An equational approach to concurrency
(Extended abstract)

F. Orejas
Facultat d'Informàtica
Universitat Politècnica
Barcelona, SPAIN

Originally, abstract data types or software modules were utilized in programming for the implementation and use of data structures, or as a mean to encapsulate classes of actions or operations. In such context, equational specifications provided a way of describing, in a highly implementation independent manner, those modules. Nowadays, the use of equational specifications has expanded to describe things like programming language semantics [4, 6, 7, 12 ] or data bases [3, 10].

From the very beginning, abstract data types were used in concurrent programming [1, 5, 8 ]; however, nobody, at my knowledge, has succeeded in applying the equational specification method in this context.

In this paper, we describe how can we use equations to specify the behaviour of concurrent processes. The underlying idea is the following: in "usual" abstract data types, the "output" functions, such as the function top on a stack, are seen as returning "values"; if we are to modelize the behaviour of a data type shared by some processes (which will also happen to be data types), such as a buffer or an operating system, we may see that an "output" function does not return "values", or, at least, does not return only a value, but it will return "something" that will affect some of the processes of the system, for example, it may make a process to wait or it may order a process to go on with a message. I have called this "something" commands, where
a command is, in fact, an operation on a process. Thus, intuitively, we will see a system of concurrent processes as a collection of processes of different types, some of them aiming to execute an operation on other processes, i.e. aiming to output a command; non-deterministically, some of the processes will do it, causing the system to "change the state".

Specifications shall be divided into two parts. In the first part, the behaviour of the individual processes is described. In the second, the behaviour of the whole system is considered, i.e. the interaction of the different processes. Both specifications will be equational, in fact, the specification of a system will be an extension of the specification of its constituent processes. The first part of the specification shall help in the design and verification of the processes, using the well-known techniques for "usual" abstract data types. The second part shall serve to prove the correctness of the whole system and to study properties such as fairness or deadlock-freeness.

1. The Underlying Model.

The Underlying model of concurrency used will be the following:

- A process is considered to be an abstract data type, i.e. from an implementation point of view of some (private) resources (for example, data) and a family of procedures on operations to manipulate such resources.

- A system of concurrent processes could be considered as a cartesian product of its constituent processes.

- A process interacts with other processes of the system by demanding the execution of some of their operations: at every moment, every process is willing to output a command (i.e. to execute an operation on other process),
non-deterministically, some of these commands will be executed: this will provoke a change of state of the interacting processes, i.e. the processes that have succeeded to output a command and the processes on which the operations have been executed.

For example, the typical schema producer-buffer-consumer may be characterized by three processes: the producer, the buffer and the consumer. In a given moment, the producer may be trying to execute a send operation on the buffer (i.e. trying to output a send command), the consumer may be trying to execute a receive operation on the buffer, and the buffer may be trying to execute a nil command (i.e. doing nothing). Non-deterministically the consumer may succeed in executing the receive command on the buffer, then the situation will be: the consumer will be in the same situation, the buffer (if it contains at least one message) may be trying to execute a consume operation on the consumer, while the consumer will be trying to execute a nil command, etc.

Comparing this model with some others, we may see that it is quite close to Brinch-Hansen's distributed processes. If we ask for symmetry in the interaction, (i.e. a process may succeed to output a command on other process only if the other process is prepared to execute it) then we will be near to CCS [11]. Also, in this case, if we restrict commands to send and receive, then we will be in CSP [9].

2. Preliminaries.

Abstract data types will be defined in a hierarchical manner, i.e. a specification will be a triple \((\Sigma, \mathcal{E}, \mathcal{T})\), where \(\Sigma\) is a signature, that is a pair \((\mathcal{S}, \mathcal{F})\), where \(\mathcal{S}\) is a set of sorts
and $F$ is an operator domain; $E$ is a set of equations (positive conditional) and $T$ is a primitive type, that is, the data type defined by $(\Sigma, E, T)$ shall be an extension of $T$.

At this stage, we shall not consider particular models of the specifications, though in some moment we shall see that initial and terminal models serve to characterize different kinds of behaviours of processes as data types.


Before going into details, let's see an example to illustrate what we consider a specification of a process.


```
type semaphore
primitive type proc
new sorts semaphore, command
new operations
  init : \rightarrow semaphore
  cause: semaphore \rightarrow semaphore (external)
  wait: semaphore \times process \rightarrow semaphore (external)
  next com: semaphore \rightarrow command
  pop com: semaphore \rightarrow semaphore
  nil : \rightarrow command
  continue: process \rightarrow command

new equations
  cause(wait(s,p))=wait(cause(s),p)
  nextcom(init)=nil
  nextcom(wait(init,p))=nil
  nextcom(cause(init))=nil
  nextcom(wait(wait(s,p_1),p_2))=nextcom(wait(s,p_1))
```
nextcom (cause(cause(s)) = nextcom (cause(s))
nextcom (wait(cause(init,p)) = continue(p)
nextcom (s) = nil ⇒ popcom (s) = s
popcom (wait(wait(s,p_1,p_2)) = wait(popcom(s,p_1),p_2)
popcom (cause(cause(s))) = cause (popcom(cause(s)))
popcom (wait(cause(init,p))) = init


In the example, the process semaphore is defined as an abstract data type, having as primitive type process, (i.e., the names of the processes that are going to use (or at least, perform wait commands) the semaphore (we could also consider it as a parameterized data type)). The new sorts introduced in the specification are command and semaphore.
The objects of sort command are the commands that the semaphore process may try to execute on other processes. Notice that, at this level, commands are considered only from a syntactic pint of view. The operations are init, cause, wait, nextcom, popcom, nil and continue; nil and continue define the valid commands for the semaphore; init gives the initial semaphore, cause and wait are external operations, i.e. operations that will be performed on demand of other processes, nextcom gives the command that the semaphore is trying to execute, initially, it is nil but as soon the semaphore has received, at least, one cause and one wait, it will be a continue command; finally, popcom describes the semaphore after "executing" on another process nextcom.

