A compositional proof system for dynamic proces creation

Eindhoven University of Technology

Department of Mathematics and Computing Science

A compositional proof system for dynamic proces creation

by

Frank de Boer

Computing Science Note 91/28
Eindhoven, September 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

A compositional proof system for dynamic process creation *

F. de Boer
Technical University Eindhoven
P.O. Box 513, 5600 MB Einhoven
The Netherlands
email:wsinfdb@info.win.tue.nl

Abstract We present a compositional proof system for a parallel language with dynamic process creation. We show how a dynamic system of processes can be described in terms of specifications of the local processes which involve a characterization of their interface with the environment. The proof system formalizes reasoning about these interfaces on an abstraction level that is at least as high as that of the programming language.

*An extended abstract of this paper appeared in the proceedings of LICS'91.

1 Introduction

The goal of this paper is to develop a compositional proof system for reasoning about the correctness of a certain class of parallel programs. 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 [Am]. POOL makes use of the structuring mechanisms of object-oriented programming [Mey], 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.

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 [AFR] and the tree-like structure in [ZREB].

In [AB] we developed for the language P a proof system based upon a generalization of the ideas underlying the proof theory of CSP ([AFR]). In that system the local behaviour of an object is specified with respect to *assumptions* about its environment. In what is called the *cooperation test* these assumptions associated with different objects have to be shown to be mutually consistent with respect to a *global invariant*, i.e., an assertion describing the global topology. A drawback of this methodology is that it provides no tools to understand and reason about a complete system in terms of its constituents.

In this paper we show how a complete system can be described in terms of its objects,

by viewing an object as an entity consisting of some internal data/activity, and an *interface* with its environment. Such an interface we model as a *history* of interactions of the object with its environment. This basic idea has been applied to a CSP-like language in [ZREB]. We generalize this idea to cope with dynamic process creation and dynamically evolving process structures. In our case an interaction will be either the creation of an object or a communication event. A specification then of a complete system can be obtained from specifications of its objects essentially by a conjunction and viewing the local history of an object as a subsequence of the global history of the system consisting of those interactions which involve the object.

Another important characteristic of our proof system is that it allows reasoning about histories and dynamically evolving *process 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.

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.

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.

Our paper is organized 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. The semantics of the programming language, the assertion languages, and that of correctness formulas is described in section 5. In section 6 we discuss soundness and completeness of the proof system. Finally, in section 7 we draw some conclusions.

2 The programming language

In this section we 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 $C \cup \{\mathsf{Int}, \mathsf{Bool}\}$ of *data types*, with typical element d, we denote by C^+. Here Int and Bool denote the types of the integers and booleans, respectively. For each $c \in C$ and $d \in C^+$ we assume $IVar_d^c$ to be the set of instance variables of type d in class c, with typical elements x_d^c and y_d^c. Such a variable x_d^c occurs in each object of class c and it can refer to objects of type d only. We assume that $IVar_d^c \cap IVar_{d'}^{c'} = \emptyset$ whenever $c \neq c'$ or $d \neq d'$. In cases where no confusion arises we omit the subscripts and superscripts.

Definition 2.1
We define the set Exp_d^c of expressions of type d in class c, with typical element e_d^c. Such an expression e_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:

$$
\begin{array}{rcll}
e_d^c & ::= & x_d^c & \\
 & | & \mathsf{self} & \text{if } d = c \\
 & | & \mathsf{nil} & \\
 & | & \mathsf{true} \mid \mathsf{false} & \text{if } d = \mathsf{Bool} \\
 & | & n & \text{if } d = \mathsf{Int} \\
 & | & e_1{}^c + e_2{}^c & \text{if } d = \mathsf{Int} \\
 & \vdots & & \\
 & | & e_{1\,d'}^c \doteq e_{2\,d'}^c & \text{if } d = \mathsf{Bool}
\end{array}
$$

An expression e_d^c 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. The intended meaning of the other expressions we assume to be self-evident. 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^c. These statements can be executed by an object of class c.

Statements can be of the following forms:

$$
\begin{array}{rcll}
S^c & ::= & x_d^c := e_d^c & \\
 & | & x_d^c := \mathsf{new}_d & \text{if } d \neq \mathsf{Int}, \mathsf{Bool} \\
 & | & x_{c'}^c ! e_d^c & \\
 & | & x_{c'}^c ? y_d^c & \\
 & | & ?y_d^c & \\
 & | & S_1^c ; S_2^c & \\
 & | & \mathsf{if}\ e^c\ \mathsf{then}\ S_1^c\ \mathsf{else}\ S_2^c\ \mathsf{fi} & \\
 & | & \mathsf{while}\ e^c\ \mathsf{do}\ S^c\ \mathsf{od} &
\end{array}
$$

