Formal Aspects of Computing (1994) 6: 269–316
© 1994 BCS

Formal Aspects of Computing

Reasoning about Dynamically Evolving Process Structures

Pierre America[1] and Frank de Boer[2]

[1]Philips Research Laboratories, Eindhoven, The Netherlands;
[2]Department of Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands

Keywords: Proof theory; Pre- and post-conditions; Process creation; Dynamically evolving process structures; Cooperation test; Soundness; Completeness

Abstract. We develop a Hoare-style proof system for reasoning about the behaviour of processes that interact via a dynamically evolving communication structure.

1. Introduction

The goal of this paper is to develop a formal system for reasoning about the correctness of a certain class of parallel programs. We shall consider programs written in a programming language, which we simply call P. The language P is a simplified relative of POOL, a parallel object-oriented language [Ame89]. POOL makes use of the structuring mechanisms of object-oriented programming [Mey88], integrated with concepts for expressing concurrency: processes and communication.

A program of our language P 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.

Correspondence and offprint requests to: Frank de Boer, Department of Computing Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands.

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. (This is the only essential difference between P and POOL: in POOL communication proceeds by a rendezvous mechanism, where a method, a kind of procedure, is invoked in the receiving object in response to a message.) Thus we see that a system described by a program in the language P 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 [AFR80] and the tree-like structure in [ZRE85].

One of the main proof theoretical problems of such an object-oriented language is how to describe and reason about dynamically evolving *pointer structures*. 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
  - dereferencing (looking at the value of an instance variable of the referenced object)
- In a given state of the system, it is only possible to mention the objects that exist in that state. Objects that have not (yet) been created do not play a role.

Strictly speaking, direct dereferencing is not even allowed in the programming language, because each object has access to its own instance variables only. However, without direct dereferencing it is not clear how to express in the assertion language even the most trivial properties of pointer structures.

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. Therefore we have to extend our assertion language to make it more expressive. We will do so by allowing the assertion language to reason about *finite sequences* of objects. (This is not uncommon in proof systems dealing with more data types than integers only [TuZ88].) Furthermore we have to define some special substitution operations to model aliasing and the creation of new objects.

To deal with parallelism, the proof theory of partial correctness we shall develop uses the concepts of *cooperation test*, *global invariant*, *bracketed section*, and *auxiliary variables*. These concepts have been developed in the proof theory of CSP [AFR80], and have been applied to quite a variety of concurrent programming languages [HoR86]. Described very briefly, this proof method applied to our language consists of the following elements:

- A *local* stage. Here we deal with all statements that do not involve communication or object creation. These statements are proved correct with respect to pre- and postconditions in the usual manner of sequential programs [Apt81, Bak80, Hoa69]. At this stage, we use *assumptions* to describe

the behaviour of the communication and creation statements. These will be verified in the next stage. In this local stage, a *local assertion language* is used, which only talks about the current object in isolation.

- An *intermediate* stage. In this stage the above assumptions about communication and creation statements are verified. Here a *global assertion language* is used, which reasons about all the objects in the system. For each creation statement and for each pair of possibly communicating send and receive statements it is verified that the specification used in the local proof system is consistent with the global behaviour.
- A *global* stage. Here some properties of the system as a whole can be derived from a kind of standard specification that arises from the intermediate stage. Again the global assertion language is used.

We have proved that the proof system is sound and complete with respect to a formally defined semantics. Soundness means that everything that can be proved using the proof system is indeed true in the semantics. On the other hand, completeness means that every true property of a program that can be expressed using our assertion language can also be proved formally in the proof system. Due to the abstraction level of the assertion language we had to modify considerably the standard techniques for proving completeness.

Our paper is organised as follows: In the following section we describe the programming language P; in section 3 we define two assertion languages, the local one and the global one. Then, in section 4 we describe the proof system. Section 5 presents an example of a correctness proof for a nontrivial program. Section 6 presents the semantics of the programming language, of the assertion languages, and of the correctness formulas. In section 7 we prove the soundness of the proof system and in section 8 we prove completeness. The expressibility of the assertion languages is studied in section 9. Finally, in section 10 we draw some conclusions.

2. The Programming Language

In this section we define the programming language P of which we shall study the proof theory. This language is related to CSP [Hoa78], in that it describes a number of processes that communicate synchronously by transmitting values to each other, but it has the additional possibility of dynamically creating processes and manipulating references to processes. It can also be compared to Smalltalk [GoR84], since it describes a dynamically evolving collection of objects where each has its own private data, but in P each object is provided with an autonomous local process and it communicates with other objects simply by exchanging a value instead of invoking a method.

A system, the result of executing a program written in P, consists of *objects*. On the one hand these objects have the properties of processes, that is, each of them has an internal activity, which runs in parallel with all the other objects in the system. On the other hand, objects are in some way like data records: they contain some internal data, and they have the ability to act on these data. An important characteristic of the objects in the language P is that they can be created dynamically: Whenever required during the execution of a program, a new object can be called into existence.

An object stores its internal data in *variables* (also called *instance variables*

to distinguish them from the other kinds of variables that we shall need in the assertion language). A variable can contain a reference to an object, which can be another object, or possibly the object under consideration itself. Alternatively, it can contain an element of a standard, built-in data type, of which our language P contains only integers and booleans. The contents of a variable can be changed by an assignment statement. The variables of one object cannot be accessed directly by other objects. They can only be read and changed by the object to which they belong.

Interaction between objects takes place by sending messages. The language P uses a synchronous communication mechanism, that is, the sender and receiver of a message perform the communication at the same time; the one that reaches its communication statement first will wait for its partner. The sender of a message must always specify the receiver explicitly. The receiver however, has the possibility of mentioning the sender that it wants to communicate with, but it can also omit the indication of its communication partner, in which case it is willing to communicate with any sender that sends a message of the correct type. A message consists of a data value, which is transferred from the sender to the receiver. This data value can be a reference to an object or it can be an integer or boolean.

In order to describe by a program the unbounded number of objects in a system, we group them into *classes*. In the language P all objects have the same structure of variables (each object has its own private variables, but the variables of all objects have the same names and types). The objects in one class however additionally execute identical local processes. In this way a class can be considered as a blueprint for creating new instances.

Let us now give a formal definition of the language P. 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 $C \cup \{\mathsf{Int}, \mathsf{Bool}\}$, 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 $IVar_d$ to be the set of instance variables of type d, with typical elements x and y (we assume $IVar_d \cap IVar_{d'} = \emptyset$ whenever $d \neq d'$, so that the type of each variable is uniquely determined). Such a variable $x \in IVar_d$ can refer to objects of type d only. The set of all instance variables ($\bigcup_d IVar_d$) we denote by $IVar$.

Definition 2.1. We define the set Exp_d^c of expressions of type d in class c, with typical element e. An expression $e \in Exp_d^c$ can be evaluated by an object of class c and the object to which it refers will be of type d.

These expressions are defined as follows:

$$
\begin{array}{rcll}
e(\in Exp_d^c) & ::= & x & \text{if } x \in IVar_d \\
 & | & \mathsf{self} & \text{if } d = c \\
 & | & \mathsf{nil} & \\
 & | & \mathsf{true} \mid \mathsf{false} & \text{if } d = \mathsf{Bool} \\
 & | & n & \text{if } d = \mathsf{Int} \\
 & | & e_1 + e_2 & \text{if } e_1, e_2 \in Exp_d^c,\ d = \mathsf{Int} \\
 & \vdots & & \\
 & | & e_1 \doteq e_2 & \text{if } e_1, e_2 \in Exp_{d'}^c,\ d = \mathsf{Bool}
\end{array}
$$

An expression e will be evaluated by a certain object α of class c. An expression of the form x denotes the value of the variable x that belongs to the object α. The expression self denotes the object α itself. The expression nil denotes no object at all. It can be used for every type, including Int and Bool. The symbols true and false stand for the corresponding values of type Bool. Every integer n can occur as an expression of type Int; it simply denotes itself. We assume that the standard arithmetic and comparison operations on integers are available, but we list only the operator '+'. We assume that all these operations result in nil whenever an error occurs (e.g., division by zero or nil as an operand). Finally, for every type we have a test for equality. The expression $e_1 \doteq e_2$ evaluates to true whenever e_1 and e_2 denote the same object (or both denote *no* object, viz. nil). Note that in the programming language we put a dot over the equality sign ($\doteq$) to distinguish it from the equality sign we use in the metalanguage.

Definition 2.2. We next 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.

Statements can be of the following forms:

$S(\in STat^c)$	$::=$	$x := e$	if $x, e \in Exp^c_d$
	$\mid$	$x :=$ new	if $x \in IVar_{c'}$
	$\mid$	$x!e$	if $x \in IVar_{c'},\ e \in Exp^c_d$
	$\mid$	$x?y$	if $x \in IVar_{c'}$
	$\mid$	$?y$	
	$\mid$	$S_1 ; S_2$	if $S_1, S_2 \in Stat^c$
	$\mid$	if e then S_1 else S_2 fi	if $e \in Exp^c_{\mathsf{Bool}},\ S_1, S_2 \in Stat^c$
	$\mid$	while e do S od	if $e \in Exp^c_{\mathsf{Bool}},\ S \in Stat^c$

An object executes the assignment statement $x := e$ by first evaluating the expression e at the right-hand side and then storing the result in its own variable x The execution of the new-statement $x :=$ new ($x \in IVar_{c'}$) by the object α consists of creating a new object β of class c' and making the variable x of the creator α refer to it. The instance variables of the new object β are initialised to nil and β will immediately start executing its local process. It is not possible to create new elements of the standard data types Int and Bool.

A statement $x!e$ is called an *output* statement and statements like $x?y$ and $?y$ are called *input* statements (in both the statements $x!e$ and $x?y$ the variable x is required to be of some type $c \in C$). Together they are called I/O statements. The execution of an output statement $x_1!e$ by an object α is always synchronised with the execution of a corresponding input statement $x_2?y$ 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_1 of the sending object α should refer to the receiving object β (therefore necessarily the type of the variable x_1 coincides with the class of β).
- If the input statement to be executed is of the form $x_2?y$, then the variable x_2 of the receiving object β should refer to the sending object α (again, this means that the type of the variable x_2 coincides with the class of α).
- The type 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.

Definition 2.3. Finally we define the set *Prog* of *programs*, with typical element ρ, as follows:

$$\rho ::= \langle c_1 \leftarrow S_1, \ldots, c_n \leftarrow S_n \rangle$$

Here we require that all the class names $c_1, \ldots, c_n$ are different and that $S_i \in STat^{c_i}$ $(1 \leq i \leq n)$. Furthermore we require for every variable x occurring in ρ that its type d is among $c_1, \ldots, c_n$, Int, Bool, and that for no variable x of type c_n an assignment $x :=$ new occurs in ρ.

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 $c_1, \ldots, c_n$. Whenever a new object of class c_i is created, it will begin to execute the corresponding statement S_i. The execution of a program starts with the creation of a single instance of class c_n, the *root object*, which begins executing the statement S_n. This root object can create other objects in order to establish parallelism. Note that no other objects of the root-class can be created.

Definition 2.4. For an expression e, a statement S and a program ρ the set of instance variables occurring in e, S and ρ is denoted by $IVar(e)$, $IVar(S)$ and $IVar(\rho)$, respectively.

