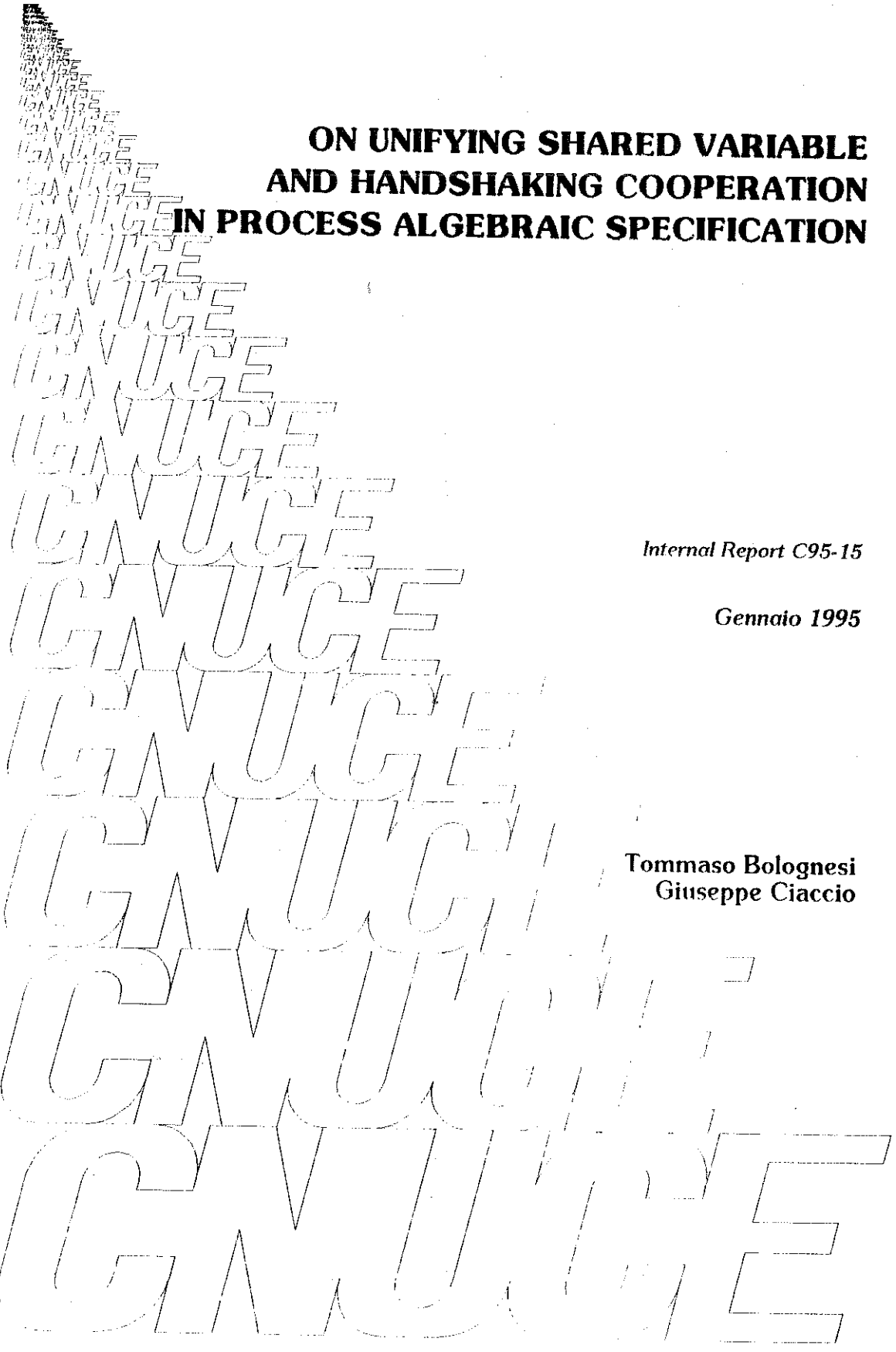
ON UNIFYING SHARED VARIABLE AND HANDSHAKING COOPERATION IN PROCESS ALGEBRAIC SPECIFICATION



Internal Report C95-15

Gennaio 1995

Tommaso Bolognesi
Giuseppe Ciaccio

On unifying shared variable and handshaking cooperation in process-algebraic specification

T. Bolognesi[a] and G. Ciaccio[b]

[a] CNUCE, C.N.R., Pisa, Italy
[b] Consorzio Pisa Ricerche, Pisa, Italy

Keywords: Constraint-oriented specification; Distributed systems; formal semantics; parallel processing; process algebra.