A statement S^c can be executed by an object of class c. The 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 $x := \mathsf{new}_d$ by the object α consists of creating a new object β of class d and making the variable x of the creator α refer to it. The instance variables of the new object β are initialized 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 $x_{c'}^c ! e_d^c$ is called an *output* statement and statements like $x_{c'}^c ? y_d^c$ and $?y_d^c$ are called *input* statements. Together they are called I/O statements. The execution of an output statement $x_{1\,c'}^{\ c} ! e_d^c$ by an object α is always synchronized with the execution of a corresponding input statement $x_{2\,c}^{\ c'} ? y_d^{c'}$ or $?y_d^{c'}$ 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 c' of β).
- If the input statement to be executed is of the form $x_{2\,c}^{\ c'} ? y_d^{c'}$, 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 c of α).
- The type d of the expression e_d^c in the output statements should coincide with the type of the destination variable $y_d^{c'}$ 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 $Prog^{c_n}$ of *programs*, with typical element ρ^{c_n}, as follows:

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

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

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

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 global proof system.

3.1 The local assertion language

We introduce a new kind of variables: Let $LogVar_d$ be an infinite set of *logical variables* of type d, with typical element z_d. We assume that these sets are disjoint from the other sets of syntactic entities. Logical variables do not occur in a program, but only in assertions.

Definition 3.1
The set $LExp_d^c$ of *local expressions* of type d in class c, with typical element l_d^c, is defined as the set Exp_d^c but for the additional rule: $l_d^c ::= z_d$.

The internal behaviour of an object will be specified with respect to a *local history*, which records the sequence of interactions of the object with its environment. Such a local history can also be understood as describing the *interface* of an object. To reason about histories we introduce a new kind of variables: Let $HistVar_t$ be an infinite set of variables of type t, with typical element z_t, where t denotes the type of histories.

Definition 3.2
The set $LHist^c$ of *local history expressions*, with typical element lh^c, is defined as follows:

$$\begin{array}{rcl} lh^c & ::= & h^c \\ & | & z_t \\ & | & \epsilon \\ & | & \langle \mathsf{self}, l_{c'}^c \rangle \\ & | & \langle l_{1\,c'}^c, \mathsf{self}, l_{2\,d}^c \rangle \\ & | & \langle \mathsf{self}, l_{1\,c'}^c, l_{2\,d}^c \rangle \\ & | & lh_1^c \circ lh_2^c \end{array}$$

The local history of an object of class c is represented by the expression h^c. The empty history is denoted by ϵ. The expression $\langle \mathsf{self}, l_{c'}^c \rangle$ denotes a history consisting

of one *creation record* which encodes the information that the object $l^c_{c'}$ has been created. The expression $\langle l_1{}^c_{c'}, \mathsf{self}, l_2{}^c_d \rangle$ denotes a history consisting of one *communication record* which encodes the information that the object $l_2{}^c_d$ has been received from $l_1{}^c_{c'}$. Analogously, the expression $\langle \mathsf{self}, l_1{}^c_{c'}, l_2{}^c_d$ denotes a history consisting of one record which encodes the information that the object $l_2{}^c_d$ has been sent to $l_1{}^c_{c'}$. Finally, the expression $lh^c_1 \circ lh^c_2$ is interpreted as the history consisting of the history lh^c_1 followed by lh^c_2.

Definition 3.3
The set $LAss^c$ of *local assertions* in class c, with typical element p^c, is defined as follows:

$$\begin{array}{rcll} p^c & ::= & l^c_d & \text{if } d = \mathsf{Bool} \\ & | & lh^c_1 \doteq lh^c_2 & \\ & | & \neg p^c & \\ & | & p^c_1 \wedge p^c_2 & \\ & | & \exists z\, p^c & \end{array}$$

We shall regard other logical connectives $(\vee, \rightarrow, \forall)$ as abbreviations for combinations of the above ones.

Local expressions l^c_d, local history expressions lh^c and local assertions p^c are evaluated with respect to the local state of an object of class c, determining the values of its instance variables and the local history h^c, plus a logical environment, which assigns values to the logical variables and the history variables (distinct from h^c). Therefore they talk about this single object in isolation. In the local assertion $\exists z\, p^c$ the variable z can be any history variable distinct from h^c or logical variable. Quantification is interpreted as usual. More specifically, quantification over objects of some class c is interpreted as ranging over all the possible objects in that class, i.e., including the objects which have not yet been created.

3.2 The global assertion language

Next we define the global assertion language. 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 $LogVar_{d^*}$ of logical variables of type d^*, which range over finite sequences of elements of type d. Therefore we now have a set $LogVar_a$ of logical variables of type a for every $a \in C^\dagger$.

