REFERENCES


THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAIN
RIA-Laboria, Domaine de Voluceau, 78150 Rocquencourt, France
and
Commissariat à l'Energie Atomique, France

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 schemes are given. We hope in this way to make a case for a 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 system designers as to what are the best practices for writing system 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 fig.1. The conventions are close to Algol 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 (boolean, real, etc...) we could have declared a channel. Then processes g, h and b are declared, each with type process. Aside from usual parameters (passed by value in this example, like INPUT at line (3)), we can declare in the header 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 a usual Algol program except for invocation of user exits and input line (e.g. at (4)) or a variable on a line of communication (e.g. at (5)). The process stays blocked on a blocked on a user exit until something is being sent on this line by another process, but nothing can prevent a process from performing a send on a line. In other words, processes communicate via fixed-size first-in-first-out (FIFO) queues. Calling instances of the process S 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 initial operator in the interface allows the processes to communicate. The new data channel and the two main processes are defined in the main program.

Fig.1. Simple parallel program S.

Fig.2. The schema P for the program S.
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 computing stations may communicate.

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

Restrictions are imposed on the behavior of computing stations:

(iii) At any given time, a computing station is either computing 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 a program element.

Remarks: First, since several computing stations may be computing simultaneously, indeed, such a system will exhibit some form of parallelism. Second, restrictions are only imposed on the behavior of computing stations, restricting only the time at which a computing station is free to communicate. Third, we do not restrict the computing stations to have a finite memory.

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

We now see the notion of parallel computation introduced above.

2.1 System

A parallel program schema is an oriented graph labeled nodes in degree 2 and edges, together with some supplementary edges (see fig. 3): incoming edges with only input vertices, mean to represent input lines, and outgoing edges, with only output 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 from one of its nodes (on the output side of the edge) to one of its nodes (on the input side of the edge). An observer on a given line witnesses its traffic, a possibly infinite sequence of objects of type D, it is called the history of the line. Since a computing station is considered to have memory, its behavior is a partial function from the domain of the input into the domain of the output, but rather a function from the histories of its input lines into the histories of its output lines.

2.2.2 Sequence domains

Let D be the set of finite or denumerably infinite sequences of elements over a set D. In D, we will denote the empty sequence by ε. The relation defined by

\[ X 
\]

is the set of finite or denumerably infinite sequences of elements over a set D. The set D is a partial order on D. The minimal element of D is ε. Any increasing chain \( X \subseteq Y \) has a least upper bound which we will call \( X \cup Y \). Hence D is a complete partial order (c.p.o.).

2.2.3 Domain of interpretation

A mapping from a partial order D into a partial order D is continuous, for any increasing chain \( \alpha \subseteq \beta \), the type of the object it may carry. The history of a line is then an element of D, which can be partially ordered in a similar way.

2.2.4 Continuous mappings

A mapping \( f \) from a complete partial order \( A \) into a complete partial order \( B \) is continuous, for any increasing chain \( \alpha \subseteq \beta \), the type of the object it may carry. The history of a line is then an element of D, which can be partially ordered in a similar way.

2.2.5 Computing schema

We are now ready to interpret the nodes in a parallel schema. To each node with input lines carrying data in \( D_1, D_2, \ldots, D_n \) and producing data in \( D_1, D_2, \ldots, D_n \) we associate a continuous function from \( D_1 \times D_2 \times \cdots \times D_n \) into \( D_1 \times D_2 \times \cdots \times D_n \). For example, in fig. 6, we specify two continuous functions \( f_1, f_2 \) in order to interpret node \( f \) in parallel schema fig. 4.

\[ f_1(x_1, x_2) = x_1 + x_2 \\ f_2(x_1, x_2) = x_1 - x_2 \]

Note that a continuous mapping is also monotonic, i.e., \( x \leq y \) \( \implies \) \( f(x) \leq f(y) \)

The following mappings: \( f_1(r,n), f_2(r,n) \) and \( f_3(r,n) \) are examples of continuous mappings:

\[ f_1(x_1, x_2) = x_1 + x_2 \\ f_2(x_1, x_2) = x_1 - x_2 \\ f_3(x_1, x_2) = x_1 \times x_2 \]