1. Introduction

In multi-party process cooperation by *shared memory* (e.g. in [5, 6, 4]), processes influence one another by writing and reading a set of shared *variables* (Figure 1.a); when, instead, cooperation is by *handshaking* (e.g. in [7, 8, 1]), mutual influence is achieved by sharing *actions*, that is, processes synchronize and exchange data on a set of shared and memory-less interaction points, or gates (Figure 1.b).

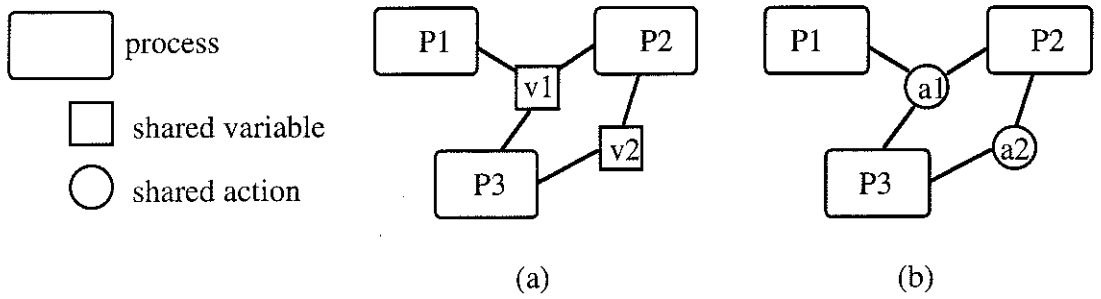


Figure 1 - Shared variable and shared action (handshaking) communication

When processes are composed in a hierarchical fashion, it seems useful to be able to declare some shared variables or actions as local, or private to the processes up to a given layer of the hierarchy; that is, some variables or actions are made no longer available to further processes in the context. An instance of this feature is found in sequential block languages, as the well-known concept of local environment. On the other hand, the following LOTOS [1, 2] process definition provides an example of local confinement of actions:

```
process P[a2] :=
    hide a1 in
            (P2[a1, a2] |[a1, a2]| P3[a1, a2])
            |[a1]|
            P1[a1]
endproc
```

Process `P` consists of three processes that interact via actions `a1` and `a2` according to the pattern of Figure 1.b (`|[Sync]|` is the LOTOS parallel composition operator, where [Sync] is the list of actions on which the argument processes are required to synchronize). However, `a1`-actions are shielded from the environment by the `hide` operator, hence

they are not available for interactions between P and its environment; correspondingly, `a1` does not appear in the list of visible actions in the process header (`[a2]`).

Sequential block languages are based on the concepts of storage and assignment. A typical inference rule of the Structural Operational Semantics (SOS) [9] for these languages is that of the sequential composition construct ';':

$$\frac{<\sigma, P> \text{ ---> } <\sigma', P'>}{<\sigma, P; Q> \text{ ---> } <\sigma', P'; Q>}$$

$<\sigma, P>$ is a *configuration* consisting of program P and of the storage (or memory) function σ: Variables $\rightarrow$ Values, recording the current values in the local memory of P. In the premiss of the rule a computation step is described, in which program P becomes program P' while the current storage σ is transformed into σ' due to some assignment command. The rule defines a possible step of program 'P; Q' with current storage σ, in terms of the step identified in the premiss.

On the other hand, in typical process algebra's the concepts of storage and assignment do not exist at all, and computation steps are characterized by the actions that a process can share with its environment. Correspondingly, the typical SOS inference rule for sequential process composition is:

$$\frac{P \text{ --}\alpha\text{--> } P'}{P; Q \text{ --}\alpha\text{--> } P'; Q}$$

Here configurations are processes without storage - σ has disappeared. An action appears as a label of the computation step, meaning that the evolving process must synchronize with the environment, or *observer*, on that action. A number of processes may cooperate by sharing actions, in the context of a parallel composition with synchronization.