Definition 3.4
We give the following typical rules characterizing the set $GExp_a^c$ of *global expressions* of type a, with typical element g_a^c:

$$\begin{array}{rcll} g_a^c & ::= & z_a & \\ & | & \mathsf{root}_a & \text{if } a = c \\ & | & g_{c'}^c.x_d^{c'} & \text{if } a = d \\ & | & |g_{d^*}^c| & \text{if } a = \mathsf{Int} \\ & | & g_{d^*}^c : g_{d'}^c & \text{if } a = d,\ d' = \mathsf{Int} \end{array}$$

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, a root object, and a global history. The expression root_c denotes the root object. 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. $|g|$ denotes the length of the sequence denoted by g. The expression $g_1 : g_2$ denotes the nth element of the sequence denoted by g_1, where n is the value of g_2 (if g_2 is less than 1 or greater than $|g_1|$, the result is nil.)

Definition 3.5
The set $GHist^c$ of *global history expressions*, with typical element gh^c, is defined as follows:

$$\begin{array}{rcl} gh^c & ::= & h \\ & | & z_t \\ & | & \epsilon \\ & | & \langle g1_{c_1}^c, g2_{c_2}^c \rangle \\ & | & \langle g1_{c_1}^c, g2_{c_2}^c, g3_d^c \rangle \\ & | & gh^c | g_{c'}^c \\ & | & gh_1^c \circ gh_2^c \end{array}$$

The expression h denotes the global history of a complete set of objects. Global history expressions are introduced to reason about the global history. The expression $\langle g1_{c_1}, g2_{c_2} \rangle$ denotes the history consisting of one creation record which encodes that $g1_{c_1}$ has created $g2_{c_2}$. The expression $\langle g1_{c_1}, g2_{c_2}, g3_d \rangle$ denotes the history consisting of one communication record which encodes that $g1_{c_1}$, has sent to $g2_{c_2}$ the object $g3_d$. The subsequence of a history gh consisting of those communication and creation records which involve the object $g_{c'}$ is denoted by $gh|g_{c'}$. Finally, the expression $gh_1 \circ gh_2$ denotes the global history gh_1 followed by gh_2.

Definition 3.6
The set $GAss^c$ of *global assertions*, with typical element P^c, is defined as follows:

$$\begin{array}{rcll} P^c & ::= & g^c_d & \text{if } d = \mathsf{Bool} \\ & | & gh^c_1 \doteq gh^c_2 & \\ & | & \neg P^c & \\ & | & P^c_1 \wedge P^c_2 & \\ & | & \exists z\, P^c & \end{array}$$

In the global assertion $\exists z\, P$ the variable z can be any history variable distinct from h or logical variable. Again, other logical connectives are regarded as abbreviations.

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 $\exists z_c\, \mathsf{true}$ is false in some state iff there are no objects of class c in this state. Quantification over history variables is interpreted as ranging over finite sequences of interactions involving only existing objects.

Next we define a transformation of a local (history) expression or assertion to a global one. This transformation will be used to specify the global behaviour of a program in terms of the local behaviour of objects.

Definition 3.7
Given a local expression l^c_d and a global expression g_c we define a global expression $l^c_d \downarrow g_c$. 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. We give the following typical cases:

$$\begin{array}{lcl} x \downarrow g & = & g.x \\ \mathsf{self} \downarrow g & = & g \end{array}$$

For a local history expression lh^c we define the global history expression $lh^c \downarrow g_c$ as follows:

$$\begin{array}{lcll} (h^c) \downarrow g & = & h|g & \\ lh \downarrow g & = & lh & \text{if } lh = z, \epsilon \\ \langle \mathsf{self}, l \rangle \downarrow & = & \langle g, l \downarrow g \rangle & \\ \langle l_1, \mathsf{self}, l_2 \rangle & = & \langle l_1 \downarrow g, g, l_2 \downarrow g \rangle & \\ \langle \mathsf{self}, l_1, l_2 \rangle & = & \langle g, l_1 \downarrow g, l_2 \downarrow g \rangle & \\ (lh_1 \circ lh_2) \downarrow g & = & lh_1 \downarrow g \circ lh_2 \downarrow g \rangle & \end{array}$$

It is important to note that $(h^c) \downarrow g = h|g$ expresses that the local history of an object g can be obtained from the global history h by considering only those interactions involving g. Finally, for a local assertion p^c we define the global assertion $p^c \downarrow g_c$ as follows:

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

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 3.8
We define a *local correctness formula* to be of the following form:

$$\{p^c\} S^c \{q^c\}.$$

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.

