ABSTRACT
In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING
The features of our mini-language are exhibited on the sample program S on Figure 1. The conventions are close to Algol1 and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type integer channel are declared at line (1), and for any simple type \( \sigma \) (boolean, real, etc.) we could have declared a \( \sigma \) channel. Then processes \( f, g \) and \( h \) are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes: at line (2) \( f \) is stated to communicate via two input lines that can carry integers, and one similar output line. The body of a process is an usual Algol program except for invocation of \( \text{wait} \) until something on an input line (e.g. at (4)) or \( \text{send} \) a variable on a line of compatible type (e.g. at (5)). The process stays blocked on a \( \text{wait} \) until something is being sent on this line by another process, but nothing can prevent a process from performing a \( \text{send} \) on a line. In other words, processes communicate via first-in first-out (fifo) queues.

Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator \( \text{par} \) initiates the concurrent activation of the processes. Such a style of programming is close to many systems using EVENT mechanisms ([1, 2, 3, 4]). A pictorial representation of the \( \text{program} \) is the schema \( P \) on Figure 2, where the nodes represent processes and the arcs communication channels between these processes.

What sort of things would we like to prove on a program like \( S \)? Firstly, that all processes in \( S \) run forever. Secondly, more precisely, that \( S \) prints out (at line (7)) an alternating sequence of 0’s and 1’s forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole systems would stop.

The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION
Informally speaking, a parallel computation is organized in the following way: some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along...
its input lines, using some memory of its own, to produce
output on some or all of its output lines. It is assumed that:

i) Communication lines are the *only way* by which com-
puting station may communicate.

ii) A communication line transmits information within an
unpredictable but *finite* amount of time.

Restrictions are imposed on the behaviour of computing sta-
tions:

iii) At any given time, a computing station is either com-
puting or waiting for information on *one* of its input
lines.

iv) Each computing station follows a sequential program.
(We call here *sequential* program what is usually called
program elsewhere).

Remarks: first, since several computing stations may be
computing simultaneously, this model indeed exhibits some
form of parallelism. Second, restriction iii) means that a
computing station cannot be waiting on data coming from
one or another of its input lines, or alternatively that no two
computing stations are allowed to *send* data on the same
channel. Third, we do not restrict the computing stations
to have a finite memory.

The reader who is mathematically inclined wan think of set
of Turing machines connected via one-way tapes, where each
machine can use its own working tape.

We formalize now the notion of parallel computation intro-
duced above.

2.1 Syntax

A parallel program *schema* is an oriented graph with labeled
nodes and edges, together with some supplementary edges
(see Figure 3): incoming edges with only end vertices, meant
to represent the input lines, and outcoming edges, with only
origin vertices, the output lines.

2.2 Semantics

2.2.1 Outline

Edges in a schema are thought of as pipes: each edge is able
to carry data of a given type *D* (e.g.: integer, boolean,
pointer, procedure, etc...).

An observer placed on the line witnesses its traffic, a (pos-
sibly infinite) sequence of objects of type *D*: it is called the
*history* of the line. Since a computing station has its own
memory, it is not a partial function from the domains of the
inputs into the domain of the outputs, but rather a function
from the histories of its input line into the histories of its
output lines.

2.2.2 Sequence domains

Let *Dω* be the set of finite or denumerably infinite sequences
of elements over a set *D*. In *Dω* we include the empty se-
quence *Λ*. The relation ⊆ defined by *X* ⊆ *Y* iff *X* is an
initial segment of *Y* is a partial order on *Dω*. The min-
imal element of *Dω* is *Λ*. Any increasing chain *ξ* in
*Dω*:

\[ X_1 \subseteq X_2 \subseteq \cdots \subseteq X_n \subseteq \cdots \]

has a least upper bound which we call \( \lim_{\omega} D \xi \). Hence, *Dω* is a complete partial order (c.p.o.).

2.2.3 Domain of interpretation

To each edge *e* in a schema, we associate a set *D*e, the type
of the objects it may carry. The history of line *e* is then an
element of *D*e.