2.2.6 Concurrency

In growing programs, the process of function f is associated with the complete function in \( N \times N \) defined recursively by:

\[ f(x,y) = \begin{cases} \phi(x,y) & \text{if } f(x,y) \neq \text{undefined} \\ \text{error} & \text{otherwise} \end{cases} \]

The process \( g \) is associated with two functions, one per output direction, defined recursively by:

\[ g_i(0) = \text{skip} \] for \( i = 1, \ldots, m \)

Similarly, the function \( h \) maps \( n \) into \( n \) as:

\[ h(x) = \begin{cases} 2x & \text{if } x \leq N \\ \text{error} & \text{otherwise} \end{cases} \]

In these examples, the notation is used to denote the set of all elements in \( N \). This notation is customarily used to denote a set of all elements in \( N \).

2.2.7 Parallelism

In parallel programs, the process of function f is associated with the complete function in \( N \times N \) defined recursively by:

\[ f(x,y) = \begin{cases} \phi(x,y) & \text{if } f(x,y) \neq \text{undefined} \\ \text{error} & \text{otherwise} \end{cases} \]

The process \( g \) is associated with two functions, one per output direction, defined recursively by:

\[ g_i(0) = \text{skip} \] for \( i = 1, \ldots, m \)

Similarly, the function \( h \) maps \( n \) into \( n \) as:

\[ h(x) = \begin{cases} 2x & \text{if } x \leq N \\ \text{error} & \text{otherwise} \end{cases} \]

In these examples, the notation is used to denote the set of all elements in \( N \). This notation is customarily used to denote a set of all elements in \( N \).

3. Fixed Point Equations

Rearranging the behaviour of a complex machine, we want to study the properties of the solution of the set of equations. To each parallel program (input sequence domain), we associate a system of equations on sequence domains in such a way that any solution is possible to solutions of the system iff it is a possible solution of the system. The solution is continuous, for any increasing chain \( \alpha \subseteq \beta \), the type of the object it may carry. The history of a line is then an element of D, which can be partially ordered in a similar way.

3.1 Induction

Induction in this case can be stated as follows: if \( f \) is an associative predicate (see Hanner, Niss, and Wullien [9]), we have:

\[ f(x_1, x_2) = f(f(x_1, x_2), f(x_3, x_4)) \]

A property of a parallel program is stated as a relation between the input sequence and the output sequence or in general between the inputs and the outputs of some communication lines. Since we may use Scott's rule, all the properties for growing properties of recursive programs studied in Wullien (10) are available, in particular structural induction and recursion induction.

3.2 Induction

Example: The system \( Z \) associated with program \( S \) is:

\[ X = f(T, Z) \\ T = h(x_1, y) \\ Z = h(x_2, y) \]

where \( f, g, h, x_1, x_2, y \) are given in 2.2.5.

As an illustration, let us prove that the history X, labeled \( x_1, x_2, y \), which satisfies the following relation, is an infinite alternating sequence of 0's and 1's. In other words, \( X = 0 \) is the minimal solution of \( \alpha \neq 0 \). Then:

\[ X = f(T, Z) \\ T = h(x_1, y) \\ Z = h(x_2, y) \]

The system \( Z \) can be reduced to a single fixed point equation:

\[ X = f(T, Z) \]

Clearly, the properties of the histories of the program \( Z \) must be satisfied by the system \( Z \). The system \( Z \) is a set of fixed points over X. The set of fixed points is well-founded, and the properties of \( f \) and \( g \) are equivalent. A unique minimal solution is outside the scope of this paper to show how a minimal solution constitutes the vector of histories of the communication lines, given a suitable implementation. Such a proof can be found in (5) in a similar set up.

The first property of this minimal solution gives us access to the most powerful rule of induction used
f(g(x), f(1, A)) = 2 \quad \text{and} \quad f(1, A) = 2

From eq. (2) and the lemma above we deduce:

With this Lemma again it is trivial to see that \( X \subseteq Y \), which proves the result. Since the mapping length is continuous, length \((X)\) is the minimal solution of \( \Gamma(X) \).