Global correctness formulas describe a complete system:

Definition 3.9
A global correctness formula is of the form

$$\{p^c\} \rho^c \{Q^c\}$$

The precondition p^c describes the initial local state of the root object. Initially this root object is the only existing object, so it is sufficient for the precondition of a complete system to describe only its local state. On the other hand, the final state of an execution of a complete system is described by an arbitrary global assertion. 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 no other objects, and if moreover this execution terminates, then the final state will satisfy the global assertion Q.

4 The proof system

We first introduce the *local* proof system which allows us to reason about the correctness of a single object.

4.1 The local proof system

The proof system for local correctness formulas is an extension of the usual system for sequential programs. Creation statements and input statements are modeled by *random assignments* to the local history.

Definition 4.1
The local proof system consists of the following axioms describing the creation and I/O statements:

Creation

$$\{\forall z_d\, p^c[z_d, h^c \circ \langle \mathsf{self}, z_d\rangle / x_d, h^c\rangle]\}x_d := \mathsf{new}\{p^c\}$$

This axiom describes the execution of a statement $x_d^c := \mathsf{new} \in \mathit{Stat}^c$ by assigning to the variable x_d a randomly choosen object of class d and extending the local history h^c with the corresponding creation record. (In case $d = c$ we have to require additionally that the identity of the created object is distinct from the creator.) The weakest precondition of the assertion p with respect to the statement $x_d^c := \mathsf{new}$ then is calculated by universally quantifying over all the possible objects of class d. Note that in the local assertion language we indeed interpret quantification over objects of class d as ranging over all the possible objects of class d.

Output

$$\{p^c[h^c \circ \langle \mathsf{self}, x, e\rangle / h^c]\}x!e\{p^c\}$$

This axiom describes the execution of an output statement $x!e \in \mathit{Stat}^c$ by extending the local history h^c by the corresponding communication record.

Input1

$$\{\forall z_d\, p[z_d, h^c \circ \langle x, \mathsf{self}, z_d\rangle / y_d, h^c]\}x?y_d\{p\}$$

This axiom describes the execution of an input statement $x?y_d \in \mathit{Stat}^c$ by assigning to the variable y_d a randomly choosen object of class d and extending the local history by the corresponding communication record. As with the axiomatization of object creation such a random choice is modeled by universally quantifying over all the possible objects of class d.

Input2

$$\{\bigwedge_{c'} \forall z_{c'}, z'_d\, p[z'_d, h^c \circ \langle z_{c'}, \mathsf{self}, z'_d\rangle / y_d, h^c]\}?y_d\{p\}$$

Finally, the execution of an input statement $?y_d \in \mathit{Stat}^c$ is described by randomly choosing both a sender and the object sent, assigning the latter object to the variable y_d and extending the local history h^c by the corresponding communication record. (In case $c' = c$ we have to require additionally that the identity of the sender is distinct from the receiver.) The random choice of the object sent is modeled by universally quantifying over all the objects of class d, and the random choice of the sender by the universal quantification over all possible objects of any class, the choice of a class being modeled by the conjunction over all $c \in C$.

When we describe the behaviour of a complete system in terms of the local behaviour of its objects we select out of the random choices made by an object α the correct ones by requiring its local history to be the sequence of those interactions of the global history which involve α.

The axiom for assignment and the rules for the other constructs are as usual.

4.2 The global proof system

In this section we show how to specify the global behaviour of a complete system in terms of the local behaviour of objects. In the following definitions, let $\rho^{c_n} = \langle c_1 \leftarrow S_1^{c_1}, \ldots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}} : S_n^{c_n}\rangle$.

Definition 4.2
The program rule of the global proof system has the following form:

$$\frac{\{\mathit{Init}_k\}S_k\{q_k\}, 1 \le k < n, \qquad \{p \wedge \mathit{Init}_n\}S_n\{q_n\}}{\{p\}\rho\{q_n \downarrow \mathsf{root} \wedge \bigwedge_{1 \le i < n} \forall z_i\, q_i \downarrow z_i\}}$$

The premisses of this rule should be interpreted as being derivable from the local proof system. Here Init_k, for $1 \le k < n$, denotes the local assertion $\bigwedge_{x \in \mathit{IVar}(S_k)}(x \doteq \mathsf{nil}) \wedge h_k \doteq \epsilon$. This assertion describes the initial local state of newly created objects of class c_k. On the other hand, Init_n denotes the local assertion $\bigwedge_c \bigwedge_{x \in \mathit{IVar}_c(S_n)}(x \doteq \mathsf{nil}) \wedge h_n \doteq \epsilon$, which describes the initial local state of the root object. So initially the variables of the root object of a type different from Int or Bool are undefined. This reflects the fact that initially only the root object exists. Note that in the conclusion of the program rule we take as precondition the precondition of the local process of the root object because initially only this object exists. The postcondition consists of a conjunction of the assertion $q_n \downarrow \mathsf{root}$ expressing that the final local state of the root object is characterized by the local assertion q_n, and the assertions $\forall z_i\, q_i \downarrow z_i$, which express that the final local state of every existing object of class c_i is characterized by the local assertion q_i.

