SEQUENTIAL EQUIVALENTS OF PARALLEL PROCESSES

BY

D. L. PARNAS

Carnegie Institute of Technology
Pittsburgh, Pennsylvania
February, 1967

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense. This contract, SD-146, is monitored by the Air Force Office of Scientific Research.

Sequential Equivalents of Parallel Processes

D. L. Parnas

ABSTRACT

This paper introduces the problem of finding a sequential process equivalent to a system of interacting discrete parallel processes. Under the assumption that the sequential process is to be composed exclusively of executions of the individual "parallel" processes in a predetermined sequence, a method of deriving optimal sequential processes is presented. Applications to the design of simulation systems and picture processing programs are discussed. Examples are taken from logic design and picture processing.

TABLE OF NOTATION

Symbol	Interpretation	Introduced on Page
Δ	Set of processes in a network	6
δ	Individual process in a network	5
Σ	Set of inputs to a process	5
σ	Individual input to a process	5
Ω	Set of outputs of a process	5
ω	Individual output of a process	5
Ψ	Set of state variables in a process	5
ψ	Individual state variable in process (examples only)	5
Υ	Algorithm describing a process	5
Ξ	Set of inputs to a network	6
ξ	Individual input to a network	6
Γ	Set of interconnectors in a network	6
γ	Individual interconnectors in a network	6
$\Re$	Set of rules describing the interconnections in a network	6
$\rightarrow$	$\omega \rightarrow \gamma \Rightarrow$ device output ω connected to interconnector γ	6
$\leftarrow$	$\sigma \leftarrow \gamma \Rightarrow$ device input σ connected to interconnector γ	6
Λ	Set of auxiliary rules describing immediate dependency of output values on input values for a device.	11
ϕ	Predicate denoting the simulation state of some part of a network	12
E_t	Denotes the execution of an algorithm simulating behavior at time t	12
$\leftarrow\!\cdot$	$\omega \leftarrow\!\cdot\ \sigma \Rightarrow$ current value of ω depends on current value of σ	12
$\sim$	Denotes restoration of state variables after simulation	15
ν	A network	6
$G(\nu)$	Simulation graph of a network	15
*	Simulation state in which all entries are marked true	17
$<\cdot$	Transitive closure of $\leftarrow\!\cdot$	21

References

[1] Knuth, D. E. and McNely, J. L., SOL - A Symbolic Language for General Purpose Systems Simulation; IEEE Transactions on Electronic Computers, August 1964.

[2] Knuth, D. E. and McNely, J. L., A Formal Definition of SOL; IEEE Transactions on Electronic Computers, August 1964.

[3] IBM Corp., White Plains, N.Y., Introduction to General Purpose Systems Simulator III.

[4] Marcovitz, A.M., Hauser, B., and Karr, SIMSCRIPT - A Simulation Programming Language; Prentice Hall, Inc., Englewood Cliffs, N.J., 1963.

[5] Dahl, O.J. and Nygaard, K., SIMULA - An ALGOL Based Simulation Language; Comm ACM, September 1966, pp.670-678.

[6] Time Sequenced Logical Simulation Based on Circuit Delay and Selective Tracing of Active Network Paths.

[7] Shalla, Leon, Automatic Analysis of Electronic Digital Circuits Using List Processing; Comm ACM, May 1966.

[8] Parnas, D., Richardson, L.C., Kohl, W.H., Preliminary Version, An Introduction to Boole-66; unpublished manual available from Computation Center, Carnegie Institute of Technology.

[9] Parnas, D.L., System Function Description - Algol; Technical Report, Carnegie Institute of Technology. (Computation Center).

[10] Parnas, D. L., A Language for Describing the Functions of Synchronous Systems; Comm ACM, February 1966.

[11] Parnas, D. L., State Table Analysis of Programs in an Algol Like Language; presented at the 21st Annual National ACM Conference and included in the proceedings of the conference.

[12] Strauss, J.C., Basic Hytran Simulation Language; 1966 FJCC Proceedings.

[13] Rosenfeld, A., and Pfaltz, J. L., Sequential Operations in Picture Processing; Journal ACM, October 1966, pp. 471-495.

[14] McCormick, Bruce, The Illinois Pattern Recognition Computer, Illiac III; IRE Transactions on Electronic Computers, June 1963.