The simplicity of the program 5 and the proof produced should not induce the reader into believing that only very simple-minded proofs are feasible. Milner and Skayrach [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 \( \Gamma(X) \) 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. Any arbitrary interconnected system, as well as processes, is legitimate. Hence, top-down design finds here a mathematical justification since we can postulate a decomposition 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.

2. A parallel program can be safely simulated on a sequential machine, provided the scheduling algorithm is fair enough, i.e., it essentially attributes some computing time to a process when warranted. If this algorithm is not fair however, the only thing that may happen is that the parallel program to produce less output than what could be expected. But what is produced is correct.

This remark and a simple argument on lengths answer the last question about program 5 raised in the first section.

6. DISCUSSION AND CONCLUSION

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

1. Large parts of operating systems are written so as to be deterministic. The method of monitors advocated by Moore narrows down the possible locations of non-determinism.

2. The primitives used and needed by that we are not too far from reality as exemplified by (1), (3), (4).

We do not think it is impossible to extend the theory to non-deterministic parallel programs, although how to satisfactorily do so is far from obvious.

In the programming language we have introduced can be extended by adding new primitive processes (i.e., that cannot be programmed as processes with inputs and sends). A typical such process is \( W \) (which is not covered by Fig. 2) that sends a time value on its output line each time some integer is received on one of its input lines. The only condition to be verified by the new primitive processes, and verified by LCF, is that the history of the output line be a continuous function of the history of the input lines.

Looking now at the results of our approach, we see the essential one as the elucidation of the notion of state of a complex system. More precisely, in Lauer [11] and Mullins [10] for example, a system is thought of as having a "state vector" and making continuous transitions from state to state. This view leads to proofs of growing sequentially with the number of states (we grow linearly) and is blind to the structure of the system, making the proofs counter-intuitive. Furthermore, one cannot deal with an unbounded number of processes, something we are forced to do "for free". Our proofs can be checked mechanically in LCF (8), another non negligible advantage since they will often be tedious and without much mathematical depth.

Our last conclusion is to recall a principle that has been often expressed in Computer Science that is central to Scott's theory of computation: a good computation is:

1. under arbitrary composition.  
2. under recursion.

ACKNOWLEDGMENTS

Many thanks to J. V. Millar and R. Millar who helped me to formulate this theory. J. M. Cadle and J. V. Millar provided most useful comments on the many draft versions of this paper.

REFERENCES


4. IBM System/360 Operating System, Control program services, Form C84-6541-0.


8) D. Lipton, Reduction A Method of Proving for Parallel Programs. CACM 18(2), Dec. 1975, 117-121.

1. INTRODUCTION

Many algorithms are naturally organized as systems of independent processes which interact and exist in concert. In this paper, we present a structured approach to the programming of such systems. This approach is embodied in a programming language which subordinates control to structure, relieving the programmer of the burden of control management and permitting process systems to be executed either sequentially or concurrently with the same result. The language was designed to reflect the clear semantic conception of process interaction presented in [1], with the result that programs are relatively easy to verify.

1.1 Concurrency, multiplicity algorithms, and pipelines

Our approach is a process is derived from Conway's original concept of coroutines, [2] which he introduced as an improved way of executing multiplex and/or algorithms. In his words, \textit{"a coroutine is an autonomous program which communicates with adjacent modules as if they were input and output subroutines."} The coroutines represent progress across each of which improves the convergence of a state of data, so that their execution can be interleaved in time according to a "demand-driven" scheduling strategy. This node of execution was described succinctly in "when the programme is computed so that it sends a send event to B, B must for a while until it executes a send command, which means it needs something from A, the control is then transferred to \textit{A} until it uses to occur, then program control is returned to \textit{B} from where it left off." Conway went on to note that coroutines can be executed in time, if parallel hardware is available. This is possible without time-independent side effects because the coroutines communicate with each other only via input/output instructions, and this in turn follows from the fact that the coroutines are relatively separate processes of a multiplex algorithm. When executed in parallel, such a system is called a "pipeline" system. The pipeline system has been concerned with generalizing this simple linear organization to buffer their communication. The classic illustration of coroutines is the cooperating between the lexical analyzer and the parser in a compiler. However, algorithms structured as a set of interacting coroutines occur in many applications besides compiling, such as input/output handling, "file" manipulation, "graph" algorithm manipulation, "parallel computation," and "artificial intelligence," [3] The UNIX operating system [4] provides a "pipelineing" in its command language which is used to connect processes together in linear pipelines for parallel execution.