Definition 4.3
We have the following consequence rule for programs:

$$\frac{p_n \to p_1, \quad \{p_1\}\rho\{Q_1\}, \quad Q_1 \to Q}{\{p_n\}\rho\{Q\}}$$

5 Semantics

In this section we define in a formal way the semantics of the programming language and the assertion languages. First, in section 5.1, we deal with the assertion languages on their own. Then, in section 5.2, we give a formal semantics to the programming language. Finally, section 5.3 formally defines the notion of truth of a correctness formula.

5.1 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^a. To be precise, we define $\mathbf{O}^d = \mathbf{Z}$, $d = \mathsf{Int}$ and $\mathbf{O}^d = \mathbf{B}$, $d = \mathsf{Bool}$, whereas for every class $c \in C$ we just take for $\mathbf{O}^c$ an arbitrary infinite set. 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). Finally, let $\mathbf{O}^t = Rec^*$, where $Rec = (\bigcup_{c,c'} \mathbf{O}^c \times \mathbf{O}^{c'}) \cup \bigcup_{c,c',d} \mathbf{O}^c \times \mathbf{O}^{c'} \times \mathbf{O}^d)$. $\mathbf{O}^t$ is the set of histories, i.e., finite sequences of creation and communication records.

Definition 5.1
We shall often use generalized Cartesian products of the form

$$\prod_{i \in A} B(i).$$

As usual, the elements of this set are the functions f with domain A such that $f(i) \in B(i)$ for every $i \in A$.

Definition 5.2
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

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

Definition 5.3
The set $LState^c$ of *local states* of class c, with typical element θ^c, is defined by

$$LState^c = \mathbf{O}^c \times \prod_d (IVar^c_d \to \mathbf{O}^d_\perp) \times \mathbf{O}^t.$$

A local state θ^c describes in detail the situation of a single object of class c at a certain moment during program execution. The first component, denoted by $\theta.\mathsf{self}$, determines the identity of the object. The values of the instance variables are given by the second component, whereas the local history of the object, denoted by $\theta.h$, is given by the last component.

It will turn out to be convenient to define the function $\bigtriangledown^c \in \mathit{Lstate}^c$ such that $\bigtriangledown^c_{(2)}(x) = \bot$, for every $x \in \bigcup_d \mathit{IVar}^c_d$ and $\bigtriangledown^c_{(3)} = \epsilon$. Note that this function $\bigtriangledown$ gives the values of the variables of a newly created object: these are all initialized to nil, furthermore the local history of this new object is initialized to the empty sequence.

Definition 5.4
The set *GState* of *global states*, with typical element σ, is defined as follows:

$$\mathit{GState} = (\prod_d P^d) \times \prod_c \Big(\mathbf{O}^c \to \prod_d (\mathit{IVar}^c_d \to \mathbf{O}^d_\bot)\Big) \times \mathbf{O}^t$$

where P^c, for every $c \in C$, denotes the set of finite subsets of $\mathbf{O}^c$, and for $d = \mathsf{Int}, \mathsf{Bool}$ we define $P^d = \{\mathbf{O}^d\}$.

A global state describes the situation of a complete system of objects at a certain moment during program execution. The first component specifies for each class the set of *existing* objects of that class, that is, the set of objects that have been created up to this point in the execution of the program. Relative to some global state σ an object $\alpha \in \mathbf{O}^d$ can be said to exist if $\alpha \in \sigma_{(1)(d)}$. For the built-in data types we have for every global state σ that $\sigma_{(1)(\mathsf{Int})} = \mathbf{Z}$ and $\sigma_{(1)(\mathsf{Bool})} = \mathbf{B}$. Note that $\bot \notin \sigma_{(1)(d)}$ for every $d \in C^+$. The second component of a global state specifies for each object the values of its instance variables. The last component, denoted by $\sigma.h$, specifies the global history.

We introduce the following abbreviations: $\sigma_{(1)(d)}$ will be abbreviated to $\sigma^{(d)}$, and $\sigma^{(d)} \cup \{\bot\}$ to $\sigma^{(d)}_\bot$. Whenever it is clear from the context that $\alpha \in \mathbf{O}^c$, we abbreviate $\sigma_{(2)(c)}(\alpha)$, i.e., the local state of α, by $\sigma(\alpha)$. Furthermore, for any variable $x \in \mathit{IVar}^c_d$, we abbreviate $\sigma_{(2)(c,d)}(\alpha)(x)$, the value of the variable x of the object α, by $\sigma(\alpha)(x)$. The global history of a state σ will be denoted by $\sigma.h$.