2.1. 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 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⟩
```

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

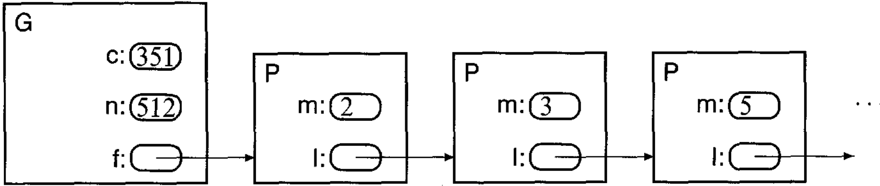

Fig. 1. Objects in the sieve program in a certain stage of the execution

3. 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. It will be used in the local proof system. The other one, the *global* assertion language, describes a whole system of objects. It will be used in the intermediate and global proof systems.

For the sake of expressiveness of the assertion languages we introduce for every $d \in C^+$ a new set $LVar_d$ of logical variables of type d, with typical element z. To be able to describe interesting properties of pointer structures we also introduce logical variables ranging over *finite sequences* of objects. To do so we first introduce for every $d \in C^+$ the type d^* of finite sequences of objects of type d. We define $C^* = \{d^* : d \in C^+\}$ and take $C^\dagger = C^+ \cup C^*$, with typical element a. Now in addition we assume for every $d \in C^+$ the set $LVar_{d^*}$ of logical variables of type d^*, which range over finite sequences of elements of type d. Therefore we now have a set $LVar_a$ of logical variables of type a for every $a \in C^\dagger$. For any two distinct types a and a' we have that $LVar_a \cap LVar_{a'} = \emptyset$, so that the type of every logical variable is uniquely determined.

3.1. The Local Assertion Language

First we introduce the set of *local expressions.*

Definition 3.1. The set $LExp_a^c$ of local expressions of type a in class c, with typical element l, is defined as follows:

$$
\begin{array}{llll}
l(\in LExp_a^c) & ::= & z & \text{if } z \in LVar_a \\
& \mid & x & \text{if } x \in IVar_a \\
& \mid & \mathsf{self} & \text{if } a = c \\
& \mid & \mathsf{nil} & \\
& \mid & n & \text{if } a = \mathsf{Int} \\
& \mid & \mathsf{true} \mid \mathsf{false} & \text{if } a = \mathsf{Bool} \\
& \mid & |l| & \text{if } l \in LExp_{d*}^c,\ a = \mathsf{Int} \\
& \mid & l_1 : l_2 & \text{if } l_1 \in LExp_{d*}^c,\ l_2 \in LExp_{\mathsf{Int}}^c,\ a = d \\
& \mid & l_1 + l_2 & \text{if } l_1, l_2 \in LExp_{\mathsf{Int}}^c \\
& \vdots & & \\
& \mid & l_1 \doteq l_2 & \text{if } l_1, l_2 \in LExp_d^c,\ a = \mathsf{Bool}
\end{array}
$$

For sequence types the expression nil denotes the empty sequence. The expression $|l|$ denotes the length of the sequence l. The expression $l_1 : l_2$ denotes the nth element of the sequence represented by l_1, where n is the integer value of l_2 (if l_2 is less than 1 or greater than $|l_1|$, the result is nil). Note that in the boolean expression $l_1 \doteq l_2$ the subexpressions l_1 and l_2 are required to be of some simple type $d \in C^+$. This restriction is introduced to facilitate the definition of the substitution operations modelling aliasing and the creation of objects (see definitions 4.3. and 4.5.).

Definition 3.2. The set $LAss^c$ of *local assertions* in class c, with typical element p, is defined as follows:

$$
\begin{array}{llll}
p(\in LAss^c) & ::= & l & \text{if } l \in LExp_{\mathsf{Bool}}^c \\
& \mid & \neg p & \text{if } p \in LAss^c \\
& \mid & p_1 \wedge p_2 & \text{if } p_1, p_2 \in LAss^c \\
& \mid & \exists z\, p & \text{if } z \in LVar_a,\ a = d, d^*,\ d = \mathsf{Int}, \mathsf{Bool}, \\
& & & \text{and } p \in LAss^c
\end{array}
$$

A local assertion is built up from boolean local expressions and the usual logical connectives. We shall regard other logical connectives ($\vee, \rightarrow, \forall$) as abbreviations for combinations of the above ones. We introduce the following syntactic notions. For a local expression l the set of instance (logical) variables occurring in l is denoted by $IVar(l)$ ($LVar(l)$). Similarly $IVar(p)$ denotes the set of instance variables occurring in the local assertion p. The set of logical variables occurring free in a local assertion p is denoted by $LVar(p)$.

An example of a local assertion in the class P of the program at the end of section 2 is

$$\neg(\mathsf{b} \doteq \mathsf{nil}) \rightarrow \forall i(2 \leq i \wedge i \leq \mathsf{m} \rightarrow \neg(i \mid \mathsf{b})),$$

which we might abbreviate to $\mathsf{b} \not\doteq \mathsf{nil} \rightarrow \forall i(2 \leq i \leq \mathsf{m} \rightarrow i \nmid \mathsf{b})$. Here i is a logical

variable of type Int. (Most of our examples will be in the context of the program at the end of section 2.)

Local expressions $l \in LExp_a^c$ and local assertions $p \in LAss^c$ are evaluated with respect to the local state of an object of class c, determining the values of its instance variables, plus a logical environment, which assigns values to the logical variables. Therefore they talk about this single object in isolation. It is important to note that we allow only quantification of logical variables ranging over (sequences of) integers and booleans. (By the way, the value nil is *not* included in the range of quantifications.) Quantification over other types would require knowledge of the set of existing objects, which is not available locally. This is made formal in lemma 6.7. of the section 6 on semantics.

3.2. The Global Assertion Language

Next we define the global assertion language.

Definition 3.3. The set $GExp_a$ of *global expressions* of type a, with typical element g, is defined as follows:

$$
\begin{array}{rcll}
g(\in GExp_a) & ::= & z & \text{if } z \in LVar_a \\
& | & \mathsf{nil} & \\
& | & n & \text{if } a = \mathsf{Int} \\
& | & \mathsf{true} \mid \mathsf{false} & \text{if } a = \mathsf{Bool} \\
& | & g.x & \text{if } g \in GExp_c,\ x \in IVar_a \\
& | & |g| & \text{if } g \in GExp_{d^*},\ a = \mathsf{Int} \\
& | & g_1 : g_2 & \text{if } g_1 \in GExp_{d^*},\ g_2 \in GExp_{\mathsf{Int}},\ a = d \\
& | & g_1 + g_2 & \text{if } g_1, g_2 \in GExp_{\mathsf{Int}} \\
& \vdots & & \\
& | & \mathsf{if}\ g\ \mathsf{then}\ g_1\ \mathsf{else}\ g_2\ \mathsf{fi} & \\
& & & \text{if } g \in GExp_{\mathsf{Bool}},\ g_1, g_2 \in GExp_a \\
& | & g_1 \doteq g_2 & \text{if } g_1, g_2 \in GExp_d
\end{array}
$$

A global expression is evaluated with respect to a complete system of objects plus a logical environment. A complete system of objects consists of a set of existing objects together with their local states. The expression $g.x$ denotes the value of the variable x of the object denoted by g. Note that in this global assertion language we must explicitly specify the object of which we want to access the internal data. The conditional expression if g_0 then g_1 else g_2 fi is introduced to facilitate the handling of aliasing (see definition 4.3). If the condition is nil, then the result of the conditional expression is nil, too.

Definition 3.4. The set $GAss$ of *global assertions*, with typical element P, is defined as follows:

$$
\begin{array}{rcll}
P & ::= & g & \text{if } g \in GExp_{\mathsf{Bool}} \\
& | & \neg P & \\
& | & P_1 \wedge P_2 & \\
& | & \exists z\, P &
\end{array}
$$

Global assertions are built up from boolean global expressions and the usual logical connectives. Again, other logical connectives are regarded as abbreviations. We introduce the following syntactic notions: For a global expression g the set of instance (logical) variables occurring in g is denoted by $IVar(g)$ ($LVar(g)$). Similarly $IVar(P)$ denotes the set of instance variables occurring in the global assertion P. The set of logical variables occurring free in a global assertion P is denoted by $LVar(P)$.

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 $\exists z\,\mathsf{true}$ is false in some state iff there are no objects of class c in this state, assuming the type of $z \in LVar_c$. More interestingly, the assertion

$$\forall p\, \exists s(s : 1 \doteq g.\mathsf{f} \wedge s : |s| \doteq p \wedge \forall i(1 \leq i < |s| \rightarrow (s : i).\mathsf{l} \doteq s : (i+1)))$$

(with $p \in LVar_{\mathsf{P}}$ and $s \in LVar_{\mathsf{P}^*}$) expresses that every object of class P is a member of the l-linked chain that starts with $g.\mathsf{f}$ (where g is the generator object).

Next we define a transformation of a local expression or assertion to a global one. This transformation will be used to verify the assumptions made in the local proof system about the I/O and new-statements. These assumptions are formulated in the local language. As the reasoning in the cooperation test uses the global assertion language we have to transform these assumptions from the local language to the global one.

Definition 3.5. Given a local expression $l \in LExp_a^c$ and a global expression $g \in GExp_c$ we define a global expression $l \downarrow g$. This expression denotes the result of evaluating the local expression l in the object denoted by the global expression g. The definition proceeds by induction on the complexity of the local expression l:

$$\begin{array}{lcll} l \downarrow g & = & l & \text{if } l = z, \mathsf{nil}, n, \mathsf{true}, \mathsf{false} \\ x \downarrow g & = & g.x & \\ \mathsf{self} \downarrow g & = & g & \\ (l_1 + l_2) \downarrow g & = & (l_1 \downarrow g) + (l_2 \downarrow g) & \\ (l_1 \doteq l_2) \downarrow g & = & (l_1 \downarrow g) \doteq (l_2 \downarrow g) & \end{array}$$

For a local assertion p we define the global assertion $p \downarrow g$ as follows:

$$\begin{array}{lcll} (\neg p) \downarrow g & = & (\neg p \downarrow g) & \\ (p_1 \wedge p_2) \downarrow g & = & (p_1 \downarrow g) \wedge (p_2 \downarrow g) & \\ (\exists z\, p) \downarrow g & = & \exists z(p \downarrow g) & z \notin LVar(g) \end{array}$$

As an example, note that $(\mathsf{b} \not\doteq \mathsf{nil} \rightarrow \forall i(2 \leq i \leq \mathsf{m} \rightarrow i \not| \mathsf{b})) \downarrow p$ is equal to $p.\mathsf{b} \not\doteq \mathsf{nil} \rightarrow \forall i(2 \leq i \leq p.\mathsf{m} \rightarrow i \not| p.\mathsf{b})$ (here $p \in LVar_{\mathsf{P}}$).

3.3. Correctness Formulas

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

Definition 3.6. We define a *local correctness formula* to be of the following form:

$$\{p\}S\{q\}$$

where $p, q \in LAss^c$ and $S \in STat^c$, for some c. Here the assertion p is called the *precondition* and the assertion q is called the *postcondition.* The meaning of such a correctness formula is described informally as follows:

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

As said before, reasoning about the local correctness of an object will be done relative to assumptions concerning those parts of its local process that depend on the environment. These parts are called *bracketed sections:*

Definition 3.7. A bracketed section is a construct of the form $\langle S_1; S_2\rangle$, where S_1 denotes an I/O statement or a new-statement, and in S_2 neither I/O statements nor new-statements occur (note that S_2 can be composed of several statements by means of sequential compositions, conditionals, or loops).

Next we define intermediate correctness formulas, which describe the behaviour of an object executing a bracketed section containing a new-statement or a communication between two objects in terms of its effects on a complete system of objects.

Definition 3.8. An *intermediate correctness formula* can have one of the following two forms:

- $\{P\}(z, S)\{Q\}$, where $\langle S\rangle$ is a bracketed section containing a new-statement, with $S \in STat^c$, and $z \in LVar_c$, for some c.
- $\{P\}(z_1, S_1) \parallel (z_2, S_2)\{Q\}$, where, for some types c and c', $z_1 \in LVar_c$ and $z_2 \in LVar_{c'}$ are distinct logical variables and $\langle S_1\rangle$ ($S_1 \in STat^c$) and $\langle S_2\rangle$ ($S_2 \in STat^{c'}$) are bracketed sections that contain I/O statements.

The logical variables z, z_1, and z_2 in the above constructs denote the objects that are considered to be executing the corresponding statements. More precisely, the meaning of the intermediate correctness formula $\{P\}(z, S)\{Q\}$ is as follows:

Every terminating execution of the bracketed section S by the object denoted by the logical variable z starting in a (global) state satisfying P ends in a (global) state satisfying Q.

The second form of intermediate correctness formula, $\{P\}(z_1, S_1) \parallel (z_2, S_2)\{Q\}$, has the following meaning:

Every terminating parallel execution of the bracketed section S_1 by the object denoted by the logical variable z_1 and of S_2 by the object denoted by z_2 starting in a (global) state satisfying P will end in a (global) state satisfying Q.

Finally, we have global correctness formulas, which describe a complete system:

Definition 3.9. A global correctness formula is of the form:

$$\{p\}\rho\{Q\}$$

Here p describes the local state of the root object. Initially the 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 an arbitrary global assertion. We assume that the types of the variables occurring in both p and Q are among the standard types Int and Bool, and the classes defined by ρ. The meaning of the global correctness formula $\{p\}\rho\{Q\}$ can be rendered as follows:

If the execution of the program ρ starts with a root object that satisfies the local assertion p, and if moreover this execution terminates, then the final state will satisfy the global assertion Q.

4. The Proof System

The proof system we present consists of three levels. The first level, called the *local* proof system, allows us to reason about the correctness of a single object. Testing the assumptions that are introduced at this first level to deal with I/O statements and new-statements, is done at the second level, which is called the *intermediate* proof system. The third level, the *global* proof system, formalises the reasoning about a complete system.

4.1. The Local Proof System

The proof system for local correctness formulas is similar to the usual system for sequential programs.

Definition 4.1. The local proof system consists of the following axiom and rules:

Assignment:

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

Sequential composition:

$$\frac{\{p\}S_1\{r\}, \quad \{r\}S_2\{q\}}{\{p\}S_1;S_2\{q\}} \tag{LSC}$$

Conditional:

$$\frac{\{p \wedge e\}S_1\{q\}, \quad \{p \wedge \neg e\}S_2\{q\}}{\{p\}\text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}\{q\}} \tag{LCOND}$$

Iteration:

$$\frac{\{p \wedge e\}S\{p\}}{\{p\}\text{while } e \text{ do } S \text{ od}\{p \wedge \neg e\}} \tag{LIT}$$

Invariance:

$$\{p\}S\{p\} \tag{INV}$$

provided that $IVar(p) \cap IVar(S) = \emptyset$.

Substitution:

$$\frac{\{p\}S\{q\}}{p[l/z]\}S\{q[l/z]\}} \tag{SUB}$$

provided that if $z \in LVar(q)$ then $IVar(l) \cap IVar(S) = \emptyset$, where z is a logical variable of the same type as the local expression l.

Conjunction:

$$\frac{\{p_1\}S\{q_1\}, \quad \{p_2\}S\{q_2\}}{\{p_1 \wedge p_2\}S\{q_1 \wedge q_2\}} \tag{CON}$$

Consequence:

$$\frac{p \to p_1, \quad \{p_1\}S\{q_1\}, \quad q_1 \to q}{\{p\}S\{q\}} \tag{LCR}$$

The substitution operation $[e/x]$ occurring in the above assignment axiom is the ordinary substitution, i.e., literal replacement of every occurrence of the variable x by the expression e. This works because at this level we have no aliasing, i.e., it is not possible that different local expressions denote the same variable. In the intermediate and global proof systems we shall have to take special measures to deal with aliasing.

4.2. The Intermediate Proof System

In this section we present the proof system for intermediate correctness formulas.

4.2.1. The Assignment Axiom

Definition 4.2. The assignment axiom in the intermediate proof system has the form

$$\{P[e \downarrow z/z.x]\}(z, x := e)\{P\} \tag{IASS}$$

First note that we have to transform the expression e to the global expression $e \downarrow z$ and substitute this latter expression for $z.x$ because we consider the execution of the assignment $x := e$ by the object denoted by z. Furthermore we have to pay special attention to the substitution $[e \downarrow z/z.x]$ because the usual substitution does not take into account that there are many different global expressions that may denote the same variable $z.x$. This problem is solved by the following definition.

Definition 4.3. Given a global expression g, an instance variable x of the same type, and a logical variable $z \in \bigcup_c LVar_c$, we define for any global expression g' the substitution $g'[g/z.x]$ by induction on the complexity of g' as follows.

$$\begin{array}{lcll} g'[g/z.x] & = & g' & \text{if } g' = z', n, \textsf{nil}, \textsf{self}, \textsf{true}, \textsf{false} \\ (g'.y)[g/z.x] & = & g'[g/z.x].y & \text{if } y \neq x \\ (g'.x)[g/z.x] & = & \multicolumn{2}{l}{\textsf{if } g'[g/z.x] \doteq z \textsf{ then } g \textsf{ else } g'[g/z.x].x \textsf{ fi}} \\ & \vdots & & \\ (g_1 \doteq g_2)[g/z.x] & = & \multicolumn{2}{l}{(g_1[g/z.x]) \doteq (g_2[g/z.x])} \end{array}$$

The omitted cases are defined directly from the application of the substitution to the subexpressions, like the last one. This substitution operation is generalised to global assertions in a straightforward manner, with the notation $P[g/z.x]$.

As an example, consider the postcondition $\forall i(1 \leq i \leq |s| \to (s : i).x \doteq i)$ for the statement $x := y + 1$, executed by the object denoted by z. The precondition given by the axiom (IASS) is

$$\forall i(1 \leq i \leq |s| \to \textsf{if } s : i \doteq z \textsf{ then } z.y + 1 \textsf{ else } (s : i).x \textsf{ fi} \doteq i).$$

The best way to justify definition 4.3. is by comparing it to the ordinary substitution $[e/x]$, which is used in the local assignment axiom (LASS). The essential property of this substitution is that the substituted expression, evaluated

in the state before the assignment, has the same value as the original expression in the state after the assignment. Quasi-formally, we could write this as

$$[\![l[e/x]]\!](\sigma) = [\![l]\!](\sigma')$$

where σ' is the state that results from executing the assignment $x := e$ in the state σ. We could say that the substitution is a way of predicting the value that an expression or assertion will have after performing an assignment.

It is easy to prove that the substitution operation defined in definition 4.3 has exactly the same property:

$$[\![g'[g/z.x]]\!](\sigma) = [\![g']\!](\sigma')$$

and

$$\sigma \models P[g/z.x] \iff \sigma' \models P$$

where σ' is the state that results from σ by changing the value of the variable x in the object denoted by z to the value $[\![g]\!](\sigma)$ that results from evaluating the expression g in the state σ.

The most important aspect of this substitution is certainly the conditional expression that turns up when we are dealing with an expression of the form $g'.x$. This is necessary because a certain form of aliasing is possible: After the assignment it may be the case that g' refers to the same object as the logical variable z, so that $g'.x$ is the same variable as $z.x$, which has the value $[\![g]\!](\sigma)$. It is also possible that, after the assignment, g' does not refer to the object denoted by z, so that the value of $g'.x$ does not change. Since we can not decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides dynamically. It is instructive to note that a similar form of aliasing arises in the case of substitution for array variables.

The notation [./.] may seem overloaded now, but it is always possible to determine the required operation from the form of the arguments.

4.2.2. The Creation of New Objects

Definition 4.4. We describe the new-statement by the following axiom of the intermediate proof system:

$$\{P[z'/z.x][\mathsf{new}/z']\}(z, x := \mathsf{new})\{P\} \qquad \text{(NEW)}$$

where z' does not occur in P. Here $[\mathsf{new}/z']$ denotes another special substitution operation which will be explained below.

This axiom reflects the fact that we can view the execution of the statement $x := \mathsf{new}$ as consisting of two steps: first the new object is created and temporarily stored in the logical variable z', and then the value of this logical variable z' is assigned to the instance variable x of the object denoted by z. The second step is dealt with by the substitution $[z'/z.x]$ of definition 4.3, which takes care of all possible complications of aliasing.

For the creation of a new object we have to define another substitution operation $[\mathsf{new}/z]$. This is complicated by the fact that the newly created object does not exist in the state just before its creation, so that in this state we can not refer to it. Fortunately, we do need this substitution for all possible global expressions, but primarily for assertions, and in an assertion the logical variable z can essentially occur in only two contexts: either one of its instance variables is

referenced, or it is compared for equality with another expression. In both cases we can predict the outcome without having to refer to the new object.

Definition 4.5. Let $z \in LVar_c$. For certain global expressions g we define $g[\mathsf{new}/z]$ by induction on the complexity of g. We only list the interesting cases.

$$
\begin{array}{lcll}
z'[\mathsf{new}/z] & = & z' & \text{if } z' \neq z \\
z[\mathsf{new}/z] & & \text{is undefined} & \\
g[\mathsf{new}/z] & = & g & \text{if } g = n, \mathsf{nil}, \mathsf{self}, \mathsf{true}, \mathsf{false} \\
(z'.x)[\mathsf{new}/z] & = & z'.x & \text{if } z' \neq z \\
(z.x)[\mathsf{new}/z] & = & \mathsf{nil} & \\
(g.y.x)[\mathsf{new}/z] & = & (g.y)[\mathsf{new}/z].x & \\
(g_1 \doteq g_2)[\mathsf{new}/z] & = & g_1[\mathsf{new}/z] \doteq g_2[\mathsf{new}/z] & \text{if } g_1, g_2 \neq z, \mathsf{if} \ \ldots \ \mathsf{fi} \\
(g_1 \doteq z)[\mathsf{new}/z] & = & \mathsf{false} & \text{if } g_1 \neq z, \mathsf{if} \ \ldots \ \mathsf{fi} \\
(z \doteq g_2)[\mathsf{new}/z] & = & \mathsf{false} & \text{if } g_2 \neq z, \mathsf{if} \ \ldots \ \mathsf{fi} \\
(z \doteq z)[\mathsf{new}/z] & = & \mathsf{true} &
\end{array}
$$

Here we have ignored the conditional expressions (these can be removed before substituting). In all the other cases not listed above, the substitution $[\mathsf{new}/z]$ can be applied to an expression by applying it to the constituent expressions. In this way $g[\mathsf{new}/z]$ is defined for all global expressions g except for z itself (and for certain conditional expressions). Again it is rather easy to prove that this substitution has the desired property: We have for global expressions g for which $g[\mathsf{new}/z]$ is defined

$$[\![g[\mathsf{new}/z]]\!](\sigma) = [\![g]\!](\sigma')$$

where σ' can be obtained from σ by creating a new object (with all its variables initialised to nil) and storing it in the variable z.

Definition 4.6. We define $P[\mathsf{new}/z]$ by induction on the complexity of the global assertion P, assuming $z \in LVar_c$, for some c.

$$
\begin{array}{lcl}
g[\mathsf{new}/z] & & \text{as in definition 4.5} \\
(\neg P)[\mathsf{new}/z] & = & \neg(P[\mathsf{new}/z]) \\
(P_1 \wedge P_2)[\mathsf{new}/z] & = & (P_1[\mathsf{new}/z] \wedge P_2[\mathsf{new}/z]) \\
(\exists z' P)[\mathsf{new}/z] & = & \exists z'(P[\mathsf{new}/z]) \qquad \text{if } z' \in LVar_a,\ a \neq c, c^* \\
(\exists z' P)[\mathsf{new}/z] & = & \exists z'(P[\mathsf{new}/z]) \vee (P[z/z'][\mathsf{new}/z]) \\
& & \qquad\qquad\qquad\qquad \text{if } z' \in LVar_c\ (z' \neq z) \\
(\exists z' P)[\mathsf{new}/z] & = & \exists z' \exists z''(|z'| \doteq |z''| \wedge P[z'', z/z'][\mathsf{new}/z]) \\
& & \qquad\qquad\qquad\qquad \text{if } z' \in LVar_{c^*},\ z'' \in LVar_{\mathsf{Bool}}.
\end{array}
$$

Here we assume that z'' does not occur in P. The substitution $[z'', z/z']$ will be defined in definition 4.7. It is important to note that for all boolean expressions g the expression $g[\mathsf{new}/z]$ is defined.

The case of quantification over the type c of the newly created object can be explained as follows: Remember that we interpret the result of the substitution in a state in which the object denoted by z does not yet exist. In the first part $\exists z'(P[\mathsf{new}/z])$ of the substituted formula the bound variable z' thus ranges

over all the old objects. In the second part the object to be created is dealt with separately. This is done by first substituting z for z' (expressing that the quantified variable z' takes the same value as the variable z) and then applying the substitution $[\mathsf{new}/z]$ (note that simply applying $[\mathsf{new}/z']$ does not give the right result in the case that z occurs in P). Together the two parts of the substituted formula express quantification over the whole range of existing objects in the new state.

Let us consider, for example, the statement $x := \mathsf{new}$, to be executed by the object indicated by the logical variable z, and the given postcondition

$$\forall v(v.x \not\doteq \mathsf{nil} \rightarrow v.y \doteq 1)$$

In order to determine the corresponding precondition given by the axiom (NEW), we first apply the substitution $[z'/z.x]$, which leads to

$$\forall v(\mathsf{if}\ v \doteq z\ \mathsf{then}\ z'\ \mathsf{else}\ v.x\ \mathsf{fi} \not\doteq \mathsf{nil} \rightarrow v.y \doteq 1)$$

We can remove the conditional expression by taking the equivalent assertion

$$\forall v((v \doteq z \wedge z' \not\doteq \mathsf{nil}) \vee (v \not\doteq z \wedge v.x \not\doteq \mathsf{nil}) \rightarrow v.y \doteq 1)$$

Now to this we apply the substitution $[\mathsf{new}/z']$ (note that we use here the $\forall$-version of the $\exists$-case of definition 4.6.), resulting in

$$\forall v((v \doteq z \wedge \mathsf{true}) \vee (v \not\doteq z \wedge v.x \not\doteq \mathsf{nil}) \rightarrow v.y \doteq 1)$$
$$\wedge ((\mathsf{false} \wedge \mathsf{true}) \vee (\mathsf{true} \wedge \mathsf{nil} \not\doteq \mathsf{nil}) \rightarrow \mathsf{nil} \doteq 1)$$

which can be simplified to

$$\forall v(v \doteq z \vee v.x \not\doteq \mathsf{nil} \rightarrow v.y \doteq 1)$$

It is perhaps instructive to go through the application of the substitution $[\mathsf{new}/z']$ to the expression $z' \not\doteq \mathsf{nil}$ in some more detail:

$$\begin{aligned} &(z' \not\doteq \mathsf{nil})[\mathsf{new}/z'] &&= \\ &\neg((z' \doteq \mathsf{nil})[\mathsf{new}/z']) &&= \\ &\neg\mathsf{false} &&= \\ &\mathsf{true} \end{aligned}$$

So the application of a substitution $[\mathsf{new}/z']$ applied to an assertion of the form $z' \doteq \mathsf{nil}$ consists of a *static evaluation* of the assertion: Since a substitution $[\mathsf{new}/z']$ interprets z' as the new object we know a priori that $z' \doteq \mathsf{nil}$ will evaluate to false.

For quantification about sequences of objects of class c, we need a slightly more elaborate mechanism. The sequences over which we quantify in the old state cannot contain the new object as an element. Therefore we use two sequence variables $z' \in LVar_{c^*}$ and $z'' \in LVar_{\mathsf{Bool}^*}$ to code one sequence of objects in the new state. The idea of the substitution operation $[z'', z/z']$ is that at the places where z'' yields true, the value of the coded sequence is the newly created object, here denoted by the variable z. Where z'' yields false, the value of the coded sequence is the same as the value of z' and where z'' delivers nil the sequence also yields nil. This is formalised in the following definition.

Definition 4.7. For certain global expressions g we define $g[z'',z/z']$ as follows (again we list only the interesting cases):

$$
\begin{array}{lcl}
z'[z'',z/z'] & & \text{is undefined} \\
\bar{z}[z'',z/z'] & = & \bar{z} \quad \text{if } \bar{z} \neq z' \\
g[z'',z/z'] & = & g \quad \text{if } g = n, \mathsf{nil}, \mathsf{self}, \mathsf{true}, \mathsf{false} \\
(g.x)[z'',z/z'] & = & g[z'',z/z'].x \\
(z' : g)[z'',z/z'] & = & \mathsf{if}\ z'' : (g[z'',z/z']) \\
 & & \mathsf{then}\ z \\
 & & \mathsf{else}\ z' : (g[z'',z/z']) \\
 & & \mathsf{fi} \\
(g_1 : g_2)[z'',z/z'] & = & g_1[z'',z/z'] : g_2[z'',z/z'] \qquad \text{if } g_1 \neq z' \\
(|z'|)[z'',z/z'] & = & |z'| \\
(|g|)[z'',z/z'] & = & |g[z'',z/z']| \quad \text{if } g \neq z'
\end{array}
$$

The generalization of the above to other global expressions and to global assertions is straightforward.

Again we have the desired property:

$$\sigma \models P[\mathsf{new}/z] \iff \sigma' \models P$$

where σ' can be obtained from σ by creating a new object (with all its variables initialised to nil) and storing it in the variable z.

We illustrate the last definition by another example of the use of axiom (NEW). Again we take the statement $x := \mathsf{new}$ executed by the object in z, but this time the postcondition is $\exists s \forall p \exists i (s : i \doteq p)$. Applying the substitution $[z'/z.x]$ leaves this assertion unchanged, so we can directly apply the substitution $[\mathsf{new}/z']$. We calculate in the following few steps (here $b \in LVar_{\mathsf{Bool}^*}$):

$$
\begin{array}{l}
\Big(\exists s \forall p \exists i (s : i \doteq p)\Big)[\mathsf{new}/z'] \\
= \exists s \exists b_{\mathsf{Bool}^*}\ |s| \doteq |b| \wedge \Big(\forall p \exists i (s : i \doteq p)\Big)[b, z'/s][\mathsf{new}/z'] \\
= \exists s \exists b\ |s| \doteq |b| \wedge \Big(\forall p \exists i (\mathsf{if}\ b : i\ \mathsf{then}\ z'\ \mathsf{else}\ s : i\ \mathsf{fi} \doteq p)\Big)[\mathsf{new}/z'] \\
\equiv \exists s \exists b\ |s| \doteq |b| \wedge \Big(\forall p \exists i (b : i \wedge z' \doteq p) \vee (\neg b : i \wedge s : i \doteq p)\Big)[\mathsf{new}/z'] \\
= \exists s \exists b\ |s| \doteq |b| \wedge \forall p \exists i \Big((b : i \wedge \mathsf{false}) \vee (\neg b : i \wedge s : i \doteq p)\Big) \\
\qquad\qquad\qquad\quad \wedge \exists i (b : i \wedge \mathsf{true}) \vee (\neg b : i \wedge \mathsf{false}) \\
\equiv \exists s \exists b \Big(|s| \doteq |b| \wedge \forall p \exists i (\neg b : i \wedge s : i \doteq p) \wedge \exists i (b : i)\Big)
\end{array}
$$

Here $\equiv$ denotes semantic equivalence, which means that the assertions on both sides will have the same truth value in every environment. (By the way, in any state that can occur in the execution of a P program, the above assertions will be true, since there is only a finite number of objects of any class.)

4.2.3. Communication

Now we define an axiom and some rules which together describe the communication between objects.

Definition 4.8. Let $z_1 \in LVar_c$ and $z_2 \in LVar_{c'}$ be two distinct variables. Furthermore let $S_1 \in STat^c$ be of the form $x?y$ or $?y$ and $S_2 = x'!e \in STat^{c'}$ such that $x \in IVar_{c'}$, $x' \in IVar_c$, and the variable y is of the same type as the expression e. Such a pair of I/O statements are said to *match*. The following is the communication axiom:

$$\{P[e \downarrow z_2/z_1.y]\}(z_1, S_1) \parallel (z_2, S_2)\{P\} \qquad \text{(COMM)}$$

Note that the communication is described by a substitution that expresses the assignment of $e \downarrow z_2$ to $z_1.y$ (definition 4.3.).

The following two rules show how one can use the information about the relationship between the receiver and the sender which must hold for the communication to take place, that is, the sender must refer to the receiver, and if the receiver executes an input statement of the form $x?y$, it must refer to the sender.

Definition 4.9. Let $z \in LVar_c$ and $z' \in LVar_{c'}$ be two distinct variables. Furthermore let $x?y, ?y \in STat^c$ and $x'!e \in STat^{c'}$ be such that both input statements match with the output statement. Then we have the following two rules:

$$\frac{\{P \wedge z.x \doteq z' \wedge z' \not\doteq \mathsf{nil} \wedge z'.x' \doteq z \wedge z \not\doteq \mathsf{nil} \wedge R\}(z, x?y) \parallel (z', x'!e)\{Q\}}{\{P\}(z, x?y) \parallel (z', x'!e)\{Q\}} \qquad \text{(SR1)}$$

and

$$\frac{\{P \wedge z'.x' \doteq z \wedge z \not\doteq \mathsf{nil} \wedge R\}(z, ?y) \parallel (z', x'!e)\{Q\}}{\{P\}(z, ?y) \parallel (z', x'!e)\{Q\}} \qquad \text{(SR2)}$$

where $R = (z \not\doteq z')$ if $c = c'$ and $R = \mathsf{true}$ otherwise.

The following rule describes the independent parallel execution of two objects.

Definition 4.10. Suppose that $S_1 \in STat^c$ and $S_2 \in STat^{c'}$ do not contain any I/O or new-statements. Furthermore let $z \in LVar_c$ and $z' \in LVar_{c'}$ be two distinct variables. Then we have the following rule:

$$\frac{\{P\}(z, S_1)\{R\}, \qquad \{R\}(z', S_2)\{Q\}}{\{P\}(z, S_1) \parallel (z', S_2)\{Q\}} \qquad \text{(PAR1)}$$

Note that this rule models the fact that the parallel execution of two local computations can be sequentialised. The next rule takes care of the case where the two bracketed sections do contain I/O statements.

Definition 4.11. Let $\langle S_1; S_2\rangle \in STat^c$ and $\langle S_1'; S_2'\rangle \in STat^{c'}$ be two bracketed sections containing I/O statements. Furthermore let $z \in LVar_c$ and $z' \in LVar_{c'}$ be two distinct variables. Then we have the rule

$$\frac{\{P\}(z, S_1) \parallel (z', S_1')\{R\}, \quad \{R\}(z, S_2) \parallel (z', S_2')\{Q\}}{\{P\}(z, S_1; S_2) \parallel (z', S_1'; S_2')\{Q\}} \qquad \text{(PAR2)}$$

Finally, the intermediate proof system contains rules corresponding to the rules for sequential composition, the conditional statement, the iterative construct, the consequence rule, and the substitution rule of the local proof system. These are straightforward modifications, so we only give the iteration rule as an example.

Definition 4.12. Let $\mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{od} \in STat^c$ and $z \in LVar_c$. Then we have the following rule:

$$\frac{\{P \wedge e \downarrow z\}(z, S)\{Q\}}{\{P\}(z, \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{od})\{P \wedge \neg e \downarrow z\}} \qquad \text{(IIT)}$$

It is worthwhile to note that the axiom corresponding to the invariance axiom is derivable in the intermediate proof system. Furthermore we note that with respect to the substitution rule we have to require that the substituted variable is different from the variable z which represents the executing object.

4.3. The Global Proof System

In this section we describe the global proof system. Two bracketed sections containing I/O statements are said to *match* if the corresponding I/O statements match. A program ρ is said to be *bracketed* if every I/O statement and new-statement occurs in a bracketed section. An *assumption* is defined to be a local correctness formula about a bracketed section. Given a set A of assumptions, $A \vdash \{p\}S\{q\}$ denotes the derivability of the local correctness formula $\{p\}S\{q\}$ from the local proof system using the assumptions of A as additional axioms.

It is interesting to note the similarity of the reasoning about I/O statements and new statements in terms of assumptions with the reasoning about *recursive procedures*. Let X be a procedure variable declared as the sequential program S. In [Apt81] we find the following rule for recursive procedures without parameters:

$$\frac{\{p\}X\{q\} \vdash \{p\}S\{q\}}{\{p\}X\{q\}}$$

The idea is to prove the correctness of the procedure call X by *assuming* that it is correct and then prove that the body (S) of X satisfies the same specification. Once such a proof has been given we are allowed to conclude the correctness of the call without assumptions. Now one of the main proof-theoretical difference between I/O statements and new statements, on the one hand, and recursive procedures, on the other hand, is that the correctness of an I/O statement (or new-statement) can not be inferred from the correctness of the statement in which it occurs: The correctness of an I/O statement (or new-statement) can only be inferred from the correctness of the other components of the program. This is why the assumptions about I/O statements and new-statements are discharged on a different level of the proof system. This intermediate level between local reasoning and reasoning about a complete system consists of what is known as the *cooperation test.*

Definition 4.13. Let $\rho = \langle c_1 \leftarrow S_1, \ldots, c_n \leftarrow S_n \rangle$ be bracketed. Furthermore, for each i, $1 \leq i \leq n$, let A_i denote a set of local correctness formulas about the bracketed sections occurring in S_i such that $A_i \vdash \{p_i\}S_i\{q_i\}$. Finally, let I be some global assertion, which we shall call the *global invariant*. Now we say that the proofs $A_i \vdash \{p_i\}S_i\{q_i\}$ cooperate with respect to the global invariant I, notation $Coop(A_1, \ldots, A_n, I)$, if the following conditions are fulfilled:

1. The invariant I does not contain any instance variable that occurs at the left-hand side of an assignment outside a bracketed section.
2. Let $\langle R \rangle$ and $\langle R' \rangle$ be two matching bracketed sections such that $\{p\}R\{q\} \in A_i$ $(1 \leq i < n)$ and $\{p'\}R'\{q'\} \in A_j$ $(1 \leq j \leq n)$. Furthermore let $z \in LVar_{c_i}$ and $z' \in LVar_{c_j}$ be two new distinct variables. Then we require

$$\vdash \{I \wedge p \downarrow z \wedge p' \downarrow z'\}(z, R) \parallel (z', R')\{I \wedge q \downarrow z \wedge q' \downarrow z'\}$$

3. Let $\langle R \rangle$ be a bracketed section containing the new-statement $x :=$ new, with

$x \in IVar_{c_j}$, such that $\{p\}R\{q\} \in A_i$. Furthermore let $z \in LVar_{c_i}$ be a new variable. Then we require that

$$\vdash \{I \wedge p \downarrow z\}(z, R)\{I \wedge q \downarrow z\}$$

and

$$\models \bigwedge_{y \in IVar(S_j)} (y \doteq \mathsf{nil}) \rightarrow p_j$$

4. The following assertion should hold:

$$\exists z(p_n \downarrow z \wedge \forall z'(z' \doteq z) \wedge \bigwedge_{1 \le i < n} (\forall z_i \, \mathsf{false})) \rightarrow I$$

Here $z_i \in LVar_{c_i}$ for $1 \le i < n$, and $z, z' \in LVar_{c_n}$ $(z' \neq z)$.

The syntactic restriction in clause 1 on occurrences of variables in the global invariant I implies the invariance of this assertion over those parts of the program that are not contained in a bracketed section. Clauses 2 and 3 imply, among others, the invariance of the global invariant over the bracketed sections.

This global invariant expresses some invariant properties of the global states arising during a computation of ρ. These properties are invariant in the sense that they hold whenever the program counter of every existing object is at a location outside a bracketed section. The above method to prove the invariance of the global invariant is based on the following semantical property of bracketed sections: Every computation of ρ can be rearranged such that at every time there is at most one object executing a bracketed section containing a new-statement, and an object is allowed to enter a bracketed section containing a I/O statement only if there is at most one other object executing a bracketed section.

Clause 2 establishes the cooperation between two arbitrary matching assumptions, where two assumptions are said to match if their corresponding bracketed sections contain matching I/O statements. Note that since there exists only one object of class c_n, the root-object, we do not have to apply the cooperation test between two matching assumptions of the set A_n.

Clause 3 discharges assumptions about bracketed sections containing new-statements. Additionally the truth of the precondition of the local process of the new object is established.

Clause 4 establishes the truth of the global invariant in the initial state. Note that the assertion $\forall z \, \mathsf{false}$ expresses that there exist no objects of class c, assuming $z \in LVar_c$. The assertion $\forall z'(z' \doteq z)$ expresses that there exists precisely one object of class c_n.

In the following definitions, let $\rho = \langle c_1 \leftarrow S_1, \ldots, c_n \leftarrow S_n \rangle$.

Definition 4.14. The program rule of the global proof system has the following form:

$$\frac{Coop(A_1, \ldots, A_n, I) \qquad A_i \vdash \{p_i\}S_i\{q_i\}, 1 \le i \le n}{\{p_n\}\rho\{I \wedge \bigwedge_{1 \le i \le n} \forall z_i \, (q_i \downarrow z_i)\}} \tag{PR}$$

Note that in the conclusion of the program rule (PR) we take as precondition the precondition of the local process of the root object because initially only this object exists. The postcondition consists of a conjunction of the global invariant and the assertions $\forall z_i \, (q_i \downarrow z_i)$ $(z_i \in LVar_{c_i})$, which express that the final local state of every object of class c_i is characterised by the local assertion q_i.

Definition 4.15. We have the following consequence rule for programs:

$$\frac{p \to p', \quad \{p'\}\rho\{Q'\}, \quad Q' \to Q}{\{p\}\rho\{Q\}} \tag{PCR}$$

Definition 4.16. Furthermore, we have a rule for *auxiliary variables*: For a given program ρ and a (global) postcondition Q, let Aux be a set of instance variables, which occur only in assignments, such that

- for any assignment $x := e$ occurring in ρ we have that $IVar(e) \cap Aux \neq \emptyset$ implies that $x \in Aux$,
- the variables of the set Aux do not occur as tests in conditionals, loops, nor do they occur in I/O statements or new-statements of ρ,
- $IVar(Q) \cap Aux = \emptyset$.

Let ρ' be the program that can be obtained from ρ by deleting all assignments to variables belonging to the set Aux. Then we have the following rule:

$$\frac{\{P\}\rho\{Q\}}{\{P\}\rho'\{Q\}} \tag{AUX}$$

The rule for auxiliary variables can be explained as follows: To be able to prove some properties of a program ρ' it may be necessary to add a number of extra variables and statements to do some bookkeeping. If these additions satisfy the above conditions, we are sure that they do not influence the flow of control of the program, and therefore they can be deleted after the proof of the enlarged program ρ is completed.

Definition 4.17. Next we have a substitution rule to remove instance variables from preconditions:

$$\frac{\{p\}\rho\{Q\}}{\{p[l/x]\}\rho\{Q\}} \tag{S}$$

provided the instance variable x does not occur in ρ or Q. In practice, this rule is mainly used for auxiliary variables.

5. An Example Proof

As an illustration of the proof system we shall now formally prove a property of the program listed at the end of section 2. We want to prove that exactly the primes up to n are generated. This amounts to proving the postcondition

$$\forall i(Prime(i) \wedge i \leq n \leftrightarrow \exists p\, p.\mathsf{m} \doteq i) \tag{5.1}$$

Here i and p are logical variables ranging over integers and objects of class P, respectively. The predicate *Prime* holds for an integer if and only if it is a (positive) prime number.

To establish this postcondition we first introduce an auxiliary variable v (for 'valid') of type Bool in class P, which indicates that the value stored in the variable b is valid. The variable v is false precisely from the moment that the object sends the value of its b variable to its neighbour until the moment that it receives a new value for the variable b. Now we take a global invariant I which is the conjunction of the following assertions:

I_1: $\forall p(p.\mathsf{m} \neq \mathsf{nil} \to Prime(p.\mathsf{m}))$

I_2: $\forall p \forall p'(p.\mathsf{m} \neq \mathsf{nil} \to (p'.\mathsf{m} \doteq nextpr(p.\mathsf{m}) \to p.\mathsf{l} \doteq p'))$

I_3: $\forall p \forall p'(p.\mathsf{b} < p'.\mathsf{b} \to p.\mathsf{m} > p'.\mathsf{m})$

I_4: $\forall p \forall p'(p.\mathsf{m} \doteq p'.\mathsf{m} \to p \doteq p')$

I_5: $\forall g \forall i(Prime(i) \wedge i < g.\mathsf{c} \to \exists p(p.\mathsf{m} \doteq i \vee p.\mathsf{b} \doteq i))$

I_6: $\forall g \forall p((p.\mathsf{m} \neq \mathsf{nil} \to 2 \leq p.\mathsf{m} < g.\mathsf{c}) \wedge (p.\mathsf{b} \neq \mathsf{nil} \to 2 \leq p.\mathsf{b} < g.\mathsf{c})$

I_7: $\forall p \forall p'(p.\mathsf{m} \neq \mathsf{nil} \wedge p'.\mathsf{b} \neq \mathsf{nil} \wedge p'.\mathsf{v} \to p.\mathsf{m} < p'.\mathsf{b})$

I_8: $\forall p(p.\mathsf{m} \neq \mathsf{nil} \wedge p.\mathsf{b} \neq \mathsf{nil} \to p.\mathsf{m} < p.\mathsf{b})$

In the above definition g is a logical variable of type G, so it will always denote the root object. The function *nextpr* applied to an integer gives the next prime number.

The assertion I_2 expresses that successive primes are stored in successive P objects in the chain. Assertion I_3 states that the prime candidates flow through the chain in their natural order. Assertion I_4 states that each prime number is uniquely represented. Next, assertion I_5 states that all the prime numbers among the candidates sent are represented or being processed. On the other hand assertion I_6 essentially expresses that all the numbers which occur in the chain have been sent by the generator. Finally, I_7 and I_8 says that a candidate is always greater than any prime number already found: I_7 states this globally, but only if the variable b containing the candidate is marked as valid by the flag v, and I_8 expresses it locally.

The easiest way to represent the outcome of the local proof system is by means of a *proof outline*. This consists of an expanded version of the program: the statements concerning auxiliary variables are added, the bracketed sections are indicated by angle brackets, and the assertions that play a role in the local proof are inserted at the proper places, surrounded by braces. The proof outline for our example program is given in Fig. 2.

Now we have to check the assumptions of the cooperation test. Here we shall deal with one pair of I/O statements and one new-statement. We first treat the following case:

$$\{I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s\}(r, ?\mathsf{m}) \parallel (s, \mathsf{l}\,!\,\mathsf{b};\ \mathsf{v} := \mathsf{false})\{I \wedge \psi_1 \downarrow r \wedge (\neg\mathsf{v}) \downarrow s\} \quad (5.2)$$

Here s and r are logical variables of type P denoting the sender and the receiver.

We prove (5.2) using the rule (PAR2). The following nontrivial premisses are needed:

$$\{I \wedge \psi_1 \downarrow r\}(s, \mathsf{v} := \mathsf{false})\{I \wedge \psi_1 \downarrow r \wedge (\neg\mathsf{v}) \downarrow s\} \quad (5.3)$$

$$\{I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s\}(r, ?\mathsf{m}) \parallel (s, \mathsf{l}\,!\,\mathsf{b})\{I \wedge \psi_1 \downarrow r\} \quad (5.4)$$

In order to prove (5.3), by the rule (IASS) and the consequence rule, we must show that the postcondition, after applying the substitution [false/s.v], is implied by the precondition. Now it is clear that I_1–I_6, I_8, and $\psi_1 \downarrow r$ are not changed by this substitution, so they are subsumed by the precondition. For the other parts we first calculate as follows:

$$\begin{aligned} ((\neg\mathsf{v}) \downarrow s)[\mathsf{false}/s.\mathsf{v}] &= (\neg s.\mathsf{v})[\mathsf{false}/s.\mathsf{v}] \\ &= \neg\mathsf{if}\ s \doteq s\ \mathsf{then\ false\ else}\ s.\mathsf{v}\ \mathsf{fi} \\ &\equiv \mathsf{true} \end{aligned}$$

$$
\begin{array}{lll}
\langle \mathsf{P} & \leftarrow & \{\psi_0 : \mathsf{l} \doteq \mathsf{nil} \wedge \mathsf{m} \doteq \mathsf{nil} \wedge b \doteq \mathsf{nil}\} \\
& & \langle ?\mathsf{m}\rangle;\ \{\psi_1 : \mathsf{l} \doteq \mathsf{nil} \wedge b \doteq \mathsf{nil}\} \\
& & \mathsf{if\ m} \not\doteq \mathsf{nil} \\
& & \mathsf{then}\ \{\mathsf{l} \doteq \mathsf{nil}\} \\
& & \quad \langle \mathsf{l} := \mathsf{new}\rangle;\ \{\mathsf{true}\} \\
& & \quad \langle ?\mathsf{b};\ \mathsf{v} := \mathsf{true}\rangle; \\
& & \quad \{\psi_2 : \mathsf{v} \wedge (\mathsf{b} \not\doteq \mathsf{nil} \rightarrow \forall i(2 \le i < \mathsf{m} \rightarrow i \nmid \mathsf{b}))\}; \\
& & \quad \mathsf{while\ b} \not\doteq \mathsf{nil} \\
& & \quad \mathsf{do}\ \{\mathsf{v} \wedge \forall i(2 \le i < \mathsf{m} \rightarrow i \nmid \mathsf{b})\} \\
& & \quad\quad \mathsf{if\ m} \nmid \mathsf{b} \\
& & \quad\quad \mathsf{then}\ \{\psi_3 : \mathsf{v} \wedge \forall i(2 \le i \le \mathsf{m} \rightarrow i \nmid \mathsf{b})\} \\
& & \quad\quad\quad \langle \mathsf{l}\,!\,\mathsf{b};\ \mathsf{v} := \mathsf{false}\rangle \\
& & \quad\quad \mathsf{fi};\ \{\neg \mathsf{v}\} \\
& & \quad\quad \langle ?\mathsf{b};\ \mathsf{v} := \mathsf{true}\rangle\ \{\psi_2\} \\
& & \quad \mathsf{od};\ \{\mathsf{b} \doteq \mathsf{nil}\} \\
& & \quad \langle \mathsf{l}\,!\,\mathsf{b}\rangle\ \{\mathsf{b} \doteq \mathsf{nil}\} \\
& & \mathsf{fi}\ \{\mathsf{b} \doteq \mathsf{nil}\} \\
& : & \{\mathsf{true}\} \\
& & \langle \mathsf{f} := \mathsf{new};\ \mathsf{c} := 2\rangle;\ \{2 \le \mathsf{c} \le \max(2, n+1)\} \\
& & \mathsf{while\ c} \le n \\
& & \mathsf{do}\ \{2 \le \mathsf{c} \le n\} \\
& & \quad \langle \mathsf{f}\,!\,\mathsf{c};\ \mathsf{c} := \mathsf{c} + 1\rangle \\
& & \mathsf{od};\ \{\mathsf{c} \doteq \max(2, n+1)\} \\
& & \langle \mathsf{f}\,!\,\mathsf{nil}\rangle \\
& & \{\mathsf{c} \doteq \max(2, n+1)\}\}\rangle
\end{array}
$$

Fig. 2. The proof outline for our example program.

Finally, we observe that $I_7[\mathsf{false}/s.\mathsf{v}]$ is the formula

$$\forall p \forall p'(p.\mathsf{m} \not\doteq \mathsf{nil} \wedge p'.\mathsf{b} \not\doteq \mathsf{nil} \wedge \mathsf{if}\ p' \doteq s\ \mathsf{then\ false\ else}\ p'.\mathsf{v}\ \mathsf{fi} \rightarrow p.\mathsf{m} < p'.\mathsf{b})$$

and this is clearly implied by I_7.

By the rule (SR2) and the axiom (COMM) the proof of the second premiss (5.4) amounts to establishing the truth of the formula

$$I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s \wedge s.\mathsf{l} \doteq r \rightarrow (I \wedge \psi_1 \downarrow r)[s.\mathsf{b}/r.\mathsf{m}].$$

Here we only show the following part of this:

$$I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s \wedge s.\mathsf{l} \doteq r \rightarrow I_1[s.\mathsf{b}/r.\mathsf{m}].$$

Now $I_1[s.\mathsf{b}/r.\mathsf{m}]$ is the formula

$$\forall p(\mathsf{if}\ p \doteq r\ \mathsf{then}\ s.\mathsf{b}\ \mathsf{else}\ p.\mathsf{m}\ \mathsf{fi} \not\doteq \mathsf{nil} \rightarrow \mathit{Prime}(\mathsf{if}\ p \doteq r\ \mathsf{then}\ s.\mathsf{b}\ \mathsf{else}\ p.\mathsf{m}\ \mathsf{fi})),$$

which is equivalent to

$$\forall p((p \neq r \to (p.\mathsf{m} \neq \mathsf{nil} \to Prime(p.\mathsf{m}))) \land (p \doteq r \to (s.\mathsf{b} \neq \mathsf{nil} \to Prime(s.\mathsf{b})))).$$

For $p \neq r$ we have that $p.\mathsf{m} \neq \mathsf{nil} \to Prime(p.\mathsf{m})$ is implied by I_1. For $p \doteq r$ we have to show that $I \land \psi_0 \downarrow r \land \psi_3 \downarrow s \land s.\mathsf{l} \doteq r$ implies $Prime(s.\mathsf{b})$. Here the main point is that there exist no primes between $s.\mathsf{m}$ and $s.\mathsf{b}$. For suppose that i is the least prime such that $s.\mathsf{m} < i < s.\mathsf{b}$. Now by I_5 and I_6 there exists a p' such that $p'.\mathsf{m} \doteq i$ or $p'.\mathsf{b} \doteq i$. If $p'.\mathsf{m} \doteq i$ then it follows by I_2 that $s.\mathsf{l} \doteq p'$. But, as $s.\mathsf{l} \doteq r$ we have $p' \doteq r$, so $p'.\mathsf{m} \doteq \mathsf{nil}$, which contradicts $p'.\mathsf{m} \doteq i$. Suppose now that $p'.\mathsf{b} \doteq i$. We then have that $p'.\mathsf{b} < s.\mathsf{b}$, so by I_3 it follows that $p'.\mathsf{m} > s.\mathsf{m}$. But since $p'.\mathsf{m}$ is a prime by I_1, and i is the least prime greater than $s.\mathsf{m}$, it follows that $p'.\mathsf{m} \geq i \doteq p'.\mathsf{b}$, which contradicts I_8. Therefore there can be no primes between $s.\mathsf{m}$ and $s.\mathsf{b}$ and then $\psi_3 \downarrow s$ implies that $s.\mathsf{b}$ is a prime number.

As another example of an assumption for the cooperation test, we consider the following new-statement:

$$\{I \land (\mathsf{l} \doteq \mathsf{nil}) \downarrow z\}(z, \mathsf{l} := \mathsf{new})\{I \land \psi_0 \downarrow z.\mathsf{l}\}$$

Here z is a logical variable ranging over objects of class P. By the axiom (NEW) the proof of this correctness formula amounts to proving the validity of

$$I \land (\mathsf{l} \doteq \mathsf{nil}) \downarrow z \to (I \land \psi_0 \downarrow z.\mathsf{l})[z'/z.\mathsf{l}][\mathsf{new}/z']$$

where z' is another new logical variable of type P. We only show that $I \land (\mathsf{l} \doteq \mathsf{nil}) \downarrow z$ implies $I_2[z'/z.\mathsf{l}][\mathsf{new}/z']$. Now $I_2[z'/z.\mathsf{l}]$ is the formula

$$\forall p \forall p'(p.\mathsf{m} \neq \mathsf{nil} \to (p'.\mathsf{m} \doteq nextpr(p.\mathsf{m}) \to \mathsf{if}\ p \doteq z\ \mathsf{then}\ z'\ \mathsf{else}\ p.\mathsf{l}\ \mathsf{fi} \doteq p'))$$

This is equivalent to

$$\forall p \forall p'(p.\mathsf{m} \neq \mathsf{nil} \to (p'.\mathsf{m} \doteq nextpr(p.\mathsf{m}) \to (p \doteq z \to z' \doteq p') \land (p \neq z \to p.\mathsf{l} \doteq p'))).$$

To this formula we apply the substitution $[\mathsf{new}/z']$, which gives us

$$\begin{aligned}
&\forall p \forall p'(p.\mathsf{m} \neq \mathsf{nil} \to (p'.\mathsf{m} \doteq nextpr(p.\mathsf{m}) \to \\
&\qquad\qquad (p \doteq z \to \mathsf{false}) \land (p \neq z \to p.\mathsf{l} \doteq p'))) \\
&\land \forall p(p.\mathsf{m} \neq \mathsf{nil} \to (\mathsf{nil} \doteq nextpr(p.\mathsf{m}) \to (p \doteq z \to \mathsf{true}) \land (p \neq z \to \mathsf{false}))) \\
&\land \forall p'(\mathsf{nil} \neq \mathsf{nil} \to (p'.\mathsf{m} \doteq nextpr(\mathsf{nil}) \to (\mathsf{false} \to \mathsf{false}) \land (\mathsf{true} \to \mathsf{nil} \doteq p'))) \\
&\land (\mathsf{nil} \neq \mathsf{nil} \to (\mathsf{nil} \doteq nextpr(\mathsf{nil}) \to (\mathsf{false} \to \mathsf{true}) \land (\mathsf{true} \to \mathsf{false})))
\end{aligned}$$

Note that the first conjunct corresponds to interpreting p and p' as ranging over the old objects. The second conjunct results from interpreting p' as the new object, the third from interpreting p as the new object, and the last from taking both p and p' as the new object. All conjuncts but the first are trivially true, and we can get the first conjunct from I_2 if we can show that $\forall p \forall p'(p.\mathsf{m} \neq \mathsf{nil} \to p'.\mathsf{m} \doteq nextpr(p.\mathsf{m}) \to p \neq z)$. Let $p \neq \mathsf{nil}$, $p' \neq \mathsf{nil}$, $p.\mathsf{m} \neq \mathsf{nil}$, $p'.\mathsf{m} \doteq nextpr(p.\mathsf{m})$, and $p \doteq z$. By I_2 we derive that $z.\mathsf{l} \doteq p'$. But this contradicts the precondition $z.\mathsf{l} \doteq \mathsf{nil}$.

The other parts of the cooperation test can be dealt with in the same way as the above ones. After that the program rule (PR) can be applied, with the following result:

$$\{\mathsf{true}\}\rho'\{I \land g.\mathsf{c} \doteq \max(2, n+1) \land \forall p\, p.\mathsf{b} \doteq \mathsf{nil}\}$$

Here ρ' is the program with the auxiliary variable v, but by applying the auxiliary variable rule (AUX) the same result can also be obtained for the original program ρ. It is easy to see that the above postcondition implies the desired assertion 5.1, so the latter can be obtained by the consequence rule (PCR).

6. Semantics