1.2 Alternative approaches to coroutines

A different approach to coroutines, typified by the SIMULA control primitives call, return and resume, is fairly widespread, [7-9] The SIMULA primitives can be used to implement Conway's style of coroutines, where control transfers are hidden in the input/output functions, but they also allow other types of interaction which occur in the inside of control relationships. Use of the resume command in particular leads to obscure control structures, because it resembles a go to command with a moving target, the sake of pipeline reliability and verification one needs to impose discipline on the use of these primitives, and when this is done [10-11] it leads to the structuring of process interaction along the lines of Conway's original proposal.

1.3 Related ideas

The evaluation mechanism used in our system has its origins theoretical work [19-20] on "call-by-need" parameter passing. The same work has inspired "lazy evaluators" for LISP [21], which in some respects behave like our process networks and can execute essentially identical modified versions of some of our examples. These systems, however, do not have any analogue of our cyclic network structures.

Communication channels are related to the streams of LAMIN, [22] which foresee their connection with coroutines. In fact, our language can be viewed as a powerful stream processing language. A simplified version of streams already exists in PRO-2 [24], in the SIMULA system SIMULA, but their functionality is limited because the lack of processes in POP-1 makes it awkward to define stream transformations.

2. THE PROGRAMMING LANGUAGE

2.1 Introduction

The language presented here provides concise and flexible means for creating complex networks of processes which can communicate directly, and which can communicate indirectly through channels. The key concepts are processes and structures called channels which interconnect processes and synchronize their activities. The channels carry information in one direction only from a producer process to one or more consumer processes, and they behave like unbounded FIFO queues.

In this section we explain how processes are declared, how they communicate via channels, and how networks of processes are created and transformed. Then we introduce a powerful functional notation and discuss iterative versus recursive reconfiguration of processes.
2.4 Reconfiguration

While a process program is running, it may be visualized as a directed graph where nodes represent processes and edges stand for communication channels. During computation, this graph may evolve in a top-down fashion: a module may be replaced by a subgraph. This subgraph can be appropriately spliced into the incoming/outgoing edges (i.e., channels) of the original process. The reconfiguration instruction has the form:

```plaintext
<description> (edges in the new subgraph)
```

and its body defines a transformation of this sort. The keyword `do` stands for "do concurrently" or "do in coroutines." Closeout is just the matching closing bracket. The body of the instruction has two sections:

1. (the declaration of new communication lines)
2. (the declaration of new processes)

In Fig. 1, the process assigns a single reconfiguration instruction. Its evaluation provides the graph transformation displayed in Fig. 2.

2.5 Activation

Within a process program, an activation instruction of the form:

```plaintext
<description> (edges in the new subgraph)
```

may be interpreted as execution of a process program. This instruction terminates when the process program completes its new work. If the network is to be set up by the reconfiguration instruction, it is set up in the context of the current program. This is automatic if the program is activated by the start instruction of Fig. 1, as the process program is set up by the start instruction.