Definition 5.5
We now define the set *LEnv* of *logical environments*, with typical element ω, by

$$\mathit{LEnv} = \prod_a (\mathit{LogVar}_a \to \mathbf{O}^a_\bot \times (\mathit{HistVar} \to \mathbf{O}^t).$$

A logical environment assigns values to logical variables and history variables. We abbreviate $\omega_{(a)}(z_a)$ to $\omega(z_a)$.

Definition 5.6
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 $\mathcal{E}_d^c \in Exp_d^c \to LState^c \to \mathbf{O}_\perp^d$ assigns a value $\mathcal{E}[\![e]\!](\theta)$ to the expression e_d^c in the local state θ^c. For example, $\mathcal{E}[\![\mathsf{self}]\!](\theta) = \theta.\mathsf{self}$.

2. The function $\mathcal{L}_d^c \in LExp_d^c \to LEnv \to LState^c \to \mathbf{O}_\perp^d$ assigns a value $\mathcal{L}[\![l]\!](\omega)(\theta)$ to the local expression l_d^c in the logical environment ω and the local state θ^c.

3. The function $\mathcal{H}_l^c \in LHist^c \to LEnv \to LState^c \to \mathbf{O}^t$ assigns a value $\mathcal{H}_l[\![lh]\!](\omega)(\theta)$ to the local history expression lh^c in the logical environment ω and the local state θ^c. For example, $\mathcal{H}_{l\,l}^c[\![h^c]\!](\theta) = \theta.h$.

4. The function $\mathcal{G}_a \in GExp_a \to LEnv \to GState \to \mathbf{O}_\perp^a$ assigns a value $\mathcal{G}[\![g]\!](\omega)(\sigma)$ to the global expression g_a in the logical environment ω and the global state σ. For example, $\mathcal{G}[\![g.x]\!](\omega)(\sigma) = \sigma(\mathcal{G}[\![g]\!](\omega)(\theta))(x)$.

5. The function $\mathcal{H}_g \in GHist \to LEnv \to GState \to \mathbf{O}^t$ assigns a value $\mathcal{H}_g[\![gh]\!](\omega)(\sigma)$ to the global history expression gh in the logical environment ω and the global state σ. For example, $\mathcal{H}_g[\![h]\!](\omega)(\sigma) = \sigma.h$.

6. The function $\mathcal{A}^c \in LAss^c \to LEnv \to LState^c \to \mathbf{B}$ assigns a value $\mathcal{A}[\![p]\!](\omega)(\theta)$ to the local assertion p^c in the logical environment ω and the local state θ^c. Here the following cases are special:

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

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

7. The function $\mathcal{A} \in GAss \to LEnv \to GState \to \mathbf{B}$ assigns a value $\mathcal{A}[\![P]\!](\omega)(\sigma)$ to the global assertion P in the logical environment ω and the global state σ. The following cases are special:

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

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

Note that the quantification ranges over $\sigma^{(d)}$, the set of *existing* objects of type d (which does not include $\perp$).

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

For sequence types, quantification ranges over those sequences of which every element is either $\perp$ or an existing object. For history types, quantification ranges over those elements of $\mathbf{O}^t$ that consist only of interactions between existing objects.

The values $\mathcal{G}[\![g_a]\!](\omega)(\sigma)$ of the global expression g_a, the global history expression $\mathcal{H}_g[\![gh]\!](\omega)(\sigma)$ and $\mathcal{A}[\![g]\!](\omega)(\sigma)$ of the global assertion P are in fact only meaningful for those ω and σ that are consistent and compatible:

Definition 5.7
We define the global state σ to be *consistent*, for which we use the notation $OK(\sigma)$ iff

1. The value in σ of a variable of an existing object is either $\perp$ or an existing object itself.
2. The history of σ describes only interactions between existing objects.
3. There exists a unique object, i.e., the root-object, for which there are no occurrences of records in the history of σ witnessing its creation. Furthermore, for any other existing object there is precisely one occurrence of a record witnessing its creation.
4. Finally, for every object different from the root-object there exist no occurrences of records in the history of σ before the record which witnesses its creation. In other words, an object different from the root-object cannot have interactions before its creation.

The formalisation of these conditions we leave to the reader. It is worthwhile to note that the first two conditions are of a purely logical nature, i.e., they determine the semantics of the global assertion language. The last two conditions, however, can be expressed in the global assertion language.

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_d

of a simple type the value $\perp$ or an existing object, and to every sequence variable z_{d^*} a sequence of which each element is an existing object or equals $\perp$. Furthermore, ω assigns to every history variable a history which consists only of interactions between existing objects. Finally, ω assigns to the logical variable root_c the root-object of σ, where c is the class to which this object belongs, and for any other class c' $\omega(\mathsf{root}_{c'})$ is undefined.