The main purpose of this paper is to explore a unifying view of the two cooperation paradigms of shared variables and shared actions, at the level of structural operational semantics. We do this by defining the semantics of a toy language offering a multi-process parallel composition operator supporting both mechanisms, and suitable operators for hiding (making local) shared variables/actions. The key idea is that accesses to shared memory should be made observable, while the others should not.

At a more general level, we are motivated by the need enhance the expressive flexibility of process algebra's when used as abstract, *formal specification languages*. In this respect, our interest is oriented towards language features that ease the task of the human specifier by supporting a direct and concise specification of the behavioural features of distributed systems, and by avoiding as much as possible the need of 'programming tricks'. For example, in process-algebraic specification languages such as LOTOS a data structure can be kept other than as a process parameter, and changes to it can be performed only by re-instantiating the process, with the modified data structure passed as a new actual parameter. We believe that bringing the traditional notion of memory state back into process algebra's, and considering state variables as 'first-class' objects at the same level of process actions, is quite useful for supporting a natural and direct specification of a wide class of distributed systems. And, in particular, this allows us to extend to state variables the so called 'constraint-oriented' specification style, which has already proved to be quite effective with respect to actions: the global behaviour of the system is captured by the composition of constraints (processes) that mutually restrict not only the allowed action sequences but also the global state configurations. The parallel

composition operator described in this paper is expected to support a specification style based on such a double-sided composition of constraints.

2. A toy language and its formal semantics

Syntax

The syntax of the generic *behaviour expression* B of our toy language is as follows:

B	::=		(* behaviour expression *)
		{Predicate-list}	(* basic process *)
		| Π(Bsync-list)	(* parallel composition *)
		| **let** v = x **in** B	(* variable hiding *)
		| **hide** α **in** B.	(* action hiding *)
Predicate	::=	P(v-list, α, v-list').	
Bsync	::=	B **sync** α-list.	

v	any element of the set Var = {v_1, ..., v_h} of variables
x	any element of a set Val of values
α	any element of the set Act = {a_1, ..., a_k} of actions
P(v-list, α, v-list')	any predicate over variables v-list, α, v-list'

Non-terminals as *v-list, α-list* and similar are defined according to the following schema:

<X>-list ::= λ | <X>, <X>-list.

Generalities on the formal semantics

The semantics of our language is given in terms of labelled transition systems. States (configurations) of such systems are behaviour expressions. The structure of a generic transition will be either

$$B \xrightarrow[\rho, \omega]{\alpha, x} B' \qquad \text{or} \qquad B \xrightarrow[\rho, \omega]{\mathbf{i}} B'$$

For compactly denoting both cases we shall write:

$$B \xrightarrow[\rho, \omega]{\alpha, x \mid \mathbf{i}} B'$$

B, B'	are behaviour expressions (also called 'processes' in the sequel)
α	is an observable action
x	is the value associated to action α
i	is the unobservable action

ρ, ω are partial functions mapping Var into Val. They are abstract representations of, respectively, the *read access* and the *write access* to some shared variables, performed by process B in evolving to B'; they associate values to variables that are *not* local to process B. If $\rho(v)$ (resp. $\omega(v)$) is undefined, then we mean that variable v is not accessed for read (resp. write) operations . Let V-read = dom(ρ) and V-write = dom(ω) be the sets of shared variables respectively read and written by a given transition; it will be generally the case that V-read $\neq$ V-write. The transition rules will ensure that a variable cannot belong *both* to V-read or V-write, *and* to the local storage of B.

A transition states that process B evolves to B' by:

- possibly modifying some of its local variables,
- performing action α with value x, or performing an unobservable action,
- possibly accessing shared variables, through read acces ρ and write access ω.

Accesses to shared memory are made observable, as labels of the transition arrow: the inference rules for parallel composition will require that processes synchronously accessing common shared variables do agree on the values read from/written to these variables - their simultaneous views of the shared memory must be consistent. Local storage is explicitly embedded in behaviour expressions through the 'let' construct, which turns shared variables into local variables.