In this section we define in a formal way the semantics of the programming language and the assertion languages. First, in section 6.1, we deal with the assertion languages on their own. Then, in section 6.2, we give a formal semantics to the programming language, making use of *transition systems*. Finally, section 6.3 formally defines the notion of truth of a correctness formula.

6.1. Semantics of the Assertion Languages

For every type $a \in C^{\dagger}$, we shall let $\mathbf{O}^a$ denote the set of objects of type a, with typical element α. To be precise, we define $\mathbf{O}^{\mathsf{Int}} = \mathbf{Z}$ and $\mathbf{O}^{\mathsf{Bool}} = \mathbf{B}$, whereas for every class $c \in C$ we just take for $\mathbf{O}^c$ an arbitrary infinite set *disjoint* from any other set $\mathbf{O}^d$, $d \neq c$. With $\mathbf{O}^d_\perp$ we shall denote $\mathbf{O}^d \cup \{\perp\}$, where $\perp$ is a special element not in $\mathbf{O}^d$, which will stand for 'undefined', among others the value of the expression nil. Now for every type $d \in C^+$ we let $\mathbf{O}^{d^*}$ denote the set of all finite sequences of elements from $\mathbf{O}^d_\perp$ and we take $\mathbf{O}^{d^*}_\perp = \mathbf{O}^{d^*}$. This means that sequences can contain $\perp$ as a component, but a sequence can never be $\perp$ itself (as an expression of a sequence type, nil just stands for the empty sequence). The set $\bigcup_c \mathbf{O}^c$ of non-standard objects we denote by $\mathbf{O}$. The set of all objects is denoted by $\mathbf{D}$.

Definition 6.1. The set of functions from a set A to a set B we denote by $A \to B$. Given a function $f \in A \to B$, $a \in A$, and $b \in B$, we use the *variant notation* $f\{b/a\}$ to denote the function in $A \to B$ that satisfies

$$f\{b/a\}(a') = \begin{cases} b & \text{if } a' = a \\ f(a') & \text{otherwise.} \end{cases}$$

We will also use in the sequel the notation $(A \to B)_\perp$ for the set $(A \to B) \cup \{\perp\}$.

Definition 6.2. The set $LState^c$ of *local states* of objects of class c, with typical element θ, is defined by

$$LState^c = \mathbf{O}^c \times (IVar \to \mathbf{D})$$

A local state θ describes in detail the situation of a single object at a certain moment during program execution. The first component determines the identity of the object and the values of the instance variables are given by the second component where we require that each variable is assigned an object of its corresponding type.

It will turn out to be convenient to define the function $\nabla \in IVar \to \mathbf{D}$ such that $\nabla(x) = \perp$, for every instance variable x. Note that this function ∇ gives the values of the variables of a newly created object: these are all initialised to nil.

Definition 6.3. The set *GState* of *global states*, with typical element σ, is defined as follows:

$$GState = \mathbf{O} \rightarrow (IVar \rightarrow \mathbf{D})_{\perp}$$

A global state describes the situation of a complete system of objects at a certain moment during program execution. Relative to some global state σ an object $\alpha \in \mathbf{O}$ can be said to *exist* if $\sigma(\alpha) \neq \perp$, i.e. if $\sigma(\alpha)$ is defined. In other words, the set $\{\alpha \mid \sigma(\alpha) \neq \perp\}$ represents the set of objects that have been created up to this point in the execution of the program. For any existing object α the values of its instance variables are specified by $\sigma(\alpha)$ (where we require as above that each variable is assigned an object corresponding to its type).

We introduce the following abbreviations: for $c \in C$, the set $\{\alpha \in \mathbf{O}^c \mid \sigma(\alpha) \neq \perp\}$ of the existing objects of class c, will be abbreviated to $\sigma^{(c)}$, and $\sigma^{(c)} \cup \{\perp\}$ to $\sigma_{\perp}^{(c)}$. For $d = \mathsf{Int}$ or $d = \mathsf{Bool}$ we put $\sigma^{(d)} = \mathbf{O}^d$.

Definition 6.4. We now define the set *LEnv* of *logical environments*, with typical element ω, by

$$LEnv = LVar \rightarrow \mathbf{D}$$

Again, it is required that each variable is assigned an object corresponding to its type.

A logical environment assigns values to logical variables.

Definition 6.5. The following semantic functions are defined in a straightforward manner. We omit most of the detail and only give the most important cases:

1. The function $\mathscr{E} \in Exp_d^c \rightarrow LState^c \rightarrow \mathbf{O}_{\perp}^d$ assigns a value $\mathscr{E}[\![e]\!](\theta)$ to the expression e in the local state θ. A typical case: $\mathscr{E}[\![\mathsf{self}]\!](\langle\alpha, s\rangle) = \alpha$.
2. The function $\mathscr{L} \in LExp_a^c \rightarrow LEnv \rightarrow LState^c \rightarrow \mathbf{O}_{\perp}^a$ gives a value $\mathscr{L}[\![l]\!](\omega)(\theta)$ to the local expression l in the logical environment ω and the local state θ.
3. The function $\mathscr{G} \in GExp_a \rightarrow LEnv \rightarrow GState \rightarrow \mathbf{O}_{\perp}^a$ gives a value $\mathscr{G}[\![g]\!](\omega)(\sigma)$ to the global expression g in the logical environment ω and the global state σ.
4. The function $\mathscr{A} \in LAss^c \rightarrow LEnv \rightarrow LState^c \rightarrow \mathbf{B}$ assigns a value $\mathscr{A}[\![p]\!](\omega)(\theta)$ to the local assertion p in the logical environment ω and the local state θ. Here the following cases are special: For a boolean expression l we have

$$\mathscr{A}[\![l]\!](\omega)(\theta) = \begin{cases} \mathsf{true} & \text{if } \mathscr{L}[\![l]\!](\omega)(\theta) = \mathsf{true} \\ \mathsf{false} & \text{if } \mathscr{L}[\![l]\!](\omega)(\theta) = \mathsf{false} \text{ or } \mathscr{L}[\![l]\!](\omega)(\theta) = \perp \end{cases}$$

   and, for $z \in LVar_a$ (with $a = d, d^*$, $d = \mathsf{Int}, d = \mathsf{Bool}$),

$$\mathscr{A}[\![\exists z\, p]\!](\omega)(\theta) = \begin{cases} \mathsf{true} & \text{if there is an } \alpha \in \mathbf{O}^a \text{ such that} \\ & \mathscr{A}[\![p]\!](\omega\{\alpha/z\})(\theta) = \mathsf{true} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

   Note that at the level of the local assertion language we only have quantification over (sequences of) integers or booleans and that the range of quantification *does not include* $\perp$.
5. The function $\mathscr{A} \in GAss \rightarrow LEnv \rightarrow GState \rightarrow \mathbf{B}$ assigns a value $\mathscr{A}[\![P]\!](\omega)(\sigma)$ to the global assertion P in the logical environment ω and the global state σ. The following cases are special: For a boolean expression g we have

$$\mathscr{A}[\![g]\!](\omega)(\sigma) = \begin{cases} \textsf{true} & \text{if } \mathscr{L}[\![g]\!](\omega)(\sigma) = \textsf{true} \\ \textsf{false} & \text{if } \mathscr{L}[\![g]\!](\omega)(\sigma) = \textsf{false or } \mathscr{L}[\![g]\!](\omega)(\sigma) = \bot \end{cases}$$

and, for $z \in LVar_d$,

$$\mathscr{A}[\![\exists z\, P]\!](\omega)(\sigma) = \begin{cases} \textsf{true} & \text{if there is an } \alpha \in \sigma^{(d)} \text{ such that} \\ & \mathscr{A}[\![P]\!](\omega\{\alpha/z\})(\sigma) = \textsf{true} \\ \textsf{false} & \text{otherwise} \end{cases}$$

Note that here d can be any type in C^+ and that the quantification ranges over $\sigma^{(d)}$, the set of *existing* objects of type d (which does not include $\bot$). Furthermore, we have, assuming $z \in LVar_{d^*}$,

$$\mathscr{A}[\![\exists z\, P]\!](\omega)(\sigma) = \begin{cases} \textsf{true} & \text{if there is an } \alpha \in \mathbf{O}^{d^*} \text{ such} \\ & \text{that } \alpha(n) \in \sigma_\bot^{(d)} \text{ for all } n \in \mathbf{N} \\ & \text{and } \mathscr{A}[\![P]\!](\omega\{\alpha/z\})(\sigma) = \textsf{true} \\ \textsf{false} & \text{otherwise} \end{cases}$$

For sequence types, quantification ranges over those sequences of which every element is either $\bot$ or an existing object.

The values $\mathscr{G}[\![g]\!](\omega)(\sigma)$ of the global expression g and $\mathscr{A}[\![P]\!](\omega)(\sigma)$ of the global assertion P are in fact only meaningful for those ω and σ that are consistent and compatible.

Definition 6.6. We define the global state σ to be *consistent*, for which we use the notation $OK(\sigma)$ iff the value in σ of a variable of an existing object is either $\bot$ or an existing object itself. Furthermore we define the logical environment ω to be *compatible* with the global state σ, with the notation $OK(\omega, \sigma)$, iff $OK(\sigma)$ and, additionally, ω assigns to every logical variable z of a simple type the value $\bot$ or an existing object, and to every sequence variable z a sequence of which each element is an existing object or equals $\bot$.

The following lemma describes the relation between a local expression (assertion) and its translation into the global assertion language.

Lemma 6.7.
For every local expression $l(\in LExp_a^c)$ and local assertion $p(\in LAss^c)$, logical environment ω, and global state σ, we have

$$\mathscr{L}[\![l]\!](\omega)(\langle\alpha, \sigma(\alpha)\rangle) = \mathscr{G}[\![l \downarrow z]\!](\omega\{\alpha/z\})(\sigma)$$

and

$$\mathscr{A}[\![p]\!](\omega)(\langle\alpha, \sigma(\alpha)\rangle) = \mathscr{A}[\![p \downarrow z]\!](\omega\{\alpha/z\})(\sigma)$$

Here $\alpha \in \sigma^{(c)}$ and $z \in LVar_c$.

Proof. Straightforward induction on the complexity of l and p. □

It is worthwhile to note that the above lemma implies that indeed the value of a local expression (assertion) only depends on the local state of the object under consideration. Proof-theoretically this is important because it guarantees that local assertions are *interference free* from the execution of co-existing objects.

In the sequel we will also use the following notation of the truth of an assertion:

$$\theta, \omega \models p$$

for $\mathscr{A}[\![p]\!](\omega)(\theta) = \text{true}$, and, similarly,

$$\sigma, \omega \models P$$

for $\mathscr{A}[\![P]\!](\omega)(\sigma) = \text{true}$.

6.2. The Transition System

We will describe the internal behaviour of an object by means of a transition system. One of the main problems that arises is the treatment of communication and creation statements, since the execution of these statements clearly depend on the environment as given by the other objects running in parallel. A general approach to this problem consists in describing communication and creation statements in terms of a local *history* which records all the communications and activations an object has been engaged in. Such a history then provides an interface between an object and its environment, and it will allow, as will be shown later, a *compositional* description of the behaviour of a complete system in terms of the local behaviour of its objects.

Formally, we define a *local configuration* to be a triple (S, θ, h), where h denotes the local history. A history consists of a sequence of *activation* and *communication* records. A pair $< \alpha, \beta >$, with $\alpha, \beta \in \mathbf{O}$, is called an *activation record*. It records the information that the object α created β. A triple $< \alpha, \beta, \gamma >$, with $\alpha, \beta \in \mathbf{O}_\perp$, is called a *communication record*. It records the information that the object γ is sent by α to β, as a special case we also allow communication records $< \perp, \beta, \gamma >$ which indicate that the sender is unknown. The empty history will be denoted by λ. Given a history h and an activation or communication record r the history resulting from appending r to h will be denoted by $h \bullet r$. To facilitate the semantics we introduce the auxiliary statement E, the empty statement, to denote termination. The following definition specifies the local transition system.

Definition 6.8. We define

- $(x := e, \theta, h) \rightarrow (E, \theta', h)$,
  here $\theta = \langle \alpha, s \rangle$ and $\theta' = \langle \alpha, s\{\mathscr{E}[\![e]\!]\ (\theta)/x\} \rangle$.
- $(x := \text{new}, \theta, h) \rightarrow (E, \theta', h\bullet < \alpha, \beta >)$,
  here $\theta = \langle \alpha, s \rangle$ and $\theta' = \langle \alpha, s\{\beta/x\} \rangle$.
- $(x!e, \theta, h) \rightarrow (E, \theta, h\bullet < \alpha, \beta, \gamma >)$,
  here $\theta = \langle \alpha, s \rangle$, $\beta = s(x) \neq \perp$, and $\gamma = \mathscr{E}[\![e]\!]\ (\theta)$.
- $(?y, \theta, h) \rightarrow (E, \theta', h\bullet < \perp, \beta, \gamma >)$,
  here $\theta = \langle \beta, s \rangle$ and $\theta' = \langle \beta, s\{\gamma/y\} \rangle$.
- $(x?y, \theta, h) \rightarrow (E, \theta', h\bullet < \alpha, \beta, \gamma >)$,
  here $\theta = \langle \beta, s \rangle$, $\theta' = \langle \beta, s\{\gamma/y\} \rangle$, and $\alpha = s(x) \neq \perp$.
- $(E; S, \theta, h) \rightarrow (S, \theta, h)$.
- $\dfrac{(S_1, \theta, h) \rightarrow (S_2, \theta', h')}{(S_1; S, \theta, h) \rightarrow (S_2; S, \theta', h')}$
- $(\text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}, \theta, h) \rightarrow (S_1, \theta, h)$,
  if $\mathscr{E}[\![e]\!]\ (\theta) = \text{true}$.
- $(\text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}, \theta, h) \rightarrow (S_2, \theta, h)$,
  if $\mathscr{E}[\![e]\!]\ (\theta) = \text{false}$.

- (while e do S od, $\theta, h) \to (S$; while e do S od, $\theta, h)$, if $\mathscr{E}[\![e]\!]\ (\theta) =$ true.
- (while e do S od, $\theta, h) \to (E, \theta, h)$, if $\mathscr{E}[\![e]\!]\ (\theta) =$ false.

Note that locally the value which is received when executing an input statement is chosen arbitrarily (the only restriction being that its type corresponds with the one of the input variable), the same holds for the identity of the object created by the execution of a new-statement. Furthermore note that in the case of the execution of an input statement $?y$ the identity of the sender is not known locally, which is indicated by $\perp$. The reflexive, transitive closure of the transition relation $\to$ is denoted by $\to^*$.

To describe the behaviour of several objects working in parallel we introduce the notion of a *global configuration*: a triple (X, σ, h), where $X \in \mathbf{O} \to STat$. The idea is that $X(\alpha)$ denotes the statement to be executed by α. The history h represents the *global history* of the complete system, as such it is assumed to be *compatible* with the global state σ in the following sense:

Definition 6.9. We define the history h to be *compatible* with the global state σ, with the notation $OK(\sigma, h)$ iff $OK(\sigma)$ and

- there exists in σ a unique object, the root-object, for which there are no activation records of h witnessing its creation,
- all the (non-standard) objects occurring in h do exist (in σ), and, conversely, for every existing object (of σ) but the root-object there exists exactly one activation record witnessing its creation,
- furthermore, for every object (but the root-object) its activities as recorded by the history h all occur after its creation,
- finally, we require that in every communication record of h the sender is determined, i.e. when $< \alpha, \beta, \gamma >$ occurs in h then $\alpha \neq \perp$.

It is interesting to note that for example an object cannot create itself: since $< \alpha, \alpha >$ records the creation of α by itself, it records an activity of α and as such is required to occur after itself (every object is created only once) which is clearly impossible.

To define the behaviour of a complete system in terms of the local behaviour of its objects we need the following: given a history h and an object α the subsequence of h consisting of all the records involving α (but for the one which records its creation) will be denoted by $[h]_\alpha$. With respect to the definition of the semantics of global configurations below it is instructive to point out the relationship between a local history h of an object α and its corresponding global view $[h']_\alpha$, where h' represents the global history of a complete system: in $[h']_\alpha$ we know additionally all the objects which have been involved in a communication with α. Formally we have that $h \leq [h']_\alpha$, where for any histories h_1 and h_2:

$h_1 \leq h_2$ iff

- $h_1 = h_2 = \lambda$, or
- $h_1 = h'_1 \bullet < \alpha, \beta >$, $h_2 = h'_2 \bullet < \alpha, \beta >$, with $h'_1 \leq h'_2$, or
- $h_1 = h'_1 \bullet < \beta, \alpha, \gamma >$, $h_2 = h'_2 \bullet < \beta', \alpha, \gamma >$, with $h'_1 \leq h'_2$ and if $\beta \neq \perp$ then $\beta = \beta'$.

Definition 6.10. Now we can define the transition relation between global configurations.

$(X,\sigma,h) \to (X',\sigma',h')$ iff for all existing (with respect to σ') objects α

$$(X(\alpha),\theta_\alpha,h_\alpha) \to^* (X'(\alpha),\theta'_\alpha,h'_\alpha)$$

where $h_\alpha \leq [h]_\alpha$ and $h'_\alpha \leq [h']_\alpha$. Furthermore, $\theta'_\alpha = \langle\alpha,\sigma'(\alpha)\rangle$, and $\theta_\alpha = \langle\alpha,\sigma(\alpha)\rangle$ if $\sigma(\alpha)$ is defined, that is, α exists also in σ, and $\theta_\alpha = \langle\alpha,\nabla\rangle$, otherwise.

The local behaviour of the objects is derived from the local transition system. The projection mechanism on the global history guarantees that the choices concerning communications as recorded by the local histories agree with each other. With respect to the creation of an object we know that indeed a new (not yet existing) object is called into existence because the history is assumed to be compatible with the global state, so every object but the root-object is created exactly once.

It is instructive to compare the above global transition relation with the transition relation between configurations of the form (X,σ) as defined along the lines as given in [ABK86]: there a transition $(X,\sigma) \to (X',\sigma')$ specifies either a local computation step of an object (including the creation of an object) or a communication between two objects. We have the following correspondence:

$(X,\sigma) \to^* (X',\sigma')$ iff there exists h and h' such that $(X,\sigma,h) \to (X',\sigma',h')$

which can be shown by induction on the length of the computation.

Now we are able to define the meaning of the following programming constructs: S, (z,S), $(z_1,S_1) \parallel (z_2,S_2)$.

Definition 6.11. We define