[15] Slotnick, Daniel, The Solomon Computer, 1962 FJCC Proceedings.

Sequential Equivalents of Parallel Processes

The design of general purpose simulation languages (e.g., SOL, SIMSCRIPT, GPSS, SIMULA ([1], [2], [3], [4], [5])), involves the development of a method of simulating many simultaneous processes on a processor capable of performing only one computation at a time. The event list techniques used to perform such simulation are well known, and this paper will assume that the reader is familiar with them. These techniques are quite suitable provided that two conditions are met:

(1) The scattering of events over time in the individual processes is sparse, i.e., it is not uncommon for long periods of time to pass without an event of interest.

(2) The conflict produced by simultaneous events is either insignificant or can be resolved by programmer specified priorities.

The first condition will often guarantee the second, since simultaneous events will be rare. If only the first condition is violated there is, at worst, loss of efficiency. If the second condition does not hold, erroneous results can be obtained from the simulation.[1]

This paper deals with the problem of conflict resolution. It presents two formal characterizations of the problem which appear to be convenient and to permit the algorithmic determination of a minimal cost sequence of individual process simulations which will simulate the simultaneous occurrence of more than one event in an interconnected structure. An algorithm for determining these sequences is presented.

---

[1] The inability of these methods to deal adequately with simultaneous events leads to the characterization of SIMULA processes as "quasi parallel". [5]

In the bulk of this paper it is assumed that all of the processes are simulated whenever there might be activity in the system. Each process is assumed to be described by an algorithm which describes the actions performed by the process at such times. The presentation is done in these terms so as not to confuse two essentially separate issues. In a later section of the paper it will be shown that it is possible to approach the efficiency found in event list simulators without loss of the conflict resolution technique developed in this paper.

Example

Consider the following network:

There are three computing devices synchronously controlled by a master clock. Each of the computing devices is fast relative to the minimum spacing between clockpulses, and can, therefore, be considered as performing its computation "instantaneously".

The first device (1) has three outputs and two inputs. It (and all devices in the network) has a memory or set of state variables. Its purpose is to produce on its output lines a set of values which are specified functions of the values of the inputs and its past history as stored in the memory. The value of output 1 is a function of input 3 and the state variables, but is independent of the values of inputs 1 and 2. The value of output 2 is a function of the state variables and is independent of all input values. If we denote outputs by ω, and inputs by σ, and the state variables by ψ we may write:

$$\omega_{1,1} = f_{1,1}(\sigma_{1,3}, \psi_1(t-1)) \qquad (1)$$

$$\omega_{1,2} = f_{1,2}(\psi_1(t-1)) \qquad (2)$$

The state variables are functions of all the inputs as well as their own values at the last clockpulse

$$\psi_1(t) = f_{1,0}(\sigma_{1,1},\sigma_{1,2},\sigma_{1,3},\psi_1(t-1) \qquad (3)$$

Since the device is "fast" and the value of $\omega_{1,1}$ at a time t depends on the value of $\sigma_{1,}$ at time t, equation (1) may be usefully interpreted as a constraint on the values of $\sigma_{1,1}$ and $\omega_{1,1}$. A similar interpretation is possible for the other equations.

Device 2 and 3 can be described by the following equations using the notation just introduced.

Device 2

$$\omega_{2,1} = f_{2,1}(\sigma_{2,1},\sigma_{2,2}, \psi(t-1)) \tag{4}$$

$$\omega_{2,2} = f_{2,2}(\sigma_1,\psi(t-1)) \tag{5}$$

$$\psi_2(t)= f_{2,0}(\sigma_{2,1},\sigma_{2,2},\sigma_{2,3},\psi_2(t-1)) \tag{6}$$

Device 3

$$\omega_{3,1} = f_{3,1}(\psi_3(t-1)) \tag{7}$$

$$\omega_{3,2} = f_{3,2}(\sigma_{3,1},\psi_3(t-1)) \tag{8}$$

$$\psi_3(t)= f_{3,0}(\sigma_{3,1},\sigma_{3,2},\psi_3(t-1)) \tag{9}$$

These devices are interconnected as shown in figure 1.