The reconfiguration program is more interesting. This form of the slice of Eramotes appears to be the basis of the author's knowledge, for the first time in (E). For each newly discovered prime, a FILTER process is created by SIFT, whose task is to remove all multiples of the prime discovered. The program is closed after the reconfiguration instruction is executed. A proof of the correctness of this program is straightforward.

```plaintext
Process INTEGRATE 0 Q 0 0 0
repeat (MODULE 0) forever
```

```plaintext
Process FILTER Q IN Q 0 0
repeat (MODULE 0) forever
```

```plaintext
Process OUTPUT Q IN Q 0 0
```

Finally, a difficulty crops up when, as a consequence of a reconfiguration, the newly created network has to output values on a channel previously bound to an output port of the parent process. As happens in SIFT for example (see Fig. 2), the source's reconfiguration, called splicing, must be included in the list of stream expressions.

```plaintext
Process INTEGRATE 0 Q 0 0
repeat (MODULE 0) forever
```

Note that this device offers a simple way to have a network shrink rather than expand. For example, the process program:

```plaintext
Process CHAIN A 0 0 0
repeat (MODULE 0) forever
```

2.6 Functional notation

The constructs explained so far are sufficient for all process programs and device much more elegant programs in a functional notation. More processes have a single output line so they are functional and much easier to describe. This is the way that ALGO, 60 permits functions along with processes. Consider the construct "if then else." After this in a functional notation, such expressions can be provided as arguments where (input channel)

```plaintext
SIFT(Q) IN Q 0 0
```

will now look like:

```plaintext
Process INTEGER 0 0
Process FILTER PRIME Q IN Q 0 0
Process SIFT Q IN Q 0 0
Process OUTPUT Q IN Q 0 0
```

This notation is very convenient because many channels are created implicitly. But stream expressions denote only algebraic sublists, a simple construct (akin to Landin's RETREAT) allows networks to be built with cycles, in a reconfiguration instruction, we allow now a list of elementary reconfigurations, where, in the familiar SIFT notation:

```plaintext
(elem.recomporte)=process call
```

In the program of Fig. 4, a reconfiguration written in this notation allows the process to be set up as follows:

```plaintext
if then else...
```

Now both occur between the channel list and within the stream expressions. Notice that the reconfiguration specifies in which channel the program is closed and as output, the stream expressions.

Finally, a difficulty crops up when, as a consequence of a reconfiguration, the newly created network has to output values on a channel previously bound to an output port of the parent process, as happens in SIFT for example (see Fig. 2). Another kind of elementary reconfiguration, called splicing, must be included in the list of stream expressions.

```plaintext
Process INTEGRATE 0 Q 0 0
repeat (MODULE 0) forever
```

Note that this device offers a simple way to have a network shrink rather than expand. For example, the process program:

```plaintext
Process CHAIN A 0 0 0
repeat (MODULE 0) forever
```

2.7 Optimizing recursion

The situation of the process SIFT is a common one. This process reconfigures into a subgraph containing a new cycle. As a result, the stream expressions could merely create in front of itself new FILTERS upon receiving inputs and thus be iterative rather than recursive. To indicate that the parent process is to be reconfigured, the command "whether..." is used. Usually the bindings of the parent process are not needed. An assignment switches inputs to a new channel needed to reconnect outputs. The transformed version of SIFT is
In our implementation, significant time and space savings result from this transform.

2.8 An example

The programming style is now much less imperative. To illustrate this, consider a problem previously created by Dijkstra. (3) One is requested to generate the first 50 elements of the form \( n^2 + n + 41 \) in increasing order, without omission or repetition. The idea of the solution is to think of each element as a single object and to notice that if we multiply it by 2, 3, or 5, we obtain subsequent elements. The technique to be shown is based on the idea that the solution sequence is the least containing sequence where all processes are interleaved in time can be made to overlap, and some parallelism is retained as well, without increasing the program's burden. The one drawback in this approach is that it does not carry out nonsequential computation, i.e., computation cannot be used to produce the final outcome of the program. The effective use of computation may even involve the recursive creation of superfluous processes.

We have developed a method of extracting useful information and the effectiveness in quasi-parallel simulations. The idea is to associate with each element of the solution sequence a useful information.

The integration of the SIC St[1]. Once activated, the process effect is to create an element, i.e., to include it in the sequence, in the case that a new element is added to the sequence. Elements are represented by a linear list containing items stored in a connected list and terminating with a reference to the current process for the channel. Consumers have pointers into this list and request new elements.

In our implementation, significant time and space savings result from this transform.