$$\mathscr{S}[\![S]\!](\theta) = \{\theta' \mid \text{ for some } h\text{: } (S,\theta,\lambda) \to^* (E,\theta',h)\}$$

Definition 6.12. We define

$$\mathscr{I}[\![(z,S)]\!](\omega)(\sigma) = \left\{\sigma' \mid \text{ for some } h,\, h'\text{: } (\{\langle\alpha,S\rangle\},\sigma,h) \to (\{\langle\alpha,E\rangle\},\sigma',h')\right\}$$

where $\omega(z) = \alpha$. (Here $\{\langle\alpha,E\rangle\}$, for example, represents $X \in \mathbf{O} \to STat$ such that $X(\alpha) = S$ and $X(\beta) = E$, for $\beta \neq \alpha$.) Moreover, we define, for a logical environment ω such that $\omega(z) = \alpha \neq \omega(z') = \alpha'$,

$$\mathscr{I}[\![(z,S) \parallel (z',S')]\!](\omega)(\sigma) =$$
$$\left\{\sigma' \mid \text{ for some } h,\, h'\text{: } (\{\langle\alpha,S\rangle,\langle\alpha',S'\rangle\},\sigma,h) \to (\{\langle\alpha,E\rangle,\langle\alpha',E\rangle\},\sigma',h')\right\}$$

To define the meaning of a program we first introduce the following definitions.

Definition 6.13. Let $\rho = \langle c_1 \leftarrow S_1,\ldots,c_n \leftarrow S_n\rangle$. Furthermore let $X \in \mathbf{O} \to STat$. We define

$$\begin{aligned} Init_\rho(X) \text{ iff } X(\alpha) &= S_i \quad \text{if } \alpha \in \mathbf{O}^{c_i},\ 1 \leq i \leq n \\ &= E \quad \text{otherwise} \end{aligned}$$

Next we define

$$Final_\rho(X,\sigma) \text{ iff } X(\alpha) = E, \text{ for } \alpha \in \sigma^{(c_i)},\ 1 \leq i \leq n$$

The predicate $Init_\rho(Final_\rho)$ characterises the set of *initial* (*final*) configurations of ρ.

Now we are able to define the meaning of a program.

Definition 6.14. The semantics of programs is defined as follows:

$$\mathscr{P}[\![\rho]\!](\theta) = \{\sigma \mid \text{for some } h\colon (X, \theta, \lambda) \rightarrow (X', \sigma, h)\}$$

where $Init_\rho(X)$ and $Final_\rho(X', \sigma')$. The local state of the root-object is given by θ and as such we assume that all the variables $x \in IVar_d$, where $d \in C$, are uninitialised (that is $\theta(x) = \bot$). Note that thus θ can be viewed as a global state where only the root-object exists.

6.3. Truth of Correctness Formulas

In this section we define formally the truth of the local, intermediate, and global correctness formulas, respectively. First we define the truth of local correctness formulas.

Definition 6.15. We define

$$\models \{p\}S\{q\} \text{ iff } \forall \omega, \theta, \theta' \in \mathscr{S}[\![S]\!](\theta) : \theta, \omega \models p \Rightarrow \theta', \omega \models q$$

Next we define the truth of intermediate correctness formulas.

Definition 6.16. We define

$$\models \{P\}(z, S)\{Q\}$$
$$\text{iff}$$
$$\forall \omega, \sigma, \sigma' \in \mathscr{I}[\![(z, S)]\!](\omega)(\sigma) : \sigma, \omega \models P \Rightarrow \sigma', \omega \models Q$$

and

$$\models \{P\}(z, S) \parallel (z', S')\{Q\}$$
$$\text{iff}$$
$$\forall \omega, \sigma, \sigma' \in \mathscr{I}[\![(z, S) \parallel (z', S')]\!](\omega)(\sigma) : \sigma, \omega \models P \Rightarrow \sigma', \omega \models Q$$

Finally, we define the truth of global correctness formulas.

Definition 6.17. We define

$$\models \{p\}\rho\{Q\} \text{ iff } \forall \omega, \theta, \sigma \in \mathscr{P}[\![\rho]\!](\theta) : \theta, \omega \models p \Rightarrow \sigma', \omega \models Q$$

7. Soundness

In this section we prove the soundness of the proof system as presented in the previous section. The soundness of the local proof system is proved by a straightforward induction on the length of the derivation (see, for example, [Apt81]). In the following subsection we discuss the soundness of the intermediate proof system.

7.1. The Intermediate Proof system

We prove the soundness of the assignment axiom (IASS) and the axiom (NEW). The soundness of the intermediate proof system then follows by a straightforward induction argument. To prove the soundness of the assignment axiom (IASS) we need the following lemma about the correctness of the corresponding substitution

operation. This lemma states that semantically substituting the expression g' for $z.x$ in an assertion (expression) yields the same result when evaluating the assertion (expression) in the state where the value of g' is assigned to the variable x of the object denoted by z.

Lemma 7.1.
For an arbitrary σ, ω such that $OK(\omega, \sigma)$ we have:

$$\mathscr{G}[\![g[g'/z.x]]\!](\omega)(\sigma) = \mathscr{G}[\![g]\!](\omega)(\sigma')$$

and

$$\mathscr{A}[\![P[g'/z.x]]\!](\omega)(\sigma) = \mathscr{A}[\![P]\!](\omega)(\sigma')$$

where $\sigma' = \sigma\{\mathscr{G}[\![g']\!](\omega)(\sigma)/\omega(z), x\}$

Proof. By induction on the complexity of g and P. We treat only the case $g = g_1.x$, all the other ones following directly from the induction hypothesis. Now:

$$\mathscr{G}[\![g[g'/z.x]]\!](\omega)(\sigma) =$$
$$\mathscr{G}[\![\textsf{if } g_1[g'/z.x] \doteq z \textsf{ then } g' \textsf{ else } g_1[g'/z.x].x \textsf{ fi}]\!](\omega)(\sigma)$$

Suppose that $\mathscr{G}[\![g_1[g'/z.x]]\!](\omega)(\sigma) = \omega(z)$. Then we have that $\mathscr{G}[\![g_1.x]\!](\omega)(\sigma') = \sigma'(\mathscr{G}[\![g_1]\!](\omega)(\sigma'))(x)$. So by the induction hypothesis we have

$$\mathscr{G}[\![g_1.x]\!](\omega)(\sigma') = \sigma'(\omega(z))(x) = \mathscr{G}[\![g']\!](\omega)(\sigma)$$

On the other hand if $\mathscr{G}[\![g_1[g'/z.x]]\!](\omega)(\sigma) \neq \omega(z)$ then:

$$\begin{array}{ll} \mathscr{G}[\![g_1[g'/z.x].x]\!](\omega)(\sigma) & = \\ \sigma(\mathscr{G}[\![g_1[g'/z.x]]\!](\omega)(\sigma))(x) & = \text{(definition of } \sigma'\text{)} \\ \sigma'(\mathscr{G}[\![g_1[g'/z.x]]\!](\omega)(\sigma))(x) & = \text{(induction hypothesis)} \\ \sigma'(\mathscr{G}[\![g_1]\!](\omega)(\sigma'))(x) & = \\ \mathscr{G}[\![g_1.x]\!](\omega)(\sigma'). & \end{array}$$

□

The following lemma states the soundness of the axiom (IASS).

Lemma 7.2.
We have

$$\models \{P[e \downarrow z/z.x]\}(z, x := e)\{P\},$$

Proof. Let σ, ω, with $OK(\omega, \sigma)$, such that $\sigma, \omega \models P[e \downarrow z/z.x]$ and $\sigma' \in \mathscr{I}[\![(z, x := e)]\!](\omega)(\sigma)$. It follows that $\sigma' = \sigma\{\mathscr{G}[\![e \downarrow z]\!](\omega)(\sigma)/\omega(z), x\}$ (note that by lemma 6.7 we have $\mathscr{G}[\![e \downarrow z]\!](\omega)(\sigma) = \mathscr{E}[\![e]\!](\langle\alpha, \sigma(\alpha)\rangle)$, with $\alpha = \omega(z)$). Thus by the previous lemma we conclude $\sigma', \omega \models P$. □

To prove the soundness of the axiom describing the new statement we need the following lemma which states the correctness of the corresponding substitution operation. This lemma states that semantically the substitution [new/z] applied to an assertion yields the same result when evaluating the assertion in the state resulting from the creation of a new object, interpreting the variable z as the newly created object.

Lemma 7.3.
For an arbitrary $\omega, \omega', \sigma, \sigma'$, $\beta \in \mathbf{O}^c \backslash \sigma^{(c)}$ such that $OK(\omega, \sigma)$, $\sigma' = \sigma\{\nabla/\beta\}$, and $\omega' = \omega\{\beta/z\}$ ($z \in LVar_c$), we have for an arbitrary assertion P:

$$\mathscr{A}[\![P[\mathsf{new}/z]]\!](\omega)(\sigma) = \mathscr{A}[\![P]\!](\omega')(\sigma').$$

The proof of this lemma proceeds by induction on the structure of P. To carry out this induction argument, which we trust the interested reader to be able to perform, we need the following two lemmas. The first of which is applied to the case $P = g$ and the second of which is applied to the case $P = \exists z P'$, $z \in LVar_a$, $a = c, c^*$.

Lemma 7.4.
For an arbitrary σ, ω, with $OK(\omega, \sigma)$, global expression g and logical variable $z \in LVar_c$ such that $g[\mathsf{new}/z]$ is defined we have:

$$\mathscr{G}[\![g]\!](\omega')(\sigma') = \mathscr{G}[\![g[\mathsf{new}/z]]\!](\omega)(\sigma)$$

where $\sigma' = \sigma\{\nabla/\beta\}$, and $\omega' = \omega\{\beta/z\}$, $\beta \in \mathbf{O}^c \setminus \sigma^{(c)}$.

Proof. Induction on the structure of g. □

The following lemma states that semantically the substitution $[z'', z/z']$ ($z'' \in LVar_{\mathsf{Bool}^*}$, $z \in LVar_c$, and $z' \in LVar_{c^*}$) applied to an assertion (expression) yields the same result when updating the sequence denoted by the variable z' to the value of z at those positions for which the sequence denoted by z'' gives the value true.

Lemma 7.5.
Let ω, σ, $\alpha = \omega(z'), \alpha' = \omega(z'')$ such that $|\alpha| = |\alpha'|$ and $OK(\omega, \sigma)$
Let $\alpha'' \in \mathbf{O}^{c^*}$ such that

- $|\alpha''| = |\alpha|$
- for $n \in \mathbf{N}$: $\alpha''(n) = \omega(z)$ if $\alpha'(n) = \mathsf{true}$
  $= \alpha(n)$ if $\alpha'(n) = \mathsf{false}$
  $= \bot$ if $\alpha'(n) = \bot$

Let $\omega' = \omega\{\alpha''/z'\}$. Then:

1. For every g such that $g[z'', z/z']$ is defined:

$$\mathscr{G}[\![g[z'', z/z']]\!](\omega)(\sigma) = \mathscr{G}[\![g]\!](\omega')(\sigma)$$

2. For every P such that z'' does not occur in it:

$$\mathscr{A}[\![P[z'', z/z']]\!](\omega)(\sigma) = \mathscr{A}[\![P]\!](\omega')(\sigma)$$

Proof. Induction on the structure of g and P. □

Now we are ready to prove the soundness of the axiom (NEW).

Lemma 7.6.
We have

$$\models \{P[z'/z.x][\mathsf{new}/z']\}(z, x := \mathsf{new})\{P\},$$

where $z \in LVar_c$, and z' is a new logical variable of the same type as x.

Proof. Let σ, ω, with $OK(\sigma, \omega)$, such that $\sigma, \omega \models P[z'/z.x][\mathsf{new}/z']$ and $\sigma' \in \mathcal{I}[\![(z, x := \mathsf{new})]\!](\omega)(\sigma)$. We have by lemma 7.3 that $\sigma'', \omega' \models P[z'/z.x]$, where $\omega' = \omega\{\beta/z'\}$, and $\sigma'' = \sigma\{\nabla/\beta\}$, with $\beta = \sigma'(\omega(z))(x)$. Now by lemma 7.1 it follows that $\sigma', \omega' \models P$. Finally, as z' does not occur in P we have $\sigma', \omega \models P$. □

7.2. The Global Proof System

In this subsection we prove the soundness of the global proof system. We will prove only the soundness of the rule (PR), the other rules being straightforward to deal with. The problem with proving the soundness of the rule (PR) is how to interpret the premise $A \vdash \{p\}S\{q\}$. We solve this problem by showing that a proof $A \vdash \{p\}S\{q\}$ essentially boils down to a finite conjunction of local assertions. But first we observe that we may restrict the premise of the rule (PR) without loss of generality to local proofs which do not make use of the invariance axiom (INV) and the substitution axiom (SUB). This can be argued as follows. Let $\vdash^-$ denote the local proof system without the axioms (INV) and (SUB). Then for any cooperating proofs (with respect to some global invariant I) $A_i \vdash \{p_i\}S_i\{q_i\}$ $(1 \leq i \leq n)$ it is not difficult to see that there exist proofs $A_i' \vdash^- \{p_i\}S_i\{q_i\}$ that cooperate as well (with respect to I), with

$$A_i' \subseteq \Big\{\{p'\}R\{q'\} \mid \{p\}R\{q\} \in A_i \text{ and } \{p\}R\{q\} \vdash \{p'\}R\{q'\}\Big\}$$

Note that a proof $\{p\}R\{q\} \vdash \{p'\}R\{q'\}$, with R a bracketed section, can only involve the assumption $\{p\}R\{q\}$, the invariance axiom, the substitution axiom, the conjunction rule, and the consequence rule.

To show that any local proof $A \vdash^- \{p\}S\{q\}$ essentially boils down to a finite conjunction of local assertions we first introduce the following.

Definition 7.7. Given a bracketed program ρ, R a substatement of ρ, we define R to be *stable* iff every bracketed section of ρ occurs inside or outside of R.

Next we define *After*(R, S), where R is a substatement of S, to be the statement to be executed when the execution of R has just terminated. On the other hand we will define *Before*(R, S) to be the statement to be executed when the execution of R is about to start.

Definition 7.8. Let R be a substatement of the statement S. We define *After*(R, S) as follows:

- If $R = S$ then $\mathit{After}(R, S) = E$
- If $S = S_1; S_2$
  then $\mathit{After}(R, S) = \mathit{After}(R, S_1); S_2$ if R occurs in S_1
  $\mathit{After}(R, S) = \mathit{After}(R, S_2)$ if R occurs in S_2
- If $S = \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$
  then $\mathit{After}(R, S) = \mathit{After}(R, S_1)$ if R occurs in S_1
  $\mathit{After}(R, S) = \mathit{After}(R, S_2)$ otherwise
- If $S = \mathsf{while}\ e\ \mathsf{do}\ S_1\ \mathsf{od}$ then $\mathit{After}(R, S) = \mathit{After}(R, S_1); S$

Next we define $\mathit{Before}(R, S) = R; S'$, where $\mathit{After}(R, S) = E; S'$.
Note that for the above definition to be formally correct we have to assume some mechanism which enables one to distinguish between different occurrences of an arbitrary statement. We will simply assume such a mechanism to exist. So in the

sequel when referring to a statement we in fact will sometimes mean a particular occurrence of that statement.

Now we can state the following lemma which reduces a proof $A \vdash^{-} \{p\}S\{q\}$ to a finite conjunction of local assertions.

Lemma 7.9.
Let S be a statement such that every I/O statement and new statement occurring in it is contained in a bracketed section. Furthermore let A be a set of assumptions about the bracketed sections occurring in S. Then, $A \vdash^{-} \{p\}S\{q\}$ iff there exist for every stable substatement R of S local assertions $Pre(R)$ and $Post(R)$ such that:

- $\{Pre(R)\}R\{Post(R)\} \in A$, for R a bracketed section
- $p \rightarrow Pre(S)$, $Post(S) \rightarrow q$
- $Pre(R) \rightarrow Post(R)[e/x]$, $R = x := e$
- $Pre(R) \rightarrow Pre(R_1)$, $Post(R_1) \rightarrow Pre(R_2)$, and $Post(R_2) \rightarrow Post(R)$, $R = R_1 ; R_2$
- $Pre(R) \wedge e \rightarrow Pre(R_1)$, $Pre(R) \wedge \neg e \rightarrow Pre(R_2)$, $Post(R_1) \rightarrow Post(R)$, and $Post(R_2) \rightarrow Post(R)$, $R = \textsf{if } e \textsf{ then } R_1 \textsf{ else } R_2 \textsf{ fi}$
- $Pre(R) \wedge e \rightarrow Pre(R_1)$, $Post(R_1) \rightarrow Pre(R)$, and $Pre(R) \wedge \neg e \rightarrow Post(R)$, $R = \textsf{while } e \textsf{ do } R_1 \textsf{ od}$

Proof. Straightforward induction on the structure of S. □

The soundness of the rule (PR) is implied by the following theorem.

Theorem 7.10.
Let $\rho = \langle c_1 \leftarrow S_1, \ldots, c_n \leftarrow S_n \rangle$ be bracketed and

$$A_i = \{\{Pre(R)\}R\{Post(R)\} \mid R \text{ a bracketed section occurring in } S_i\}$$

be such that the local proofs $A_i \vdash^{-} \{p_i\}S_i\{q_i\}$ cooperate with respect to the global invariant I.

Next let $(X, \theta, \lambda) \rightarrow (X', \sigma, h)$ be a global computation such that $Init_\rho(X)$, and for every $\alpha \in \sigma^{(c_i)}$ $(1 \leq i \leq n)$ we have that $X'(\alpha)$ equals $Before(R, S_i)$ or $After(R, S_i)$, with R a bracketed section, or $X'(\alpha) = E$. Finally, assume that: $\theta, \omega \models p_n$. Then:

1. $\sigma, \omega \models I$
2. For every $\alpha \in \sigma^{(c_i)}$ $(1 \leq i \leq n)$ we have:
   - if $X'(\alpha) = Before(R, S_i)$ then $\langle \alpha, \sigma'(\alpha) \rangle, \omega \models Pre(R)$,
   - if $X'(\alpha) = After(R, S_i)$ then $\langle \alpha, \sigma'(\alpha) \rangle, \omega \models Post(R)$,
   - if $X'(\alpha) = E$ then $\langle \alpha, \sigma'(\alpha) \rangle, \omega \models q_i$.

Proof. The proof proceeds by induction on $|h|$, i.e. the length of the history h.
$|h| = 0$:

Because $\theta, \omega \models p_n$, and by the cooperation test we are given the validity of

$$\exists z_n (p_n \downarrow z_n \wedge \forall z_n' (z_n' \doteq z_n) \wedge \bigwedge_{1 \leq i < n} (\forall z_i \textsf{false})) \rightarrow I$$

we have that $\theta, \omega \models I$ (with θ viewed as a global state where only the root-object exists). Now σ agrees with θ with respect to the variables occurring in I (the computation consists solely of a local computation of the root-object which does

not involve the creation of objects or communications and as such it is guaranteed not to affect the instance variables of I), so $\sigma, \omega \models I$. The remaining part of this case then proceeds by a straightforward induction on the length of the local computation of the root-object using lemma 7.9.

Next we consider the case $|h| > 0$:
First, let $h = h' \bullet < \alpha, \beta >$, with $\alpha \in O^{c_i}, \beta \in O^{c_j}$. We may assume without loss of generality that we can decompose the computation into $(X, \theta, \lambda) \to (X'', \sigma', h')$ and $(X'', \sigma', h') \to (X', \sigma, h)$ such that the latter computation starts with the creation of β by α and proceeds with the local computations of α and β. Of course this can be proved formally. However a formal proof being straightforward but rather tedious notationally we think we are justified in giving only the following informal explanation: Consider the moment of the computation that the object α is about to enter the bracketed section execution of which consists of the creation of β. From that moment on all objects execute independently from each other, which implies that from here on we can sequentialise the local computations in an arbitrary way.

Let for $c_k \in \{c_1, \ldots, c_n\}$, $\gamma(\neq \alpha, \beta) \in \mathbf{O}^{c_k}$, $X'(\gamma) = X''(\gamma) = Before(R', S_k)$, for some bracketed section R'. From the induction hypothesis we know that $\langle \gamma, \sigma'(\gamma) \rangle, \omega \models Pre(R')$. But $\sigma'(\gamma) = \sigma(\gamma)$, so $\langle \gamma, \sigma(\gamma) \rangle, \omega \models Pre(R')$. Analogously for $X''(\gamma) = X'(\gamma) = After(R', S_k), E$ (R' a bracketed section).

Furthermore it follows by the induction hypothesis (and lemma 6.7) that

$$\sigma', \omega\{\alpha/z\} \models (I \wedge Pre(R) \downarrow z)$$

where $X''(\alpha) = Before(R, S_i)$, with R the bracketed section which gives rise to the creation of β by α. Here $z \in LVar_{c_i}$ is a fresh variable. We are given that (by the soundness of the intermediate proof system)

$$\models \{I \wedge Pre(R) \downarrow z\}(z, R)\{I \wedge Post(R) \downarrow z\}$$

Let s be the local state of α just after the execution of the bracketed section R. We have that

$$\sigma'\{s/\alpha, \nabla/\beta\} \in \mathscr{I}[\![(z, R)]\!](\omega\{\alpha/z\})(\sigma').$$

So we may infer that

$$\sigma'\{s/\alpha, \nabla/\beta\}, \omega\{\alpha/z\} \models (I \wedge Post(R) \downarrow z)$$