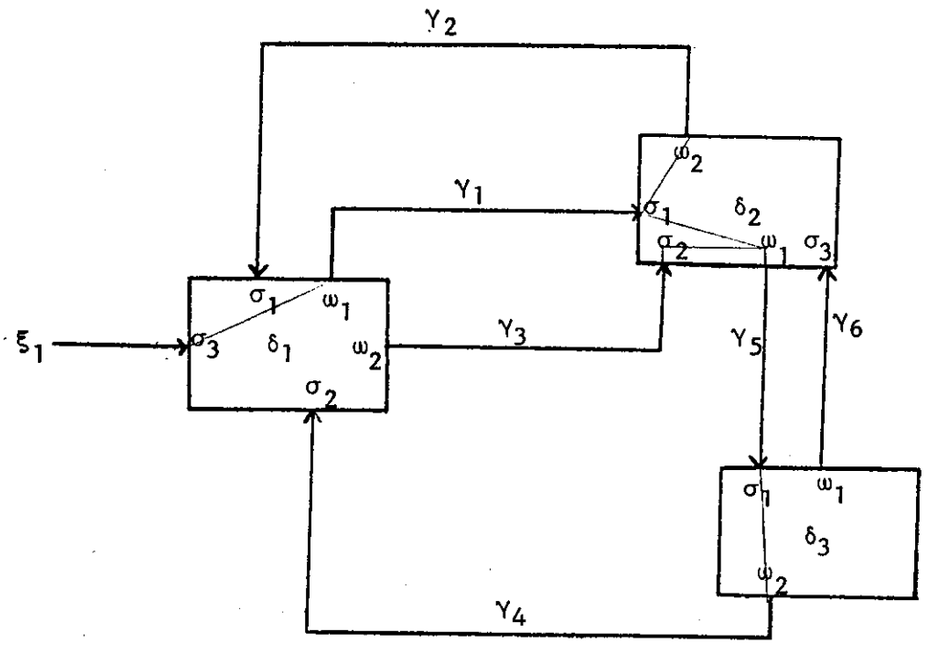


Figure 1

Input 3 to Device 1 is from some external source, and will be correct at the time that the clockpulse occurs. At that time each computing device will read its inputs, compute its state variables and outputs, but its speed is sufficiently great that it will be able to recompute these in the event that the inputs change during this time. After the correct values are determined (all constraints satisfied) the system will be idle until the next clockpulse.

The problem considered by this paper is the simulation of this parallel processing by a sequence of simulations of the individual devices. In other words, given a set of three algorithms which will compute the functions described above (and therefore simulate the devices), in what order should they be executed in order to guarantee that the values computed will be the same as those which would be computed by the actual system during one clockpulse? Further, if there is more than one correct sequence, it is desirable to find one which is, in some sense, minimal cost.

For the example shown a correct simulation sequence is:

Simulate Device 1 and restore its state variables to their old values but pass its outputs on to Device 2; simulate Device 2 restoring its state variables to their old values, passing the outputs on; simulate Device 3; simulate Device 1; simulate Device 2.

There are other correct sequences.

The derivation of such sequences from descriptions of the network is best described by the following development.

The Characterizations

General

It is assumed that it is convenient to represent a system of parallel processes, by describing each process individually, and specifying the means by which they communicate. Each process is to be described without reference to other processes in the system. Each process has a specified set of inputs

and outputs. The communication of processes then described in terms of "interconnectors". Each input or output is associated with one interconnector. Each interconnector is associated with exactly one process output but may be associated with any number of process inputs. This viewpoint seems most natural in considering hardware systems. It is not, however, restricted to such systems. Hardware devices would be described in terms of the algorithms that they implement or execute. The "interconnectors" serve only to equate the input to one such algorithm with the output of another. Any system may be viewed in this way. The reader may choose to read "processor" or "device" for "process" without loss of accuracy.

The two characterizations of the problem differ only in the way that the processes are described. In the first characterization, processes are described by a single algorithm, while in the second this algorithm is replaced by two or more algorithms, one computing the internal state of the process, the others computing the outputs. The first model is, in the opinion of the author, more natural; however, the second permits a more efficient simulation in many interesting cases. The difference between the two characterizations will be discussed in more detail in a later section.