2.2.4 Continuous mappings

A mapping *f* from a complete partial order *A* into a com-
plete partial order *B* is *continuous* iff, for any increasing
chain *α* of *A*:

\[ f(\lim_{A} \alpha) = \lim_{B} f(\alpha) \]
Note that a continuous mapping is also monotonic, i.e. \( x \subseteq y \Rightarrow f(x) \subseteq f(y) \). The following mappings: \( F \) (for first), \( R \) (for remainder) and \( A \) (for Appendix) are examples of continuous
mappings:

- \( F \): to any sequence \( x \) in \( D^\omega \), \( F \) associates the (unit length) sequence constitued of the leftmost element of
\( x \).
- \( R \): to any sequence \( x \) in \( D^\omega \), \( R \) associates the sequence of the right of its leftmost element.
- \( A \): takes two arguments \( L_1 \) and \( L_2 \) in \( D^\omega \) to produce the sequence: leftmost element of \( L_1 \) followed by \( L_2 \).

More precisely, \( F, R, \) and \( A \) obey the axioms:

1. \( A(\Lambda) = \Lambda \)
2. \( A(A,X) = X \)
3. \( A(F(X),R(X)) = R(A(X,Y)) = Y \)

For properties such as deadlock, we shall need to talk formally about the length of a sequence. An elegant way to do so within our formalism is to take the integers with their usual order and complete them with an extra element \( \infty \) to produce the sequence: leftmost element of \( L_1 \) followed by \( L_2 \).

For the interpretation of nodes to continuous functions can be understood in concrete terms:

- a) **Monotonicity** means that receiving more input at a computing station can only provoke it to send more output. Indeed this a crucial property since it allows parallel operation: a machine need not have all of its input to start computing, since future input concerns only future output.

b) Furthermore continuity prevents any station from deciding to send some output only after it has received an infinite amount of input.

Any process written in the simple programming language of Section 1 correspond to a set of continuous functions. The recursive definition of these functions is obtained by the usual method of McCarthy for converting flow-chart programs to recursive definitions.

### 3. FIXPOINT EQUATIONS

Rather than studying the behaviour of a complex machine, we want to study the properties of the solution of a set of equations. To each parallel program (i.e. interpreted schema) we associate a set \( \Sigma \) of equations on sequence domains in such a way that a set of sequences is a possible solution of the system iff it is a possible set of histories for the communication’s lines of the program:

i) To every line \( e \), of type \( D_e \), associate to a variable \( X_e \) ranging over \( D_e \).

ii) If \( X_1, X_2, \ldots, X_n \) are the variables associated to the input lines and \( i_1, \ldots, i_k \) are the sequences fed as inputs on the lines include the equations:

\[
\begin{align*}
X_1 &= i_1 \\
& \vdots \\
X_k &= i_k
\end{align*}
\]

iii) For each node \( f \), interpreted with the functions \( f_1, \ldots, f_p \), with input variables \( X_1, \ldots, X_n \) output \( X'_1, \ldots, X'_p \) include \( p \) equations in \( \Sigma' \):

\[
\begin{align*}
X'_1 &= f_1(X_1, \ldots, X_n) \\
& \vdots \\
X'_p &= f_p(X_1, \ldots, X_n)
\end{align*}
\]
The program $P$ and the associated system $\Sigma_P$.

Clearly, the histories of the lines of the program $P$ have to satisfy the system $\Sigma_P$. $\Sigma_P$ is a set of fixpoint equations over c.p.o.’s, where the operators are continuous. It is a well-known mathematical result (see for example Milner [5]) that such a system admits a unique minimal solution. It is outside the scope of this paper to show that this minimal solution constitutes indeed the vector of the communication’s lines, given a suitable implementation. Such a proof can be found in Cadiou [6] in a similar set up.