Basic processes

The basic process in our language is formed by a list of predicates (Predicate-list). We shall write

$$P((r_1, ..., r_m), \alpha, (w_1, ..., w_n))$$

for denoting a predicate where $r_1, ..., r_m$ are the *read access* variables, α is an action, and $w_1, ..., w_n$ are the *write access* variables, which appear primed in the predicate body. It is generally the case that $\{r_1, ..., r_m\} \neq \{w_1, ..., w_n\}$. For example,

$$(v_2 > v_1) \text{ and } (a_1 = v_2 - v_1) \text{ and } (v_3' = v_2 + a_1)$$

is a predicate $P((v_1, v_2), a_1, (v_3))$ reading variables v_1 and v_2, and writing variable v_3. Note that 'a_1' is understood as a variable: the predicate states that if $v_2 > v_1$ then an a_1-action is possible, taking value $(v_2 - v_1)$; as a consequence of the action, variable v_3 is assigned value $(v_2 + a_1)$.

In a basic process, the listed predicates are understood as alternative ways of performing different actions, or even the same action, where the difference refers to the sets of involved variables and the conditions/effects on their values. If more than one predicate is satisfiable at a given moment, then the choice is arbitrary.

Let the following predicate list

$$B = \{ P^i(R^i, \alpha^i, W^i) \mid i = 1, ..., n, R^i \subseteq Var, W^i \subseteq Var \}$$

be a generic basic process. Its transitions are defined by the following rule:

$$\rho : R^k \to Val,\ \omega : W^k \to Val,\ P^k[\rho(r)/r, r \in R^k][x/\alpha^k][\omega(w)/w', w \in W^k]$$

$$\frac{}{\{P^i(R^i, \alpha^i, W^i) \mid i = 1, ..., n\} \xrightarrow[\rho\ ,\ \omega]{\alpha^k,\ x} \{P^i(R^i, \alpha^i, W^i) \mid i = 1, ..., n\}}\ k \in \{1, ..., n\}$$

where X[a/b] denotes substitution of 'a' for every (free) instance of 'b' in X. All the variables read and written by a basic process B are understood as shared, that is, there is no concept of local storage associated to B where their values are kept; in fact, all these values appear as observable to other processes in the environment of B, as ρ and ω labels of the transition arrow. Predicates chosen for 'execution' are never removed from the list: each predicate can be 'executed' infinitely often. In this respect, our basic processes bear some similarity with UNITY programs, which are essentially unordered sets of assignments involving shared variables.

A refined definition of basic processes including also internal actions ('i') would be trivial; in that case, read and write accesses could be de-coupled from observable actions.

As an example, we derive below a transition for a basic process consisting of the single predicate $(a_1 \leq v \text{ and } v' = v - a_1)$, where a_1 is the action variable and v is a shared read/write access variable.

$$\frac{(a_1 \leq v \text{ and } v' = v - a_1)\ [10/v]\ [3/a_1]\ [7/v']}{\{(a_1 \leq v \text{ and } v' = v - a_1)\} \xrightarrow[[v \mapsto 10],\ [v' \mapsto 7]]{a_1,\ 3} \{(a_1 \leq v \text{ and } v' = v - a_1)\}}$$

Note that the transition arrow carries the accesses to v, since v is non-local w.r.t. the process.

Parallel composition

The parallel composition construct involves n processes, each one associated with a list of actions following the keyword 'sync'. Collectively these lists define the inter-process communication pattern: all processes owning action α in their lists must always *synchronize* when performing α, and must agree on the associated value, while the other processes can only execute α-actions independently of one-another (*interleaving*). The synchronizing processes must agree on the values they read from/write to common shared variables.

The inference rule for synchronization is:

$$\frac{\forall i \in I_\alpha:\ B^i \xrightarrow[\rho^i,\ \omega^i]{\alpha,\ x} B^{i\prime},\quad \forall i, j \in I_\alpha:\ \rho^i \textbf{ compatible } \rho^j,\ \omega^i \textbf{ compatible } \omega^j}{\Pi(B^i \textbf{ sync } \alpha\text{-list}^i \mid i = 1, ..., n) \xrightarrow[\cup_{i=1,...,n}\rho^i,\ \cup_{i=1,...,n}\omega^i]{\alpha,\ x} \Pi(X^i \textbf{ sync } \alpha\text{-list}^i \mid i = 1, ..., n)}$$

where:

- Predicate **compatible** expresses the requirement that two read (resp. write) memory accesses μ and μ' agree on common accessed variables:
  μ **compatible** μ' := $\forall v, x, x':\ v \mapsto x \in \mu \wedge v \mapsto x' \in \mu' \Rightarrow x = x'$.
- $I_\alpha = \{i \mid \alpha \in \alpha\text{-list}_i\}$ is the set of indices of those processes B^i which have action α in their synchronization list, and hence must synchronize when performing α.
- $X^i = \textit{if } i \in I_\alpha \textit{ then } B^{i\prime} \textit{ else } B^i$.