The First Characterization

1. Processes

Definition 1

A process δ, is a quadruple $(\Sigma, \Omega, \Psi, \Upsilon)$ where Σ is a set of input variables, Ω is a set of output variables, Ψ is a set of internal state variables, and Υ is an effective algorithm such that each execution of Υ will determine new values for Ω and Ψ based on the values of Σ and previously determined values of Ψ, $\Omega \cap \Sigma = \Omega \cap \Psi = \Sigma \cap \Psi = \phi$ (the empty set). We will denote a member of Σ by σ, a member of Ω by ω, and a member of Ψ by ψ.

The quadruple describes a process or device which operates in discrete steps. The inputs are presumed to be determined externally to the process. The outputs are calculated by the process as functions of the externally determined inputs and the values of the state variables which were determined during the previous event. The requirement that the intersection of Ω and Ψ be null is for expositional clarity and is not essential.

2. The networks

Definition 2

A network, ν, is a quadruple $(\Delta, \Xi, \Gamma, \Re)$ where Δ is a set of processes, Ξ is a set of input variables (variables whose values are determined by processes outside the network), Γ is a set of interconnector variables (variables whose values are determined within the network), and $\Re$, a set of statements about the network, is described below.

We denote a member of Δ by δ, a member of Ξ by ξ, and a member of Γ by γ.

We write $\omega \rightarrow \gamma$ $((\omega \rightarrow \gamma) \in \Re)$ where ω is an output of a process δ, and γ is determined by the structure of the network to be equal to the value of ω. This may be read "ω determines γ". It is sometimes convenient to write $\delta \rightarrow \gamma$ where it is not necessary to indicate which output of δ is involved.

We write $\sigma \leftarrow \gamma$ $((\sigma \leftarrow \gamma) \in \Re)$ where σ is an input of a process δ and γ is an interconnector to indicate that the value of σ is determined by the structure of the network to be the value of γ. Again, for convenience, we may write $\delta \leftarrow \gamma$ to indicate that the process δ has an input from the interconnector γ where the particular member of Σ need not be distinguished.

We will also write $\sigma \leftarrow \xi$, $((\sigma \leftarrow \xi) \in \Re)$ where σ is a process input and ξ is a network input, to indicate that the value of σ is determined by the structure of the network to be equal to the value of ξ. The set of statements just described constitutes $\Re$.

The network is our means of describing the interconnection of the separately defined processes.[2] The interconnection is delayless, "connected" process inputs and outputs are always equal.

This type of description of a system is termed "structural", and closely corresponds to a "block diagram" picture of a system.

The following definition will be used later.

Definition 3

A network $\nu = (\Delta, \Xi, \Gamma, \Re)$ is said to be complete if and only if:

1. For each process $\delta = (\Sigma, \Omega, \Psi, \Upsilon)$ and for each $\sigma \in \Sigma$ there is one and only one $\gamma \in (\Xi \cup \Gamma)$ such that $(\sigma \leftarrow \gamma) \in \Re$.

This requires that each input to a process be connected to (or determined by) one and only one interconnector or network input.

2. For each interconnector γ in the network there is one and only one process $\delta_i = (\Sigma_i, \Omega_i, \Psi_i, \Upsilon_i)$ and one and only one output $\omega \in \Omega_i$ such that $(\omega \rightarrow \gamma) \in \Re$.

This requires that each interconnector be connected to one and only one output of a single process in the network.

If an input σ to a process δ is to be a function of two process outputs ω_1 and ω_2 this can be achieved by the introduction of a delayless process with a single output, the proper function of two inputs. The restriction that no interconnector be connected to more than one process output and no process input be connected to more than one interconnector, therefore, does not preclude any useful systems. The reader will note that an interconnector may be used as input to 0, 1, or more, processes in the network. Hereafter we shall assume that all networks are complete and will not state this explicitly.

---

[2] The set of interconnectors, Γ, may be thought of as a set of wires or patch cords, the set of inputs are wires which originate outside of the system. $\Re$ may be thought of as instructions for wiring the network connecting each wire to the output of one device and (here the analogy is slightly strained) to any number of device inputs. Γ is not strictly necessary to the development that follows, but simplifies its presentation.