5.2 The semantics of the programming language

In this section we describe a compositional semantics of the programming language. First, we define the semantics of statements:

Definition 5.8
We define the semantic function $\mathcal{S}^c \in \mathit{Stat}^c \to \mathit{LState}^c \to \mathcal{P}(\mathit{LState}^c)$ by induction on the complexity of statements. We give the following cases:

Creation

$$\mathcal{S}[\![x_c := \mathsf{new}]\!](\theta) = \left\{\theta\{\alpha, \theta.h \circ \langle \theta.\mathsf{self}, \alpha\rangle/x, \theta.h\} : \alpha \in \mathbf{O}^c\right\}$$

So the execution of a statement $x_c := \mathsf{new}$ consists of randomly choosing an object $\alpha \in \mathbf{O}^c$, assigning α to x_c, and appending to the local history $\theta.h$ the record witnessing the creation of α. ($\langle\alpha,\beta\rangle$, for $\alpha,\beta \in \bigcup_c \mathbf{O}^c$, denotes the history consisting of the record which witnesses the creation of β by α, and, for $\alpha,\beta \in \mathbf{O}^t$, $\alpha \circ \beta$ denotes the the concatenation of α and β.)

Output

$$\mathcal{S}[\![x!e]\!](\theta) = \left\{\theta\{\theta.h \circ \langle \theta.\mathsf{self}, \theta(x), \mathcal{L}[\![e]\!](\theta)\rangle/\theta.h\}\right\}$$

The execution of an output statement $x!e$ consists of appending to the local history $\theta.h$ the corresponding communication record. (For $\alpha,\beta \in \bigcup_c \mathbf{O}^c$, $\gamma \in \bigcup_d \mathbf{O}^d_\perp$, the history consisting of the communication record witnessing the transmission of γ from α to β, is denoted by $\langle\alpha,\beta,\gamma\rangle$.)

Input1

$$\mathcal{S}[\![x?y_d]\!](\theta) = \left\{\theta\{\alpha, \theta.h \circ \langle \theta(x), \theta.\mathsf{self}, \alpha\rangle/y_d, \theta.h\} : \alpha \in \mathbf{O}^d\right\}$$

The execution of an input statement $x?y$ consists of randomly choosing an object of the appropriate type, assigning it to the variable y, and extending the local history $\theta.h$ with the corresponding communication record.

Input2

$$\mathcal{S}[\![?y_d]\!](\theta) = \left\{\theta\{\alpha, \theta.h \circ \langle\beta, \theta.\mathsf{self}, \alpha\rangle/y_d, \theta.h\} : \alpha \in \mathbf{O}^d, \beta \in \bigcup_c \mathbf{O}^c\right\}$$

The execution of an input statement $?y$ consists of randomly choosing a sender and the object sent, assigning the latter object to the variable y, and extending the local history $\theta.h$ by the corresponding communication record.

The other constructs are dealt with in the standard way. (The semantics of the iteration construct is given by the least fixed point of a continous higher-order operator.)

Definition 5.9
We define the semantic function $\mathcal{P}^c \in \mathit{Prog}^c \to \mathit{LState}^c \to \mathcal{P}(\mathit{GState})$ as follows: Let $\rho = \langle c_1 \leftarrow S_1^{c_1}, \ldots c_{n-1} \leftarrow S_{n-1}^{c_n} : S_n^{c_n}\rangle$. For θ such that $\mathit{Init}_\rho(\theta)$ we define

$$\mathcal{P}[\![\rho]\!](\theta) = \{\sigma' : \mathcal{S}[\![S_n]\!](\theta) = \sigma'(\mathsf{root}_{\sigma'}) \wedge \forall \alpha \in \sigma'^{(c_i)} \mathcal{S}[\![S_i]\!](\nabla_{c_i}) = \sigma'(\alpha),\ 1 \leq i < n\}$$

and for θ such that $\mathit{Init}_\rho(\theta)$ does not hold we define $\mathcal{P}[\![\rho]\!](\theta) = \emptyset$. Here $\mathit{Init}_\rho(\theta)$ holds if and only if $\theta(x_c) = \perp$, $c \in C$, $x_c \in \mathit{IVar}_c^{c_n}$. Furthermore, $\mathsf{root}_{\sigma'}$ denotes the root object of σ'.

It is important to note that by requiring the local histories to be projections of the global history we enforce agreement between the local choices concerning communications. Furthermore, since we consider only consistent states, the local choices concerning the identities of created objects are correct in the sense that no object is created twice. Formally one can prove the correctness of the semantics of statements with respect to a "standard" operational semantics as described in [ABKR].