The inference rule for interleaving is:

$$\frac{B^k \xrightarrow[\rho,\ \omega]{\alpha,\ x \mid \mathbf{i}} B^{k\prime},\quad \alpha \notin \alpha\text{-list}^k}{\Pi(B^i \textbf{ sync } \alpha\text{-list}^i \mid i = 1, ..., n) \xrightarrow[\rho,\ \omega]{\alpha,\ x \mid \mathbf{i}} \Pi(Y^i \textbf{ sync } \alpha\text{-list}^i \mid i = 1, ..., n)}\ k \in \{1, ..., n\}$$

where $Y^i = \textit{if } (i = k) \textit{ then } B^{k\prime} \textit{ else } B^i$.

We derive below a transition of two basic processes which synchronize with each other on action a_1 while accessing shared variable v. The two basic processes

B1 := $\{(a_1 \le v \text{ and } v' = v - a_1)\}$
B2 := $\{(a_1 + w = 0 \text{ and } w' = v + a_1)\}$

are composed in parallel: $\Pi(B1 \textbf{ sync } a_1, B2 \textbf{ sync } a_1)$.

$$\frac{(a1 \le v \text{ and } v' = v - a1)\ [10/v]\ [3/a1]\ [7/v']}{B1 \xrightarrow[[v \mapsto 10],\ [v' \mapsto 7]]{a_1, 3} B1} \qquad \frac{(a1+w = 0 \text{ and } w' = v+a1)\ [10/v][-3/w]\ [3/a1]\ [13/w']}{B2 \xrightarrow[\begin{bmatrix} v \mapsto 10 \\ w \mapsto -3 \end{bmatrix},\ [w' \mapsto 13]]{a_1, 3} B2}$$

$$\Pi(B1 \textbf{ sync } a_1, B2 \textbf{ sync } a_1) \xrightarrow[\begin{bmatrix} v \mapsto 10 \\ w \mapsto -3 \end{bmatrix}, \begin{bmatrix} v' \mapsto 7 \\ w' \mapsto 13 \end{bmatrix}]{a_1, 3} \Pi(B1 \textbf{ sync } a_1, B2 \textbf{ sync } a_1)$$

The accesses to shared memory performed by B1 and B2 are consistent over the common variables (v); the synchronization between B1 and B2 may hence occur. The whole read (write) access is given by the union of the read (write) accesses of B1 and B2.

Hiding shared variables into local storage

The 'let' construct allows a process to create a local memory and store values in it, by turning a shared variable into a local variable. Process **let** v = y **in** B assigns a value y to local variable v. Read access ρ performed by B on its shared variable v must agree with the value currently associated with v in the 'let' clause. When B writes variable v, via write access ω, the 'let' clause is updated and variable v is bound to the new value. In doing so, any reference to v is removed from ρ and ω, since v is no longer shared. The rule is:

$$\frac{B \xrightarrow[\rho, \omega]{\alpha, x\ \mathbf{i}} B',\ \ \rho(v) \in \{y, \perp\},\ \ \omega(v) \in \{z, \perp\}}{\textbf{let } v = y \textbf{ in } B \xrightarrow[\rho - [v \mapsto y],\ \omega - [v \mapsto z]]{\alpha, x\ \mathbf{i}} \textbf{let } v = q \textbf{ in } B'}$$

where

- x, y, z, q range over Val
- '$\rho(v) \in \{y, \perp\}$' means that value y is read from variable v, or that v is not accessed (the symbol '⊥' indicates that ρ(v) is undefined). The meaning of '$\omega(v) \in \{z, \perp\}$' is analogous.
- q = *if* (ω(v) = z) *then* z *else* y .

According to the rule above, when a variable is made local by the 'let' clause the accesses to it are dropped from ρ and ω; since basic processes are assumed not to own local variables, we are guaranteed that no variable can be at the same time shared and local to any process.