We shall assume that any process in the network can be simulated individually. The problem of simulating the network then is one of finding a sequence of individual process simulations which will duplicate the behavior of the processes functioning simultaneously. This sequence can then be executed once for each event to be simulated. The sequence is clearly not arbitrary. For example, if in some network $\nu = (\Delta, \Xi, \Gamma, \Re)$ $\Re$ contains $\delta_1 \rightarrow \gamma_1$ and $\delta_2 \leftarrow \gamma_1$ then it would appear that δ_1 must be simulated before δ_2. However, our definition of network would also allow $\delta_2 \rightarrow \gamma_2$ and $\delta_1 \leftarrow \gamma_2$ to be in $\Re$. This seems to imply that δ_2 must be simulated before δ_1, an apparent contradiction.

While it would be quite convenient to rule networks containing loops (such as the one above) out of the class of well-formed networks, to do so would eliminate many useful structures. Feedback loops such as the one described are essential.

In networks of delayless devices (devices in which an output at some time τ is a function of an input at time τ) the presence of such a loop indicates that the network is ill-formed and has defined behavior only in special cases. In logic design such delayless loops are referred to as race conditions since they can result in instabilities. In the rare case where delayless loops do not cause instability and can be used, they are known in the trade as "cheat paths". In the sections immediately following it is assumed that delayless loops are undesirable, and we require that well formed networks not have such paths. In a later section there is a discussion of techniques that remove this restriction for various classes of "cheat paths".[3]

Multiple Simulations of a Process During an Event

Under the characterization of simulation problems that has been presented it is sometimes necessary to simulate one or more processes more than once during a simulated clockpulse. The following example is one such case.

[3] In several years of advising students on the use of a logic simulator which did not permit "races", the author was never shown a "race" with well defined behavior which could not be replaced by a well formed network with lower cost.

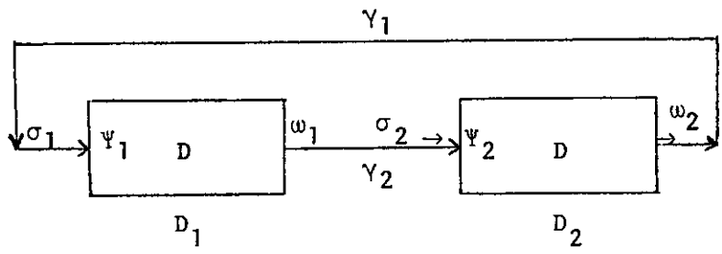


Figure 2

Each element, D_i, is a 1 unit delay device. Each has one input; σ_i one state variable ψ_i and one output ω_i. The algorithm $Y_{i_{i=1,2}}$ can be written as follows:

$\omega_i \leftarrow \psi_i$
$\psi_i \leftarrow \sigma_i$ ('$\leftarrow$' indicates an assignment in the usual sense)

If D_1 were simulated first (by executing Y_1) the value of Y_1 might be incorrectly computed, since the value of ω_2 used as the value σ_1 might not be the actual value of ω_2. A correct solution can be guaranteed by executing Y_1, restoring ψ_1 to its old value, executing Y_2, and then executing Y_1 again.[4]

Augmented Processor Descriptions

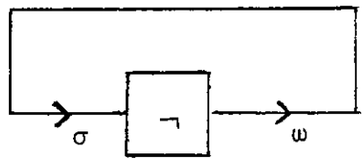
The quadruple description of processes (Definition 1) is complete in the sense of completely defining the process. We shall, however, assume that this description is augmented by certain declarations about the behavior of the process. These declarations contain no information that was not already implied by the quadruple; our insistence on a set of declarations frees us from the potentially unsolvable task of determining these properties from the description of the algorithm.[5]

[4] The alert reader may have noted that this technique is not necessary for the case noted and that a cheaper simulation is possible. This is the motivation for the second problem characterization and will be discussed later.

[5] The task is potentially unsolvable because the class of algorithms allowed is unlimited and includes, for example, algorithms describing Turing Machines, etc.

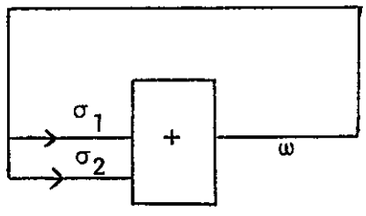
The information in these declarations is information which is needed before any device is simulated.