3.2 Corroborative mode of operation

In this mode, activation of processes is strictly demand-driven. Since the demand must originate somewhere, access is selected to deliver an improved view of the network, and the demands of this driving process propagate through the network as a demand for transmission primitives and receive information. The selection of the driving process is based on the activation of the instruction is restricted to the initial driving process. Normally, it is that process which is responsible for issuing (e.g., printing) the ultimate outcome.

Transmission: FUTPUT + GETOUT + DICT (11).

Each GET and PUT may involve transfer of control. Assume that the empty input channel C causes suspension of the running process for C. The channel C is made hungry to indicate that there is no corresponding transfer on it. Applying PUT to a hungry output channel causes suspension of the receiving process. The consuming process is notified when the empty output channel is completed.

Remarks: (1) There is no transfer of control when GET is applied to a nonempty input channel or PUT is applied to a nonhungry output channel.

(2) If a result of a GET operation causes the scheduler to examine the process itself, further computation is impossible and deadlocks have been avoided.

(3) Reconfiguration

Except for the driving process, any process which is active is trying to synchronize with the output channel. After such a process reconfigures, the scheduler attempts to pass control to the (possibly new) process for that hungry channel.

When the driving process reconfigures, it may be further suspended. In the case that control remains, it may again be passed to the last process creating a new process, the scheduler determines the next process.

3.3 Parallel mode of execution

After a PUT instruction on an input channel, the algorithm stops incrementing. The last process created is activated.

The last process created in a reconfiguration is the outermost one in the last elementary reconfiguration.

3.4 Examples

3.4.1 Outline

In an operational point of view, a process network consists of a collection of independent machines which interact by making demands upon or sending data among communication channels. Processes are represented by data structures containing local access environment controls and global constraints, i.e., computations were interleaved in time can be made to overlap, and some parallelism is retained as well, without increasing the program's burden. The one drawback in this approach is that it does not carry out nonsequential computation, i.e., computation cannot be used to produce the final outcome of the program. The effective use of computation may even involve the recursive creation of superfluous processes.

We have developed a method of extracting useful information and the effectiveness in quasi-parallel simulations. The idea is to associate with each element of the solution sequence a useful information.

The integration of the SIC St[1]. Once activated, the process effect is to create an element, i.e., to include it in the sequence, in the case that a new element is added to the sequence. Elements are represented by a linear list containing items stored in a connected list and terminating with a reference to the current process for the channel. Consumers have pointers into this list and request new elements.

In our implementation, significant time and space savings result from this transform.

3.2 Corroborative mode of operation

In this mode, activation of processes is strictly demand-driven. Since the demand must originate somewhere, access is selected to deliver an improved view of the network, and the demands of this driving process propagate through the network as a demand for transmission primitives and receive information. The selection of the driving process is based on the activation of the instruction is restricted to the initial driving process. Normally, it is that process which is responsible for issuing (e.g., printing) the ultimate outcome.

Transmission: FUTPUT + GETOUT + DICT (11).

Each GET and PUT may involve transfer of control. Assume that the empty input channel C causes suspension of the running process for C. The channel C is made hungry to indicate that there is no corresponding transfer on it. Applying PUT to a hungry output channel causes suspension of the receiving process. The consuming process is notified when the empty output channel is completed.

Remarks: (1) There is no transfer of control when GET is applied to a nonempty input channel or PUT is applied to a nonhungry output channel.

(2) If a result of a GET operation causes the scheduler to examine the process itself, further computation is impossible and deadlocks have been avoided.

(3) Reconfiguration

Except for the driving process, any process which is active is trying to synchronize with the output channel. After such a process reconfigures, the scheduler attempts to pass control to the (possibly new) process for that hungry channel.

When the driving process reconfigures, it may be further suspended. In the case that control remains, it may again be passed to the last process creating a new process, the scheduler determines the next process.

3.3 Parallel mode of execution

After a PUT instruction on an input channel, the algorithm stops incrementing. The last process created is activated.

The last process created in a reconfiguration is the outermost one in the last elementary reconfiguration.