Thus, to specify equationally a process we may follow this method:

1) Specify the basic data types, i.e. booleans, integers, messages, etc.

2) Specify the process names known by the process, or else we shall be specifying a parameterized data type.

3) Combine the specifications 1) and 2) and extend the result with the commands that the process will be trying to execute on other processes.
And 4) Finally extend the specification obtained in 3) with the appropriate signature (including the operations nextcom and popcom) and the appropriate equations.


A system of concurrent processes may be considered as the cartesian product of its constituent processes. These means:

a) A system will be an extension of the combination of the specifications of all its processes obtained as it was explained in section 3.

b) To every process name $p$ there should be a sort name $p$ of one of the processes.

c) Also commands should be related to operations of the signature of the processes.

d) In the system, there should be tupling and untupling operations.

e) If we want to specify a non-deterministic behaviour of the system, we shall need a predicate $P$ to express the successor relation between states of the system.

f) Equations shall express the interaction of processes the following sense. If a semaphore is a process of the system we may have, amongst others, the following equation:

$$\text{nextcom}(\overline{\text{sem}(s)}) = \text{continue}(p) \Rightarrow P(s, \text{assign}(\text{assign}(s, p, \text{continue}(\overline{p(s)})), \text{sem}, \text{popcom}(\overline{\text{sem}(s)}))$$

where $\overline{\text{sem}}$ and $\overline{p}$ are the untupling operations associated to $\text{sem}$ and $p$ sorts, $\text{continue}$ is the operation associated to the continue command and assign is an operation defined as the assign operation on arrays.
5. **Behaviours of processes and systems.**

Usually, a process may be characterized in terms of two kind of behaviours: the "real" behaviour, i.e. the way the process really works, and the observable behaviour, i.e. the way the system behaves from an external point of view, that is not paying attention to internal details. This situation resembles quite closely the relation between initial and terminal algebra semantics in characterizing the behaviour of an abstract data type equationally specified. Moreover, it is clear that if we have a process equationally specified, the initial algebra semantics will characterize the somewhat "real" behaviour of the process, but what it is more interesting is that we may prove that the observable behaviour of the process may be characterized in terms of the terminal algebra semantics.

Observable behaviour of an equationally specified process may be defined in terms of contexts. A context is just a series of operations that may be applied on any object of the data type (i.e. any process situation). As the operations that may be applied from outside to a process are just the external ones, we may restrict the contexts to those having only external operations, i.e. external contexts. Usually a context defines a function from the type of interest to some of the primitive sorts. Here the answer to a context to a given situation is the set of series of commands that the process may output when all the operations in the external context are executed, supposing that popcom operations occur at any time i.e. in our framework an external context ec would define a function ec from the process sort to \(\mathcal{O}\) (command \(\infty\)). The, two specifications \(s_1\) and \(s_2\) (with the same signature and primitive type) would define...
processes $T_{S_1}$ and $T_{S_2}$ observably equivalents iff for every external context $ec$ and every term $t$, $ec(t)$ is the same for both specifications. Finally it may be proved the following theorem.

**Theorem 5.1.**

Two specifications of processes $s_1$ and $s_2$ (with common signature and primitive type) are observably equivalent iff their terminal algebras are isomorphic (terminal in the category of extensions of the primitive type).

Also, in our framework we may study properties concerning the global behaviour of the system, for example, we may study absence od deadlock:

**Definition 5.2.**

A process $p$ is deadlocked when the system is in state $S$ iff $\forall S'$ such that $P(S,S')$, nextcom ($p(S')$) = nil.

A system in state $S$ is deadlock free iff for every process $p$ and every $S'$ such that $P(S,S')$, it holds that $p$ is not deadlocked in $S'$.

6. References

(1) Brinch-Hansen, P.

(2) Brinch-Hansen, P.

(3) Brodie, M.L., Zilles, S.N. (eds.)

(4) Broy, M., Wirsing, M.
"Programming languages as abstract data types", in proc. of 5th CAAP, Lille 1980.
(5) Dijkstra, E.W.
"Hierarchical ordering of sequential processes",

(6) Gaudel, M.C.
"Génération et preuve de compilateurs basées sur une
sémantique formelle des langages de programmation",
These d'Etat, Paris 1980.

(7) Goguen, J.A., Parsaye-Ghomi, K.
"Algebraic denotational semantics using parameterized
abstract modules", in 'Formalization of Programming

(8) Hoare, C.A.R.
"Monitors: an operating system structuring concept",

(9) Hoare, C.A.R.
"Communicating sequential processes", CACM 21, 8

(10) Maibaum, T.S.E.
"Data base instances, abstract data types and data
base specification", to be published.

(11) Milner, R.

(12) Pair, C.
"Abstract data types and algebraic semantics of pro-
gramming languages", Theoretical Computer Science 18,