The behavior of a network with loops, cannot be determined without further knowledge concerning the properties of the processes involved in those loops. Figures 3 and 4 show simple networks with undefined behavior. The network in Figure 3 has undefined behavior because there is no value which satisfies the constraints; the network in Figure 4 has two possible solutions, and, unlike the example in Figure 5, there is no way to discuss the initial state, since there is no state variable. Figure 5 shows a network with a structure identical to that of Figure 3, but its behavior is completely defined (provided that the initial state of the device is specified). The difference stems from the nature of the devices in each example. The delay introduced by the device in Figure 5 is responsible for the network's behavior being completely defined.



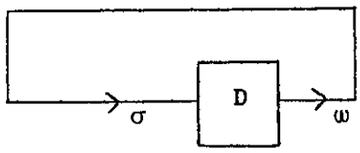
delayless device such that $\omega \leftarrow \neg\, \sigma$
ω, σ Boolean variables

Figure 3



delayless device $\omega \leftarrow \sigma_1 + \sigma_2$
(Boolean "or")

Figure 4



simple delay unit
(described earlier)

Figure 5

These considerations motivate the following definitions:

Definition 4

An augmented process δ is a 5-tuple $(\Sigma, \Omega, \Psi, Y, \Lambda)$ where Σ, Ω, Ψ, Y have the same interpretation that they had in Definition 1 and Λ is a set of rules of the form $\omega \leftarrow\cdot \sigma$ where $\omega \in \Omega$ and $\sigma \in \Sigma$.

Definition 5

The state of an augmented process $\delta = (\Sigma, \Omega, \Psi, Y, \Lambda)$ is a Boolean vector with one entry associated with Ψ and one entry associated with each member of Ω.

Definition 6

The input state of a process is a Boolean vector with one entry associated with each member of Σ.

The interpretation of these vectors is straightforward. At times during a simulation some of the inputs, outputs, and state variables of the process will be known to have correct values, while others may be either correct or incorrect values, but cannot be guaranteed correct. The state vectors describes this situation in detail.

Notation:

$\phi_t(\sigma)$ is a predicate which is true if the entry in the state vector associated with an input σ is true for time t.

ϕ_t (ω) is a predicate which is true if the entry in the state vector corresponding to ω is true for time t.

ϕ_t (Ψ) is a predicate which is true if the entry in the state vector corresponding to Ψ is true for time t.

E_t (Υ) will denote the execution of Υ using values of Ψ determined to be correct at time, t-1.

ϕ_t (ξ) (where $\xi \in \Xi$) = true.

The meaning of Λ is given by the following. We write $\omega \leftarrow\!\cdot\, \sigma$ as shorthand for $(\omega \leftarrow\!\cdot\, \sigma) \in \Lambda$.

Definition 7

For an augmented process $\delta = (\Sigma, \Omega, \Psi, \Upsilon, \Lambda)$, Λ is a set of rules of the form $\omega \leftarrow\!\cdot\, \sigma$. with the property that

$$\forall\, \omega \in \Omega\ [[\forall\, \sigma \in \Sigma\ [(\omega \leftarrow\!\cdot\, \sigma) \Rightarrow \phi_t\ (\sigma)]] \Rightarrow [E_t\ (\Upsilon) \Rightarrow \phi_t\ (\omega)]]$$

This definition indicates that Λ is a set of rules of the form $\omega \leftarrow\!\cdot\, \sigma$ such that if the correct values of all σ such that $\omega \leftarrow\!\cdot\, \sigma$ are known for a time t we can calculate the correct value of ω (at the same time) by executing Υ (using the values of Ψ computed for t-1).

Note that our definition allows Λ to contain more rules than necessary.

Finally, we define $\phi_t(\Psi)$ by the following:

$$[\forall\, \sigma \in \Sigma\ [\phi_t\ (\sigma)]] \wedge E_t(\Upsilon) \Leftrightarrow \phi_t\ (\Psi)$$

In other words, correct values for the state variables can be determined if and only if the process algorithm is executed with all input variables known to be correct.

Example

Figure 1 shows a schematic diagram of the network described earlier. The exact nature of the processes is irrelevant so Ψ and Υ for each of the processes

is unspecified. In the figure a line within the block representing a process and connecting an input to an output indicates a delayless dependency.