No process B may assume that its shared variables (as opposed to local variables) preserve their values between consecutive transitions : once B has performed a write access to a shared variable, it may happen that other processes in the environment further modify it, asynchronously w.r.t. B.

We derive now a transition for a 'let' expression enclosing the already introduced basic process $B1 := \{(a_1 \leq v \text{ and } v' = v - a_1)\}$.

$$\frac{B1 \xrightarrow[[v \mapsto 10],\,[v' \mapsto 7]]{a1,\,3} B1}{\textbf{let } v = 10 \textbf{ in } B1 \xrightarrow[\varnothing,\varnothing]{a1,\,3} \textbf{let } v = 7 \textbf{ in } B1}$$

Variable v is shared w.r.t. process B1 - accesses to v appear as labels of the transition performed by B1. But, once the 'let' construct is applied to B1, the variable becomes local and is bound to a value. Now the current value of v is no longer accessible to further processes in the context - accesses to v no longer appear as labels of the transition. The read access to v performed by B1 matches the current value expressed by the 'let' clause; such value is then modified according to the write access performed by B1 itself.

Hiding observable actions

The 'hide', found e.g. in the CSP [3] and LOTOS [1] process algebra's, is used for turning an observable action α into a local, unobservable action, denoted '**i**'. The effect of this operator on shared actions is somehow analogous to the effect of the 'let' operator on shared variables: when process B peforms an '**i**' action, no process in the environment of B is allowed to synchronize with it.

The value x originally associated with the action α is no longer observable, nor it is explicitly stored in any local memory of B. However, it may well be that B actually preserves this value by means of a predicate referring to α, which writes x into some variable. The two simple rules of 'hiding' require no comments.

$$\frac{B \xrightarrow[\rho,\,\omega]{\alpha,\,x} B'}{\textbf{hide } \alpha \textbf{ in } B \xrightarrow[\rho,\,\omega]{\mathbf{i}} \textbf{hide } \alpha \textbf{ in } B'} \qquad \frac{B \xrightarrow[\rho,\,\omega]{\alpha',\,x \mid \mathbf{i}} B', \quad \alpha' \neq \alpha}{\textbf{hide } \alpha \textbf{ in } B \xrightarrow[\rho,\,\omega]{\alpha',\,x \mid \mathbf{i}} \textbf{hide } \alpha \textbf{ in } B'}$$

3. An example

We specify below an auction system (see Figure 2). Visitors can perform *user*-actions, by which they identify themselves via an identifier (*id*) and make an *offer*. An offer is possible only if higher than the previous offer. The current (best) offer (*MaxOffer*) is kept and periodically *reset* (supposedly when satisfactory to the auctioneer). The reset-action also adds the winning offer to a *Total*. The system maintains a set of legal user identifiers (*IdSet*), by allowing *update*-actions, which *add* or *remove* a given set (*ids*) of user identifiers to the current set.

The following variables and constants are involved in the specification.

```
actions
        reset           : constant
        user            : record
                              id     :Nat;
                              offer  :Nat;
        update          : record
                              type   :{add, remove};
                              ids    :set of Nat;
variables
        Total, MaxOffer     : Nat;
        IdSet               : set of Nat;
```