The first property of this minimal solution gives us access to the most powerful rule of induction used in proving programs correct (see Manna, Ness, Vuillemin [7]), i.e. Scott’s rule:

**Property 1 (Kleene).** The minimal solution

$$\{ Y(X_1), Y(X_2), \ldots , Y(X_n) \}$$

of the system $\Sigma_P = \{ X_i = \tau_i (X_1, \ldots , X_n) \mid i \in [1,n] \}$ where the $\tau_i$ are terms built out of continuous operators is $\lim_{i \to +\infty} (X_1, \ldots , X_n)$ where

$$X_i^0 = \Lambda (i \in [1,n])$$

(Strictly speaking there might be $n$ different $\Lambda$’s)

$$X_i^{i+1} = \tau_i (X_1, \ldots , X_n) \quad (i \in [1,n])$$

Scott’s induction rule in this case can be stated as follows, if $P$ is an admissible predicate (see Manna, Ness, Vuillemin [7]):

$$P(\Lambda, \ldots , \Lambda) \quad P(X_1, \ldots , X_n) \supseteq P(\tau_1 (X_1, \ldots , X_n), \ldots , \tau_n (X_1, \ldots , X_n))$$

A property of a parallel program is stated as a relation between the input sequences and the output sequences or in general between the histories of some communication lines. Since we may use Scott’s rule, all the techniques for proving properties of recursive programs studied in Vuillemin [8] are available, in particular structural induction and recursion induction.

**Example 2.** The system $\Sigma_S$ associated with program $S$ is:

$$\Sigma_S = \begin{cases} X = f(Y, Z) \\ Y = h(T_1, 0) \\ Z = h(T_2, 1) \\ T_1 = g_1(X) \\ T_2 = g_2(X) \end{cases}$$

where $f, g_1, g_2$ and $h$ are given in 2.2.5.

As an illustration, let us prove that the history $X$ which is exactly what $S$ prints out, is an infinite alternating sequence of 0’s and 1’s. In other words, if $\mathcal{X}$ is the minimal fixpoint of $X = A({\{0\}}, A({\{1\}}, X))$ then $X = \mathcal{X}$.

The system $\Sigma_S$ can be reduced to a single fixpoint equation:

$$X = f(h(g_1(X), 0), h(g_2(X), 1))$$

Using the definition of $f$ and $h$, and the properties of $F$ and $A$ we transform Equation (1) to

$$X = A({\{0\}}, A({\{1\}}, f(g_1(X), g_2(X))))$$

**Lemma 1.** For all $U, U = f(g_1(U), g_2(U))$

**Proof.** By structural induction. The Lemma is obviously true for $\Lambda$, and for any sequence of length 1. Assume it is true for $V$, then:

$$f(g_1(A({\{a\}}, A({\{b\}}, V))), g_2(A({\{a\}}, A({\{b\}}, V)))) = A({\{a\}}, A({\{b\}}, f(g_1(V), g_2(V)))) = A({\{a\}}, A({\{b\}}, V))$$

by induction hypothesis. □

From Equation (2) and this Lemma above we deduce:

$$\mathcal{X} \subseteq X$$

With this Lemma again it is trivial to see that $X \subseteq \mathcal{X}$, which is proves the result. Since the mapping length is continuous, $\text{length}(\mathcal{X})$ is the minimal solution (in $\mathbb{N}$) of

$$\text{length}(X) = 2 + \text{length}(X)$$

which is obviously $\infty$. Hence $T_1$ and $T_2$ are infinite sequences and so are $Y$ and $Z$. We have thus answered the first two questions raised in Section 1 about program $S$. □

The simplicity of the program $S$ and the proof produced should not induce the reader into believing that only very simple minded proofs are feasible. Milner and Weyrauch [9] used the system LCF, based on Scott’s induction rule, to check mechanically the complete proof of the correctness of a small compiler, a very large proof indeed. LCF can be readily used for our purposes and very large and trustworthy proofs could be produced on this system.

**Property 2 (Scott).** The minimal solution of the $\Sigma_P$ is a continuous function of the parameters of the system, in particular the values of the input streams, or the operators of the system.