Figure 6 gives a description of this network and the processes in the notation just introduced. It is suggested that the reader study the diagram and description to become familiar with the notation.

$$\nu = (\Delta, \Xi, \Gamma, \Re)$$

$$\Delta = \{\delta_1, \delta_2, \delta_3\}$$

$$\Xi = \{\xi_1\}$$

$$\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6\}$$

$$\Re = \{\xi_1 \rightarrow \sigma_{13}, \omega_{1,1} \rightarrow \gamma_1, \omega_{1,2} \rightarrow \gamma_3,$$
$$\omega_{2,1} \rightarrow \gamma_5, \omega_{2,2} \rightarrow \gamma_2, \omega_{3,1} \rightarrow \gamma_6, \omega_{3,2} \rightarrow \gamma_4,$$
$$\sigma_{1,1} \leftarrow \gamma_2, \sigma_{1,2} \leftarrow \gamma_4, \sigma_{2,1} \leftarrow \gamma_1, \sigma_{2,2} \leftarrow \gamma_3$$
$$\sigma_{2,3} \leftarrow \gamma_6, \sigma_{3,1} \leftarrow \gamma_5\}$$

$$\delta_1 = (\Sigma_1, \Omega_1, \Psi_1, \Upsilon_1, \Lambda_1)$$
$$\Sigma_1 = \{\sigma_{1,1}, \sigma_{1,2}, \sigma_{1,3}\}$$
$$\Omega_1 = \{\omega_{1,1}, \omega_{1,2}\}$$
$$\Upsilon_1, \Psi_1 \text{ are unspecified}$$
$$\Lambda_1 = \{\omega_{1,1} \leftarrow\cdot \sigma_{1,3}\}$$

$$\delta_2 = (\Sigma_2, \Omega_2, \Psi_2, \Upsilon_2, \Lambda_2)$$
$$\Sigma_2 = \{\sigma_{2,1}, \sigma_{2,2}, \sigma_{2,3}\}$$
$$\Omega_2 = \{\omega_{2,1}, \omega_{2,2}\}$$
$$\Upsilon_2, \Psi_2 \text{ are unspecified}$$
$$\Lambda_2 = \{\omega_{2,1} \leftarrow\cdot \sigma_{2,1}, \omega_{2,1} \leftarrow\cdot \sigma_{2,2}, \omega_{2,2} \leftarrow\cdot \sigma_{2,1}\}$$

$$\delta_3 = (\Sigma_3, \Omega_3, \Psi_3, \Upsilon_3, \Lambda_3)$$
$$\Sigma_3 = \{\sigma_{3,1}\}$$
$$\Omega_3 = \{\omega_{3,1}, \omega_{3,2}\}$$
$$\Upsilon_3, \Psi_3 \text{ are unspecified}$$
$$\Lambda_3 = \{\omega_{3,2} \leftarrow\cdot \sigma_{3,1}\}$$

Figure 6

It is now necessary to extend the concept of simulation state to the network as a whole.

Definition 8

The simulation state of a complete network $\nu = (\Delta, \Xi, \Gamma, \Re)$ is a Boolean vector with one entry corresponding to each process in Δ and one entry corresponding to each member of Γ. The predicate ϕ is defined in a manner analogous to the previous definition. The values of the entries in the state vector are defined in the terms of the values of the individual process state vectors by the following:

For any process δ, $\phi(\delta) = \phi(\Psi)$ where Ψ is the set of state variables of δ.

For any interconnector γ, $\phi(\gamma) = \phi(\omega)$ where ω is the process output such that $\omega \rightarrow \gamma$. Note that $\phi(\gamma)$ is uniquely determined because of our assumption of completeness (Definition 3).

Finally, we can define the value of the input state vector of a process in terms of the state vector for the network by the following:

$$\forall \gamma \in \Gamma\ [\forall \delta \in \Delta\ [\forall \sigma \in \Sigma\ [\sigma \leftarrow \gamma \Rightarrow \phi(\sigma) = \phi(\gamma)]]]$$

Changes in the simulation state of a network are caused by executions of process algorithms (simulation of the process). The change made by the execution of any process algorithm is defined above provided that each process is executed while $\phi_{t-1}(\Psi)$. (This follows from the definition of $E_t(\Upsilon)$).

Figure 7 shows the state vector for the example of figures 1 and 6.

STATE VECTOR = $(\phi(\gamma_1), \phi(\gamma_2), \phi(\gamma_3), \phi(\gamma_4), \phi(\gamma_5), \phi(\gamma_6), \phi(\gamma_1), \phi(\gamma_2), \phi(\gamma_3))$

Figure 7

Definition 9

We shall define <simulation sequence> by the following: (<Δ> denotes a member of Δ).