As z does not occur in I and σ agrees with $\sigma'\{s/\alpha, \nabla/\beta\}$ with respect to the variables occurring in I (note that the local computations of α and β do not affect the variables of I) we have that $\sigma, \omega \models I$.

Since $\langle \alpha, s \rangle, \omega \models Post(R)$ and $\langle \beta, \nabla \rangle, \omega \models p_j$ (note that we are given the validity of the assertion $\bigwedge_{x \in IVar(S_j)}(x \doteq \mathsf{nil}) \to p_j$) we then can proceed by a straightforward induction on the length of the local computation of α and β, respectively, using lemma 7.9.

Finally, let $h = h' \bullet < \alpha, \beta, \gamma >$, where, say, $\alpha \in O^{c_i}, \beta \in O^{c_j}$: Again, we may assume without loss of generality that we can decompose the computation $(X, \theta, \lambda) \to (X', \sigma, h)$ into $(X, \theta, \lambda) \to (X'', \sigma', h')$ and $(X'', \sigma', h') \to (X', \sigma, h)$ such that the latter computation starts with the communication between α and β and proceeds with the local computations of α and β.

For $\gamma' \neq \alpha, \beta$ the desired result follows from the induction hypothesis as in the previous case. Let $X''(\alpha) = Before(R, S_i)$ and $X''(\beta) = Before(R', S_j)$, where R and R' are the bracketed sections which give rise to the communication $< \alpha, \beta, \gamma >$. By the induction hypothesis it then follows that

$\sigma', \omega \models I$
$\langle \alpha, \sigma'(\alpha) \rangle, \omega \models Pre(R)$ and
$\langle \beta, \sigma'(\beta) \rangle, \omega \models Pre(R')$

Let $z \in LVar_{c_i}, z' \in LVar_{c_j}$ be fresh variables. It then follows that

$$\sigma', \omega\{\alpha, \beta / z, z'\} \models (I \wedge Pre(R) \downarrow z \wedge Pre(R') \downarrow z')$$

We are given the validity of the formula

$$\{I \wedge Pre(R) \downarrow z \wedge Pre(R') \downarrow z'\}$$
$$(z, R) \parallel (z', R')$$
$$\{I \wedge Post(R) \downarrow z \wedge Post(R') \downarrow z'\}$$

Let s and s' be the local states of α and β just after the execution of the bracketed sections R and R'. It then follows that

$$\sigma'\{s, s'/\alpha, \beta\} \in \mathscr{I}[\![(z, R) \parallel (z', R')]\!](\omega\{\alpha, \beta/z, z'\})(\sigma')$$

From this we derive that $\sigma'\{s, s'/\alpha, \beta\}, \omega \models I$ (so, since the local computations of α and β do not affect the variables of I, we have $\sigma, \omega \models I$), $\langle \alpha, s \rangle, \omega \models Post(R)$, and $\langle \beta, s' \rangle, \omega \models Post(R')$. Finally, we proceed by a straightforward induction on the length of the local computation of α and β, respectively, using lemma 7.9. □

8. Completeness

In this section we prove the completeness of the proof system, that is, we prove that an arbitrary valid global correctness formula is derivable. We start with an outline of the approach followed. Let $\{p\}\rho\{Q\}$ be a valid correctness formula. The *strongest postcondition* of a program ρ with respect to a precondition p, notation $SP(p, \rho)$, is defined as follows:

$$\sigma, \omega \models SP(p, \rho) \text{ iff } \exists \theta : \theta, \omega \models p \text{ and } \sigma \in \mathscr{P}[\![\rho]\!](\theta)$$

It follows that $\models \{p\}\rho\{SP(p, \rho)\}$ and for every valid correctness formula $\{p\}\rho\{Q\}$ we have $\models SP(p, \rho) \rightarrow Q$. Thus to prove the derivability of any valid correctness formula $\{p\}\rho\{Q\}$, it would be sufficient to show the expressibility of $SP(p, \rho)$ and the derivability of $\{p\}\rho\{SP(p, \rho)\}$. We consider the derivability of $\{p\}\rho\{SP(p, \rho)$. Let $\rho = \langle c_1 \leftarrow S_1, \ldots, c_n \leftarrow S_n \rangle$. Consider the local correctness formulas $\{p_i\}S_i\{SP(p_i, S_i)\}$, where $p_i = \bigwedge_{x \in IVar(S_i)} (x \doteq \mathsf{nil})$, for $1 \leq i < n$, $p_n = p$, and

$$\theta, \omega \models SP(p, S) \text{ iff } \exists \theta' : \theta' \models p \text{ and } \theta \in \mathscr{S}[\![S]\!](\theta')$$

Assuming the derivability of $\{p_i\}S_i\{SP(p_i, S_i)\}$ the rule (PR) would allow to derive

$$\{p\}\rho\{\bigwedge_i \forall z_i\, (SP(p_i, S_i) \downarrow z_i)\}$$

Thus it would be sufficient to show the validity of $\bigwedge_i \forall z_i\, (SP(p_i, S_i) \downarrow z_i) \rightarrow SP(p, \rho)$. Let $\sigma, \omega \models \bigwedge_i \forall z_i\, (SP(p_i, S_i) \downarrow z_i)$. It follows that for every $\alpha \in \sigma^{(c_i)}$ $(1 \leq i \leq n)$ there exists a local computation $(S_i, \theta_\alpha, \lambda) \rightarrow (E, \langle \alpha, \sigma(\alpha) \rangle, h_\alpha)$, where $\theta_\alpha, \omega \models p$, if α equals the root-object, and $\theta_\alpha = \langle \alpha, \nabla \rangle$, otherwise. However from this we cannot conclude that there exists a computation of ρ resulting in σ

because the local histories h_α can be incompatible in the sense that there does not exist a global history h such that for all existing α we have $h_\alpha \leq [h]_\alpha$. To be able to express this additional information we incorporate the local histories in the local state as auxiliary variables. We then introduce a global invariant I to express the compatibility of the local histories.

Given the above outline we will first discuss completeness of the local proof system. In the second subsection we introduce local histories in the programming language and in the assertion languages. Then we show completeness by showing that the proofs of the local correctness formulas $\{p_i\}S_i\{SP(p_i, S_i)\}$ cooperate with respect to the global invariant as informally described above. In section 9 we discuss the expressibility of the local assertions $SP(p_i, S_i)$ and the global invariant.

Without loss of generality we may assume that the sets C and $IVar$ are finite.

8.1. Completeness of the Local Proof System

In this subsection we prove the completeness of the local proof system, i.e. the derivability of any valid local correctness formula $\{p\}S\{q\}$. Now it is well-known that the local proof system is complete for correctness formulas of statements in which there occur no I/O or new-statements. We want to generalise this result to arbitrary statements. Since the local proof system only allows one to reason about I/O or new-statements in terms of assumptions, completeness of the local proof system amounts to proving the derivability of a valid correctness formula $\{p\}S\{q\}$, where every I/O and new-statement of S occurs in a bracketed section, from a set of assumptions about the bracketed sections of S. Of course in principle we could simply introduce all the valid correctness formulas of bracketed sections as assumptions, however since we are interested in constructing proofs we require the set of assumptions to be recursive.

Definition 8.1. Let $Pre(R) = \bigwedge_{x \in IVar}(x \doteq z_x)$ for any bracketed section R. We define $Post(R) = SP(Pre(R), R)$. Here z_x is a logical variable uniquely associated with the variable x.

The variables z_x are used to freeze the initial values of their corresponding variables x. In section 9 we show how to express in the local assertion language the strongest postcondition of a statement R with respect to a precondition p. The formula $\{Pre(R)\}R\{Post(R)\}$ we call the strongest correctness formula of R. Note that we have $\models \{Pre(R)\}R\{Post(R)\}$, for any R. More interestingly the assertion $Post(R)$ describes the semantics of R in the following sense: for any local states θ and θ' we have $\theta' \in \mathscr{S}[\![R]\!](\theta)$ iff $\theta', \omega \models Post(R)$, where ω assigns to every logical variable z_x the value of x in θ.

Now we have the following completeness result for the local proof system.

Theorem 8.2.
Let A denote the set of correctness formulas $\{Pre(R)\}R\{Post(R)\}$, with R a bracketed section of S; then we have for any valid formula $\{p\}S\{q\}$

$$A \vdash \{p\}S\{q\}$$

Proof. The proof proceeds by induction on the structure of S. We treat the case that S is a bracketed section, all other cases are treated as in the standard completeness proof of the local proof system for statements without I/O or new-statements. Let R be a bracketed section for which the formula $\{p\}R\{q\}$ is valid. Without loss of generality we may assume that the variables z_x do not occur in p and q (otherwise substitute fresh logical variables for the variables z_x in p and q, follow the proof below for the resulting correctness formula, and conclude with an application of the substitution rule to return to the original assertions p and q). We have the following instance of the invariance axiom

$$\{p'\}R\{p'\}$$

where p' results from p by substituting every variable x by its corresponding variable z_x. By the conjunction rule we derive from $\{Pre(R)\}R\{Post(R)\}$ and the above instance of the invariance axiom the formula

$$\{Pre(R) \wedge p'\}R\{Post(R) \wedge p'\}$$

Now it is not difficult to check that the validity of the correctness formula $\{p\}R\{q\}$ implies the validity of the assertion $Post(R) \wedge p' \rightarrow q$. Thus an application of the consequence rule gives the formula

$$\{Pre(R) \wedge p'\}R\{q\}$$

Then we proceed by an application of the (local) substitution rule, substituting x for z_x, which gives us after a trivial application of the consequence rule our desired conclusion:

$$\{p\}R\{q\}$$

It is worthwhile to note that the above proof follows basically the structure of the completeness proof for recursive procedures as described in, for example, [Apt81].
□

8.2. Histories

As explained above we want to modify a program ρ by adding to it assignments to so-called *history* variables, i.e., auxiliary variables which record for every object its history, the sequence of communication records and activation records the object has participated in. In order to be able to compute the local history of an object we extend our programming language. We do so by introducing instance variables ranging over sequences. Thus we have now also for sequence types d^* a set of instance variables $IVar_{d^*}$. We redefine now the set of expressions Exp^c_a, with typical element e.

Definition 8.3. The new definition for the set of expression is as follows:

$e(\in Exp^c_a)$	$::=$	x	if $x \in IVar_a$
	$\mid$	self	if $d = c$
	$\mid$	nil	
	$\mid$	n	
	$\mid$	true $\mid$ false	if $d =$ Bool
	$\mid$	$\lvert e\rvert$	if $e \in Exp^c_{d*}$, $a =$ Int
	$\mid$	$e_1 \circ e_2$	if $e_1, e_2 \in Exp^c_{d*}$, $a = d^*$
	$\mid$	$\langle e\rangle$	if $e \in Exp^c_d$, $a = d^*$
	$\mid$	$e_1 + e_2$	if $e_1, e_2 \in Exp^c_d$, $d =$ Int
		$\vdots$	
	$\mid$	$e_1 \doteq e_2$	if $e_1, e_2 \in Exp^c_d$, $a =$ Bool

The expression $e_1 \circ e_2$ denotes the concatenation of the sequences e_1 and e_2. The sequence consisting of the element e is denoted by $\langle e\rangle$.

The set of statements $STat^c$ is defined as before. Assignment statements now additionally may have the form $x := e$ (where $x, e \in Exp^c_{d^*}$). Programs are defined as before.

Definition 8.4. The set of local expressions $LExp^c_a$ is extended as follows.

$l(\in LExp^c_a)$	$::=$	z	if $z \in LVar_a$
	$\mid$	x	if $x \in IVar_a$
	$\mid$	self	if $a = c$
	$\mid$	nil	
	$\mid$	n	if $a =$ Int
	$\mid$	true $\mid$ false	if $a =$ Bool
	$\mid$	$l_1 : l_2$	if $l_1 \in LExp^c_{d^*}$, $l_2 \in LExp^c_{\mathsf{Int}}$, $a = d$
	$\mid$	$\lvert l\rvert$	if $l \in LExp^c_{d^*}$
	$\mid$	$\langle l\rangle$	if $l \in LExp^c_d$, $a = d^*$
	$\mid$	$l_1 \circ l_2$	if $l_1, l_2 \in LExp^c_{d^*}$, $a = d^*$
	$\mid$	$l_1 + l_2$	if $l_1, l_2 \in LExp^c_{\mathsf{Int}}$
		$\vdots$	
	$\mid$	$l_1 \doteq l_2$	if $l_1, l_2 \in LExp^c_d$, $a =$ Bool

Local assertions are defined as before.

As we do not want to redefine our substitution operations we do *not* change our global assertion language but for allowing instance variables ranging over finite sequences. As a consequence we have to redefine the definition of the transformation of a local expression, local assertion to a global expression, global assertion, respectively. We do so by viewing the global expressions $\langle g\rangle$, $g_1 \circ g_2$ as abbreviations in the following sense: Suppose the expression $\langle g\rangle$ occurs in the assertion P. Let P' be such that $P'[\langle g\rangle/z] = P$. Then we can view P as an abbreviation of the assertion

$$\exists z(\lvert z\rvert \doteq 1 \wedge z : 1 \doteq g \wedge P')$$

In case an expression of the form $g_1 \circ g_2$ occurs in P, let P' now be such that $P'[g_1 \circ g_2/z] = P$. Then we can view P as an abbreviation of the assertion

$$\begin{array}{l} \exists z(|z| \doteq |g_1| + |g_2| \wedge \\ \quad \forall i(1 \leq i \leq |g_1| \rightarrow z : i \doteq g_1 : i) \wedge \\ \quad \forall i(|g_1|| < i \leq |z| \rightarrow z : i \doteq g_2 : i) \wedge \\ \quad P') \end{array}$$

We can define now $l \downarrow g$ simply as follows:

$$\begin{array}{lcl} z \downarrow g & = & z \\ x \downarrow g & = & g.x \\ \mathsf{self} \downarrow g & = & g \\ \langle l \rangle \downarrow g & = & \langle l \downarrow g \rangle \\ l_1 \circ l_2 \downarrow g & = & l_1 \downarrow g \circ l_2 \downarrow g \\ & \dots & \end{array}$$

The transformation $p \downarrow g$ is defined accordingly.

Note that the resulting assertion, in the case that p contains these newly introduced expressions, really is an abbreviation of an assertion.

Next we show how to extend a given program with auxiliary variables such that the resulting program additionally computes the local history of each object. Unfortunately we cannot simply represent a history by a sequence variable, since sequence variables range over sequences of objects of a particular type. However we can encode a local history using for each class c variables $out_c, in_c, act_c \in IVar_{c^*}$, where out_c records the sequence of objects of class c to which an object has been sent, in_c records the sequence of objects of class c from which an object has been received, and act_c records the sequence of objects of class c that have been created. Furthermore we need sequence variables for each type d which record the values sent, received, respectively. The details are left to the reader. In the following we will simply represent the data structure used to encode the local history by a 'virtual' sequence variable t which ranges over sequences of objects not necessarily of the same type. Updates to the data structure which model the updates to the local history corresponding to a communication or activation will be represented by the corresponding assignments to the virtual variable t. Thus we have the following definition:

Definition 8.5. First we transform an arbitrary I/O statement and new statement into a bracketed section which includes a corresponding update to the local history as represented by t:

$$\begin{array}{lcl} x?y & \Rightarrow & \langle x?y; t := t \circ \langle x, \mathsf{self}, y \rangle \rangle \\ x!e & \Rightarrow & \langle x!e; t := t \circ \langle \mathsf{self}, x, e \rangle \rangle \\ ?y & \Rightarrow & \langle ?y; t := t \circ \langle \mathsf{nil}, \mathsf{self}, y \rangle \rangle \\ x := \mathsf{new} & \Rightarrow & \langle x := \mathsf{new}; t := t \circ \langle \mathsf{self}, \mathsf{nil}, x \rangle \rangle \end{array}$$

Note that we model an activation by a communication to an unknown receiver. Since in communication records the receiver is always known we thus can retrieve from a sequence of objects the corresponding history. For example the sequence $\langle \alpha, \mathsf{nil}, \beta, \alpha', \beta', \gamma' \rangle$, where $\beta' \neq \mathsf{nil}$ represents the history $< \alpha, \beta >, < \alpha', \beta', \gamma' >$.

Let for an arbitrary unbracketed statement S the result of applying the above transformation, defined by induction on the structure of S, be denoted by $[S]$. Given a program $\rho = \langle \leftarrow S_1, \ldots, c_n \leftarrow S_n \rangle$, we put $[\rho] = \langle c_1 \leftarrow [S_1], \ldots, c_n \leftarrow [S_n] \rangle$.

8.3. Completeness of the Global Proof System

After having gone through the preparatory work of the previous subsections we are now ready to prove completeness of the global proof system. First we observe that the proofs $A \vdash \{p\}S\{SP(p,S)\}$, for *any* bracketed S and precondition p, with A the set of correctness formulas $\{Pre(R)\}R\{Post(R)\}$, R a bracketed section of S, *do* cooperate with respect to the empty global invariant $I = \mathsf{true}$ (which we leave the involved reader to verify)! So in fact the global invariant is only used to strengthen the postcondition of the conclusion of the rule (PR), and the proof obligations of the cooperation test are used to establish the truth of the global invariant on termination. In this respect our approach differs from the standard completeness proof given in [AFR80] for CSP where the global invariant is necessary for the cooperation test. This difference is essentially due to the fact that our local correctness proofs $A \vdash \{p\}S\{SP(p,S)\}$ are *modular* in the sense that they can be used in different programs, whereas in [AFR80] the local proofs encode information of the specific structure of the program given. In a similar way our assumptions are modular too, i.e., they can be used in different statements, whereas in [AFR80] the assumptions encode information about the specific structure of the entire local statement in which the bracketed sections occur.

We will now give the formal definition of the global invariant I.

Lemma 8.6.
There exists a global assertion I such that $\sigma, \omega \models I$ iff there exists a global history h such that h is compatible with σ (formally, $OK(\sigma, h)$), and for all existing α we have $\sigma(\alpha)(t) \leq [h]_\alpha$.

Proof. See section 9. □

We next show that the proofs $A \vdash \{p\}[S]\{SP(p,[S])\}$, for *any* (unbracketed) statement S (such that no sequence variables occur in S) and precondition p, with A the set of formulas $\{Pre(R)\}R\{Post(R)\}$, R a bracketed section of $[S]$, cooperate with respect to the global invariant I as defined as above. Actually, it only requires a straightforward 'unravelling' of the definitions to see that the cooperation test holds trivially.