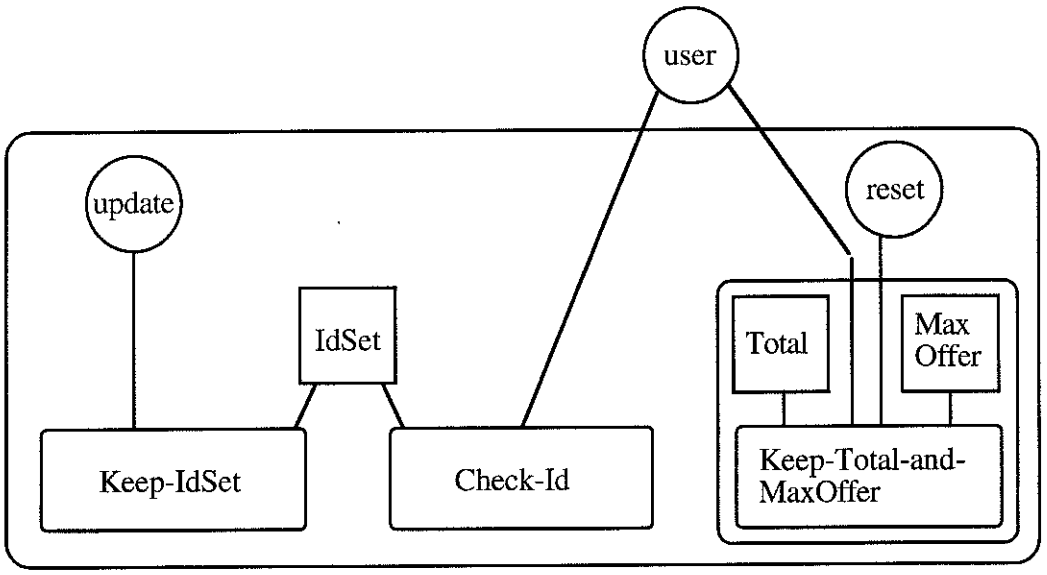


Figure 2 - Structure of the Auction System

Specification of the Auction System

```
(* update and reset actions are internal to the system *)
hide update in
hide reset in

(* the set IdSet of legal (user) identifiers is local to the system and
initially empty *)
let IdSet = {} in
```

```
(* the system consists of the parallel compos. of three processes ...*)
      P      (Keep-IdSet                          sync (),
             Check-Id                             sync (user),
             let Total = 0 in
               let MaxOffer = 0 in
                   Keep-Total-and-MaxOffer sync (user)
             )

(* ... which are expanded below *)

Keep-Id-Set
(* adds and removes user id's to set IdSet of legal identifiers *)
      (P1((Idset), update, (IdSet)) :=
             update.type = add and IdSet' = IdSet ∪ update.ids
             or
             update.type = remove and IdSet' = IdSet - update.ids.
      )

Check-Id
(* accepts actions only from legal users *)
      (P1((Idset), user, ()) := user.id ∈ IdSet.)


Keep-Total-and-MaxOffer
(* accepts user offers only if better than the current best one (P1),
or takes winning offer and adds it to the current Total (P2) *)
      (P1((MaxOffer), user, (MaxOffer)) :=
             user.offer > MaxOffer and MaxOffer' = user.offer,
      P2((Total, MaxOffer), reset, (MaxOffer)) :=
             MaxOffer' = 0 and Total' = Total + MaxOffer
      )
```

The structure of the specification is illustrated in Figure 2. Rounded rectangles correspond to behaviour expressions. Variables (boxes) and actions (circles) inside a rounded rectangle are local to that process.

4. Conclusions

We have illustrated a way to unify the notions of process communication via shared variables and shared actions (handshaking) at an operational semantics level. These two paradigms are supported by a single parallel composition operator. Our notation also treats uniformly variables and actions with respect to hiding operators. The two alternative approaches of (i) sequential imperative languages, where configurations are memory-program pairs ($<\sigma, P>$) and of (ii) process algebra's, where configurations are memory-less behaviour expressions, have been combined by equipping behaviour expressions with local storage. The concept of sharing variables is achieved via a notion of observability, analogous to the one found in process algebra's for synchronizing processes on shared actions: read and write accesses to shared variables are made observable by letting them appear as labels of the transition arrow; processes can then cooperate only if they have consistent views on these shared variables and actions.

References

[1] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", *Computer Networks and ISDN Systems*, Vol. 14, No 1, North-Holland, 1987.

[2] E.Brinksma (ed.), ISO - Information Processing Systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour, ISO IS8807, Feb. 1989, ISO, Geneva.

[3] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, "A Theory of Communicating Sequential Processes", *Journal of the ACM*, **31**(3), July 1984, pp. 560-599.

[4] K. M.Chandy, J. Misra, *Parallel Program Design* (Addison-Wesley, Reading, MA) 1988.

[5] E. W. Dijkstra, "The structure of the "THE" multiprogramming system", *Comm. ACM* **11**(5), 1968, pp. 341-346.

[6] C. A. R. Hoare, "Monitors: an operating system structuring concept", *Comm. ACM* **17**(10), 1974, pp. 549-557.

[7] C. A. R. Hoare, "Communicating sequential processes", *Comm. ACM* **21**(8), 1978, pp. 666-677.

[8] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[9] G. D. Plotkin, "A structural approach to operational semantics", Tech. Rep. DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.