5.3 Truth of correctness formulas

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

Definition 5.10
We define

$$\models \{p^c\}S^c\{q^c\} \text{ iff } \forall \omega, \theta_1, \theta_2 \in \mathcal{S}[\![S^c]\!](\theta_1) : \theta_1, \omega \models p^c \Rightarrow \theta_2, \omega \models q^c.$$

Next, we define the truth of global correctness formulas.

Definition 5.11
We define

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

6 Soundness and Completeness

In this section we discuss the soundness and the completeness of the proof system.

The soundness of the global proof system follows from the soundness of the local proof system and the validity of the program rule and the consequence rule for programs. Soundness of the local proof system is established by a standard induction on the length of the derivation. The validity of the program rule and the consequence rule follow immediately from the definition of the semantics of the programming language and the assertion languages.

By applying the coding techniques developed in [dB] one can prove the expressibility of the strongest postcondition of a precondition p with respect to a statement S. Formally,

Lemma 6.1
For every local assertion p and statement S there exists a local assertion $SP(p, S)$ such that

$$\theta, \omega \models SP(p, S) \text{ iff there exists } \theta' \in \mathcal{S}[\![S]\!](\theta) \text{ such that } \theta', \omega \models p.$$

Essentially by an application of the standard techniques for proving completeness of sequential programming languages (see [Apt]) we can prove the completeness of the local proof system.

Theorem 6.2
For every local assertions p and q, statement S we have

$$\models \{p\}S\{q\} \text{ implies } \vdash \{p\}S\{q\}.$$

Now we can prove the completeness of the global proof system.

Theorem 6.3
For every local assertion p, global assertion Q, program ρ we have

$$\models \{p\}\rho\{Q\} \text{ implies } \vdash \{p\}\rho\{Q\}.$$

Proof
Let p, Q and $\rho = \langle c_1 \leftarrow S_1^{c_1}, \ldots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}} : S_n^{c_n} \rangle$ such that $\models \{p\}\rho\{Q\}$. By the definition of the strongest postcondition we have $\models \{p \wedge Init_n\}S_n\{SP(p \wedge Init_n, S_n)\}$ and, for $1 \leq i < n$, $\models \{Init_i\}S_i\{SP(Init_i, S_i)\}$. So by the comnpleteness of the local

proof system we derive the derivibility of the above local correctness assertions. An application of the program rule thus gives us

$$\vdash \{p\}\rho\{SP(p \wedge Init_n, S_n) \downarrow \mathsf{root} \wedge \bigwedge_{1 \leq i < n} \forall z_i SP(Init_i, S_i) \downarrow z_i\}.$$

From the semantics of the programming language and the global assertion language it follows that

$$\models SP(p \wedge Init_n, S_n) \downarrow \mathsf{root} \wedge \bigwedge_{1 \leq i < n} \forall z_i SP(Init_i, S_i) \downarrow z_i \rightarrow Q.$$

An application of the consequence rule thus gives us the desired result:

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

□

7 Conclusion

We presented a compositional proof system for a parallel language with dynamic process creation. We proved the soundness and completeness with respect to a formal semantics. Further research will be devoted to the generalization of the presented proof method to the language POOL where processes communicate by means of a rendezvous mechanism.

Another important issue is the application of the proposed proof method and a comparison with the non-compositional proof method developed in [dB].

References

[Am] P.H.M. America: *Issues in the Design of a Parallel Object-Oriented Language.* ESPRIT project 415A, Doc. No. 452, Philips Research Laboratories, Eindhoven, the Netherlands, November 1988. Also in Formal Aspects of Computing.

[AB] P. America and F.S. de Boer. *A proof system for process creation.* Working Conference on Programming Concepts and Methods, Sea Gallilee, Israel, 1990.

[ABKR] P. America, J.W. de Bakker, J.N. Kok and J.J.M.M. Rutten. *Operational semantics for a parallel object-oriented language.* Conference Record of the 13th Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida, 1986, pp. 194-208.

[Apt] K.R. Apt. *Ten years of Hoare logic: a survey — part I.* ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp. 431–483.

[dB] F.S. de Boer. *Reasoning about dynamically evolving process structures.* Ph. D. thesis 1991.

[AFR] K.R. Apt, N. Francez, W.P. de Roever: *A proof system for Communicating Sequential Processes*, ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, July 1980, pp. 359–385.

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

[ZREB] J. Zwiers, W.P. de Roever, P. van Emde Boas: *Compositionality and concurrent networks: soundness and completeness of a proof system.* In Proceedings of the 12th ICALP, Nafplion, Greece, July 15–19, 1985, Springer LNCS 194, pp. 509–519.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. .
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.