First we show how to discharge assumptions about bracketed sections containing a new-statement.

Lemma 8.7.
Let $R = x := \mathsf{new}; t := t \circ \langle \mathsf{self}, \mathsf{nil}, x \rangle$ be a bracketed section occurring in $[S]$ ($S \in STat^c$), then:

$$\vdash \{I \wedge Pre(R) \downarrow z\}(z, R)\{I \wedge Post(R) \downarrow z\}$$

where $z \in LVar_c$ is a new variable.

Proof. By the axioms (IASS) and (NEW) it suffices to show that:

$$\models I \wedge Pre(R) \downarrow z \rightarrow (I \wedge Post(R) \downarrow z)[z.t \circ \langle z, \mathsf{nil}, z.x \rangle / z.t][z'/z.x][\mathsf{new}/z']$$

where $z' \in LVar_{c_j}$ is a new variable. It is worthwhile to notice that the substitution $[z.t \circ \langle z, \mathsf{nil}, z.x\rangle / z.t]$ actually represents a corresponding multiple substitution to the sequence variables used to encode the local history.

Let $\sigma, \omega \models I \wedge Pre(R) \downarrow z$ and $\sigma' \in \mathscr{I}[\![(z,R)]\!](\omega)(\sigma)$. It is easy to check that $\sigma', \omega \models I$. Furthermore from $\sigma' \in \mathscr{I}[\![(z,R)]\!](\omega)(\sigma)$ it follows that $\langle \alpha, \sigma'(\alpha)\rangle \in \mathscr{S}[\![R]\!](\langle \alpha, \sigma(\alpha)\rangle)$, where $\alpha = \omega(z)$. From $\sigma, \omega \models Pre(R) \downarrow z$ we subsequently infer by lemma 6.7 that $\langle \alpha, \sigma(\alpha)\rangle, \omega \models Pre(R)$. By the definition of $Post(R)$ we thus derive that $\langle \alpha, \sigma'(\alpha)\rangle, \omega \models Post(R)$, or, in other words, $\sigma', \omega \models Post(R) \downarrow z$.

Summarizing we have

$$\sigma', \omega \models (I \wedge Post(R) \downarrow z),$$

from which in turn it is not difficult to derive by an application of the lemmas 7.1 and 7.3 that:

$$\sigma, \omega \models (I \wedge Post(R) \downarrow z)[z.t \circ \langle z, \mathsf{nil}, z.x\rangle][z'/z.x][\mathsf{new}/z']$$

□

Next we show how to discharge assumptions about matching bracketed sections.

Lemma 8.8.
For two arbitrary matching bracketed sections R_1 and R_2, occurring in, say, $[S_1], [S_2]$ ($S_1 \in STat^c, S_2 \in STat^{c'}$), we have:

$$\begin{array}{ll} & \{I \wedge Pre(R_1) \downarrow z \wedge Pre(R_2) \downarrow z'\} \\ \vdash & \qquad (z, R_1) \parallel (z', R_2) \\ & \{I \wedge Post(R_1) \downarrow z \wedge Post(R_2) \downarrow z'\} \end{array}$$

where $z \in LVar_c, z' \in LVar_{c'}$ are two new distinct variables.

Proof. We prove the following case, the other ones are treated in a similar way: Let $R_1 = x!e; t := t \circ \langle \mathsf{self}, x, e\rangle$, $R_2 = ?y; t := t \circ \langle \mathsf{nil}, \mathsf{self}, y\rangle$, and $c \neq c'$. Let $[in]$, $[out]$ abbreviate the substitution $[z'.t \circ \langle \mathsf{nil}, z', z'.y\rangle / z'.t]$ and $[z.t \circ \langle z, z.x, e \downarrow z\rangle / z.t]$, respectively. (As above, note that actually these substitutions represent multiple substitutions to the sequence variables used to encode the local history).

By the axioms (COMM) and (IASS), the rules (SR2), (PAR1) and (PAR2) it suffices to show that:

$$\begin{array}{l} \models I \wedge Pre(R_1) \downarrow z \wedge Pre(R_2) \downarrow z' \wedge z.x = z' \wedge z.x \neq \mathsf{nil} \rightarrow \\ (I \wedge Post(R_1) \downarrow z \wedge Post(R_2) \downarrow z')[in][out][e \downarrow z / z'.y]. \end{array}$$

Let

$$\sigma, \omega \models (I \wedge Pre(R_1) \downarrow z \wedge Pre(R_2) \downarrow z' \wedge z.x = z' \wedge z.x \neq \mathsf{nil})$$

and

$$\sigma' \in \mathscr{I}[\![(z, R_1) \parallel (z', R_2)]\!](\omega)(\sigma)$$

(note that such σ' exists because $\sigma, \omega \models z.x = z' \wedge z.x \neq \mathsf{nil}$). Furthermore let $\alpha = \omega(z)$ and $\beta = \omega(z')$. It is easy to check that $\sigma', \omega \models I$. From $\sigma' \in \mathscr{I}[\![(z, R_1) \parallel (z', R_2)]\!](\omega)(\sigma)$ it follows that $\langle \alpha, \sigma'(\alpha)\rangle \in \mathscr{S}[\![R_1]\!](\langle \alpha, \sigma(\alpha)\rangle)$ and $\langle \beta, \sigma'(\beta)\rangle \in \mathscr{S}[\![R_2]\!](\langle \beta, \sigma(\beta)\rangle)$. From $\sigma, \omega \models Pre(R_1) \downarrow z \wedge Pre(R_2) \downarrow z'$ it follows by lemma 6.7 that $\langle \alpha, \sigma(\alpha)\rangle, \omega \models Pre(R_1)$ and $\langle \beta, \sigma'(\beta)\rangle, \omega \models Pre(R_2)$. By the

definition of $Post(R_1)$ ($Post(R_2)$) we then infer that $\langle\alpha,\sigma'(\alpha)\rangle,\omega \models Post(R_1)$ ($\langle\beta,\sigma'(\beta)\rangle,\omega \models Post(R_2)$), that is (by lemma 6.7), $\sigma',\omega \models Post(R_1) \downarrow z \wedge Post(R_2) \downarrow z'$.

Summarizing we have

$$\sigma',\omega \models (I \wedge Post(R_1) \downarrow z \wedge Post(R_2) \downarrow z').$$

From this in turn we derive by applying lemma 7.1 that

$$\sigma,\omega \models (I \wedge Post(R_1) \downarrow z \wedge Post(R_2) \downarrow z')[in][out][e \downarrow z/z'.y].$$

□

We are now ready for the completeness theorem.

Theorem 8.9.
Every valid correctness formula $\{p\}\rho\{Q\}$ in which there occur no sequence variables, is derivable:

$$\vdash \{p\}\rho\{Q\}.$$

Proof. Let $\{p\}\rho\{Q\}$ be a valid correctness formula, with $\rho = \langle c_1 \leftarrow S_1,\ldots,c_n \rightarrow S_n\rangle$. It is not so difficult to see that we may assume without loss of generality that $C = \{c_1,\ldots,c_n\}$ (note that by definition only variables of type $d \in \{c_1,\ldots,c_n,\mathsf{Int},\mathsf{Bool}\}$ are allowed to occur in p and Q). By lemma 8.2 we have

$$A_i \vdash \{p_i\}[S_i]\{SP(p_i,[S_i])\}.$$

where A_i denotes the set of formulas $\{Pre(R)\}R\{Post(R)\}$, R a bracketed section of $[S_i]$. furthermore, $p_n = p \wedge (t \doteq \mathsf{nil})$, and for $1 \leq i < n$ we have $p_i = \bigwedge_{x\in IVar}(x \doteq \mathsf{nil})$. Note that the precondition p_n of the root-object additionally initialises the local history.

It is not difficult to prove that

$$\models \exists z(p_n \downarrow z \wedge \forall z'(z \doteq z) \wedge \bigwedge_{1\leq i<n} (\forall z_i' \;\mathsf{false})) \rightarrow I.$$

(Here $z,z' \in LVar_{c_n}$ and $z_i' \in LVar_{c_i}$ are new variables. From this and the lemmas 8.7 and 8.8 it follows that $Coop(A_1,\ldots,A_n,I)$. Thus an application of the program rule (PR) gives the derivability of the formula

$$\{p_n\}\;[\rho]\;\{I \wedge \bigwedge_{1\leq i\leq n} \forall z_i\;(SP(p_i,[S_i]) \downarrow z_i)\}$$

Next we prove

$$\models I \wedge \bigwedge_{1\leq i\leq n} \forall z_i\;(SP(p_i,[S_i])) \rightarrow Q$$

Let the antecedent of the implication be true with respect to some logical environment ω and some global state σ. For $\alpha \in \sigma^{(c_i)}$ we have that $\langle\alpha,\sigma(\alpha)\rangle,\omega \models SP(p_i,[S'_i])$ implies the existence of a local computation

$$(S_i',\theta_\alpha,\lambda) \rightarrow^* (E,\langle\alpha,\sigma(\alpha)\rangle,h_\alpha)$$

for some local history h_α, where $\theta_\alpha,\omega \models p_n$, if α equals the root-object, and $\theta_\alpha = \langle\alpha,\nabla\rangle$, otherwise. Since $\sigma,\omega \models I$ there exists a global history h such that for any existing object α we have $h_\alpha = \sigma(\alpha)(t) \leq [h]_\alpha$. It then follows from the

definition of the semantics of global configurations that $(X, \theta, \lambda) \rightarrow (X', \sigma, h)$, for some θ such that $\theta, \omega \models p_n$, where $Init(X)$ and $Final_{[\rho]}(X', \sigma)$. In other words $\sigma \in \mathscr{P}[\![[\rho]]\!](\theta)$. Now the validity of the formula $\{p\}\rho\{Q\}$ implies the validity of $\{p_n\}[\rho]\{Q\}$, from which we conclude $\sigma, \omega \models Q$.

By the consequence rule we thus have

$$\vdash \{p_n\}[\rho]\{Q\}$$

Applying the substitution rule (S) (substituting nil for t) then gives us, after an trivial application of the consequence rule,

$$\vdash \{p\}[\rho]\{Q\}$$

An application of the rule (AUX) then finishes the proof. □

9. Expressibility

In this section we discuss how to express in the local assertion language the strongest postcondition of a statement S with respect to a precondition p, as defined in the section on the completeness of the proof system. The expressibility of the global invariant I is straightforward albeit rather tedious, and is left to the reader to check. We assume throughout this section the sets C and $IVar$ to be finite. This section presupposes some knowledge of general coding techniques to *arithmetise* the semantics of programs and assertions (see, for example, [TuZ88]).

The problem of expressing the strongest postcondition of a statement S with respect to a precondition p essentially boils down to showing how to express the semantics of a statement S. One of the main difficulties arises from the absence of general quantification in the local assertion language: we cannot store directly a local computation in logical variables (we have only quantification of logical variables ranging over integers and booleans in the local assertion language). Instead we have to arithmetise local computations. To this end we fix for an arbitrary d an injection $[\,]_d \in O^d_\perp \rightarrow N$ such that $[\perp]_d = 0$. Given these coding functions we can use standard coding techniques ([TuZ88]) to encode a state, a local configuration, and a sequence of local configurations. From such a code of a computation sequence then we can extract the code of the last state by some appropriate arithmetical operation, but then the problem arises how to express that this code corresponds with the code of a given local state. This problem arises because we cannot represent the coding functions $[\,]_c$, $c \in C$, more precisely we cannot assume the existence of a local assertion $[x]_c \doteq f_x(n)$, where $x \in IVar_c$ and $n \in LVar_{\mathsf{Int}}$, such that $\theta, \omega \models [x]_c \doteq f_x(n)$ iff $[\theta(x)]_c$ equals the encoded value of x retrieved from the code $\omega(n)$ of the local state θ by some arithmetical operation f_x. To see this we first introduce the notion of an *object-space isomorphism (osi)*:

Definition 9.1. An *object-space isomorphism* (*osi*) is a family of functions $f = \langle f^d \rangle_{d \in C^+}$, where $f^d \in \mathbf{O}^d_\perp \rightarrow \mathbf{O}^d_\perp$ is a bijection, $f^d(\perp) = \perp$ and f^d, for $d = \mathsf{Int}, \mathsf{Bool}$, is the identity mapping.

We have the following lemma which states that the assertion language cannot distinguish isomorphic states:

Lemma 9.2.
For any local state θ, logical environment ω, local assertion p, and *osi* f we have:

$$\theta, \omega \models p \text{ iff } f(\theta), f(\omega) \models p$$

(Here $f(\theta)$ and $f(\omega)$ are defined in the obvious way).

It should be clear that the above lemma implies that the coding functions $[\ ]_c$ cannot be represented in the assertion language.

However we can formulate a local assertion $p(n)$ such that whenever $\theta, \omega \models p(n)$ then there exists an object-space isomorphism between θ and θ', where θ' is encoded by $\omega(n)$. The basic idea is that instead of trying to relate immediately a local state and its code we introduce for each pair of variables x and y, of any type different from Int and Bool, the local assertion $x \doteq y \leftrightarrow f_x(n) \doteq f_y(n)$, where f_x and f_y denote the representation of some arithmetical operation which extracts from the integer value of n the encoded values of x and y, respectively. For the standard types Int and Bool on the other hand it is not difficult to see that we may assume the representability of the corresponding coding functions as indicated above. Let $lcode(n)$ be the conjunction of all these assertions then we can show that whenever $\theta, \omega \models lcode(n)$ then there exists an *osi* f such that $f(\theta) = \theta'$, where θ' is encoded by n.

The following theorem then justifies that indeed it is sufficient to establish an object-space isomorphism.

Theorem 9.3.
Let the object-space isomorphism f be given. Then whenever

$$(S, \theta, h) \rightarrow^* (S', \theta', h')$$

we have

$$(S, f(\theta), f(h)) \rightarrow^* (S', f(\theta'), f(h'))$$

(Here $f(h)$ denotes the history resulting from the pointwise application of f to h.)

10. Conclusion

We have developed a proof system for the partial correctness of programs of a parallel language with dynamic process creation. The basic ingredients for dealing with parallelism in this proof system are the same as in a proof system for CSP [AFR80], but they have been enhanced considerably to deal with the present, much more powerful programming language. One of the main problems we solved is how to reason about the dynamically evolving pointer structures that can arise during the execution of a program.

We have proved that the system is sound and complete. Our completeness proof differs considerably from the usual completeness proofs for proof systems based on the same methodology. One of the main difference consists of that our completeness proof is based on a *compositional* semantics. An interesting consequence of this is that one of the key ideas of the standard completeness proof, namely the *merging lemma* ([Apt83]) which states that under certain conditions local computations can be 'merged' into a global computation, in our approach derives immediately from the compositionality of the underlying semantics. Another striking difference consists in the definition of the specifications of the components of a system. Whereas usually these specifications are based on assertions which code computation sequences, our local proofs are based on the more abstract notion of the strongest postcondition. In this way we obtain a *modular* completeness proof in the sense that the specifications of the components

can be used in the context of any complete system of processes. Finally, a last interesting difference concerns our observation that local specifications based on the expressibility of the strongest postcondition do pass the cooperation test with the empty global invariant. In our approach a global invariant is only used to strengthen the conjunction of the strongest postconditions of the components, in order to imply any valid postcondition of the complete system. For this we showed that it is sufficient for the global invariant to express compatibility of the local computations in terms of the local histories.

Summarizing, our completeness proof can be characterised as being structured along the lines of the *compositional* proof theories for extensions of CSP as developed, for example, in [ZRE85]. Here the general reasoning pattern is not only based on *states*, as in our case, but also on *traces* (communication histories). Thus one is inclined to conclude that the so-called non-compositional proof method itself allows as a special case compositional reasoning!

We have already mentioned that our language and our proof techniques can be considered as very powerful extensions to CSP [Hoa78, AFR80, Apt83]. A different kind of extension, where processes can split themselves recursively into subprocesses, is dealt with in the above-mentioned [ZRE85], for example. In [Mel91], a language similar to ours is tackled with trace-based reasoning. The problem of dynamic pointer structures however is not dealt with explicitly.

Interesting future topics are extensions of the proof system to reason about properties other than partial correctness, for example, absence of deadlock. Another issue concerns the relation between the kind of language we studied and Milner's π-calculus.

Acknowledgements

We thank Jaco de Bakker, Arie de Bruin, Joost Kok, John-Jules Meyer, Jan Rutten and Erik de Vink, members of the Amsterdam Concurrency Group, for participating in discussions of preliminary versions of the proof system presented in this paper.

References

[Ame89] America, P. H. M.: Issues in the Design of a Parallel Object-Oriented Language. *Formal Aspects of Computing*, **1**, 366–411 (1989).

[ABK86] America, P. H. M., de Bakker, J. W., Kok, J. N. and Rutten, J. J. M. M.: Operational Semantics of a Parallel Object-Oriented Language. *Conference Record of the 13th Symposium on Principles of Programming Languages*, St. Petersburg, Florida, pp. 194–208, 1986.

[AFR80] Apt, K. R., Francez, N. and de Roever, W. P.: A Proof System for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, **2**, 359–385 (1980).

[Apt81] Apt, K. R.: Ten Years of Hoare logic: A Survey—Part I. *ACM Transactions on Programming Languages and Systems*, **3**, 431–483 (1981).

[Apt83] Apt, K. R.: Formal Justification of a Proof System for Communicating Sequential Processes. *Journal of the ACM*, **30**, 197–216 (1983).

[Bak80] de Bakker, J. W.: *Mathematical Theory of Program Correctness*. Prentice-Hall International, 1980.

[GoR84] Goldberg, A. and Robson, D.: *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, 1984.

[Hoa69] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. *Communications of the ACM*, **12**, 567–580,583 (1969).

[Hoa78] Hoare, C. A. R.: Communicating Sequential Processes. *Communications of the ACM*, **21**, 666–677 (1978).

[HoR86] Hooman, J. and de Roever, W. P.: The Quest Goes On: Towards Compositional Proof Systems for CSP. In J.W. de Bakker, W.P. de Roever, G. Rozenberg (eds.): *Current Trends in Concurrency*, Springer LNCS 224, pp. 343–395, 1986.

[Mel91] Meldal, S.: Axiomatic Semantics of Access Type Tasks in Ada. Report No. 100, Institute of Informatics, University of Oslo, Norway, May 1986. Appeared as "Complete Axiomatic Semantics of Spawning," *Distributed Computing*, **3**, 159–174 (1991).

[Mey88] Meyer, B.: *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[TuZ88] Tucker, J. V. and Zucker, J. I.: *Program Correctness over Abstract Data Types, with Error-State Semantics.* CWI Monographs 6, North-Holland, 1988.

[ZRE85] Zwiers, J., de Roever, W. P. and van Emde Boas, P.: Compositionality and concurrent networks: soundness and completeness of a proof system. *Proceedings of the 12th ICALP*, Nafplion, Greece, Springer LNCS 194, pp. 509–519, 1985.

Received February 1992
Accepted in revised form January 1993 by J. Parrow