed systems is introduced. In the full paper a formal proof of the soundness and completeness of the system is given.

In [1] a non-compositional proof system for dynamically evolving object structures is introduced based upon a generalization of the ideas underlying the proof theory of CSP [3]. An interesting consequence of a compositional reasoning pattern is that it requires no formalization of the phenomenon of aliasing (this contrasts with the general emphasis in [1] on aliasing in reasoning about object-oriented systems).

In [4] a compositional proof system is presented for a restricted class of distributed systems which are based on a FIFO-buffered asynchronous commu-

nication mechanism and sequential control structures which only allow *local non-determinism*. In this paper we have presented a generalization of the proof system in [4] which applies also to distributed systems based on a synchronous communication mechanism and which allows more rich control structures involving global non-determinism. This generalization consists of the introduction of general communication history variables which record the sequence of communications of a system. In this paper it is shown that in general in object-based distributed system the information about the communications of an object can be logically separated in the specification of its interface from the information about its creations. This separation of concerns allows for a simple logical formulation of the compatibility of the interfaces of the objects of a whole system.

Further research will concern extensions of the proof system to typical object-oriented concepts like *sub-typing* and *inheritance*. Another line of research is the development of specification and verification tools based on verification systems like PVS [5].

References

- [1] P. America and F.S. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*. Vol. 6, No. 3, 1994.
- [2] P. America, J.W. de Bakker, J.N. Kok and J.J.M.M. Rutten. Operational semantics for a parallel object-oriented language. *Conference Record of the 13th Symposium on Principles of Programming Languages (POPL)*, St. Petersburg, Florida, 1986, pp. 194-208.
- [3] K.R. Apt, N. Francez, W.P. de Roever: A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3, July 1980, pp. 359-385.
- [4] F.S. de Boer. Reasoning about asynchronous communication in dynamically evolving object structures. *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98)*, 1998.
- [5] S. Owre, J. Rushby and N. Shankar. PVS: A Prototype Verification System. *Proceedings of the 11th Conference on Automated Deduction*. Vol. 607, *Lecture Notes in Artificial Intelligence*, 1992.
- [6] J. Zwiers, W.P. de Roever, P. van Emde Boas: Compositionality and concurrent networks: soundness and completeness of a proof system. *Proceedings of the 12th ICALP, Nafplion, Greece, July 15-19, 1985*, Springer LNCS 194, pp. 509-519.