In more concrete terms, Property 2 means that, in this model of parallel computation:

1. Arbitrary interconnection of systems, as well as processes, is legitimate. Hence, top-down design finds here a mathematical justification since we can postpone the decision to implement a given function by a single process or a set of interconnected processes: this decision will not introduce perturbations in the remainder of the system.
A recursive parallel schema is a set \( F_1, F_2, \ldots, F_l \) of parallel schemata in which some nodes may be labeled \( \text{parallel} \). A recursive parallel schema is a set of fixpoint equations that contain no new fixpoint equations. We state the main results (Courcelle, Kahn, Vuillemin [10]):

**Theorem 1**: The equivalence of schemata containing uninterpreted processes and \( n \)-duplicators is decidable.

**Theorem 2**: There exists a unique minimal schema \( \hat{S} \) (i.e. containing a minimum number of process nodes) equivalent to a given schema \( S \).

**Theorem 3**: The system of equations corresponding to \( S \) containing the minimum number of equations are obtained by taking minimal cuts of \( \hat{S} \).

The results concerning recursive parallel schemata are much harder. Restricting ourselves to recursive processes with one input and one output we know (Courcelle-Vuillemin [11]):

**Theorem 4**: Equivalence of recursive parallel schemata is decidable.

### 5. SCHEMATOLOGY

Structural properties of parallel programs are discovered in studying parallel program schemata. For example we prove that the schemata on Figure 8 are equivalent, i.e. whatever process \( f \) and \( g \) may be the two resulting programs will be equivalent.

(Nota: these schemata are partially interpreted: the node 2 called a 2-plicator sends a copy of each input on each output line. We allow such nodes in schemata because they introduce no new fixpoint equations).

We state the main results (Courcelle, Kahn, Vuillemin [10]):

**Theorem 1**: The equivalence of schemata containing uninterpreted processes and \( n \)-duplicators is decidable.

**Theorem 2**: There exists a unique minimal schema \( \hat{S} \) (i.e. containing a minimum number of process nodes) equivalent to a given schema \( S \).

**Theorem 3**: The system of equations corresponding to \( S \) containing the minimum number of equations are obtained by taking minimal cuts of \( \hat{S} \).

The results concerning recursive parallel schemata are much harder. Restricting ourselves to recursive processes with one input and one output we know (Courcelle-Vuillemin [11]):

**Theorem 4**: Equivalence of recursive parallel schemata is decidable.

### 6. DISCUSSION AND CONCLUSION

The kind of parallel programming we have studied in this paper is severely limited: it can produce only determinate programs. We argue however that:

i) large parts of operating systems are written so as to be determinate. The method of monitors advocated by Hoare narrows down the possible locations of non-determinacy.
ii) the primitives wait and send $x$ on $y$ that we studied are not too far from reality as exemplified by [1, 3, 4].

iii) we do not think it is impossible to extend the theory to non-determinate parallel programs, although how to satisfactorily do so is far from obvious.

iv) the programming language we have introduced can be extended by adding new primitive processes (i.e. that cannot be programmed as processes with wait and send). A typical such process is
\begin{verbatim}
WARN (integer in X,Y; logical out Z)
\end{verbatim}
that sends a true value on its output line each time some integer is received on either of its input lines. The only condition to be verified by the new primitive processes, and verified by WARN, is that the history of the output line be a continuous function of the histories of the input lines.

Looking now at the merits of our approach, we see the essential one as the eradication of the notion of state of a complex system. More precisely, in Lauer [12] and Gilbert [13] for example, a system is thought of as having a huge "state vector" and making non-deterministic transitions from state to state. This view leads to proofs growing exponentially with the number of processes (we grow linearly) and is blind to the structure of the system, making the proofs counter-intuitive. Furthermore it cannot deal with an unbounded number of processes, something we get almost "for free". Our proofs can be checked mechanically in LCF [14], another non-negligible advantage since they will often be tedious but without great mathematical depth.

Our last conclusion is to recall a principle that has been so often fruitful in Computer Science and that is central in Scott’s theory of commutation: a good concept is one that is closed

1. under arbitrary composition
2. under recursion.

Acknowledgments
Many thanks to J. Vuillemin and R. Milner who helped me to formulate this theory. J.M. Cadiou and J. Vuillemin provided most useful comments on many draft versions of this paper.

7. REFERENCES