<process reference> ::= <Δ> | $\sim$ <Δ>

<simulation sequence> ::= <process reference> |

<simulation sequence>, <process reference>.

The interpretation of a simulation sequence is:

(1) Each process reference of the form <Δ> denotes the execution of the algorithm of the process referenced.

(2) Each process reference of the form $\sim$ <Δ> denotes the execution of the algorithm of the process referenced followed by the restoring of the values of Ψ_δ to the values they held before the execution (and the restoration of ϕ (Ψ_δ) to <u>false</u>).

(3) The sequence of process references is to be read from left to right.

We are now in a position to restate the problem under consideration as:

<u>Problem</u>: Given a network $\nu = (\Delta, \Xi, \Gamma, \Re)$ and a set C of costs associated with the execution of each process algorithm, find a minimal cost simulation state from all entries <u>false</u> to all entries <u>true</u>.

<u>Solution</u>: (1) Construct a simulation graph, $G(\nu)$, by the following rules:

(a) There is an initial node 0 labeled with the "<u>false</u> state" (all entries <u>false</u>).

(b) From each node n labeled with simulation state Φ draw an arc outward for each $\delta \in \Delta$ such that $\phi(\delta)$ = false.

(c) Label the terminal node of each arc drawn in (b) with the simulation state that would result from simulating the process δ with the network in simulation state Φ. This state, Φ', can be determined from definitions 5 - 8. (All entries not <u>true</u> by those definitions are <u>false</u>). If $\phi'(\delta)$[6] = <u>true</u> then label the arc δ else label the arc $\sim \delta$.

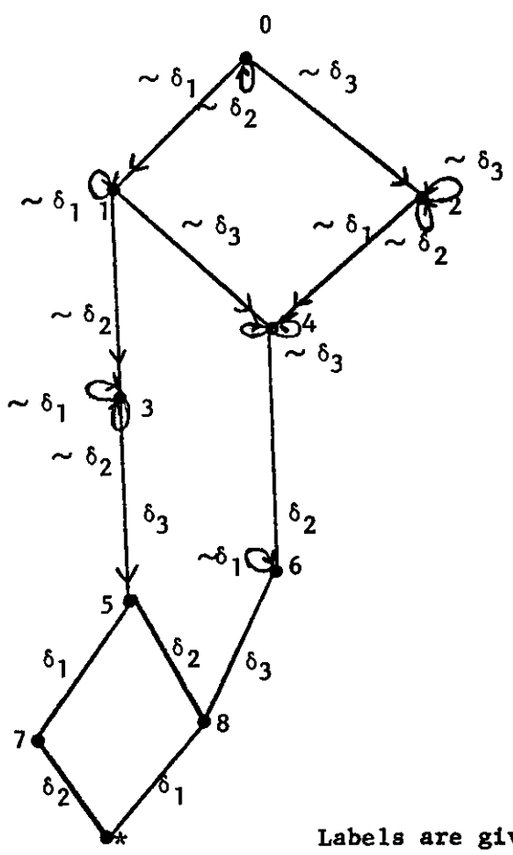
[6] $\phi'(\delta)$ is the entry in Φ' corresponding to δ.

(d) If there already is a node with label ϕ' combine it with that just produced.

(e) Continue the process (go to (b)) until there are no nodes which have not been used in (b).

The above process must terminate since the number of possible nodes is finite. Figure 8 shows the graph obtained by applying the above to the example of Figure 1. Node labels are given in Table 1.

FULL GRAPH $G(\nu)$ for Example of Figure 6



Labels are given in Table 1.

Figure 8

Table 1

Labels of Nodes in Figures 8 and 9

NODE	LABEL
0	(f, f, f, f, f, f, f, f, f)
1	(t, f, t, f, f, f, f, f, f)
2	(f, f, f, f, f, t, f, f, f)
3	(t, t, t, f, t, f, f, f, f)
4	(t, f, t, f, f, t, f, f, f)
5	(t, t, t, t, t, t, f, f, t)
6	(t, t, t, f, t, t, f, t, f)
7	(t, t, t, t, t, t, t, f, t)
8	(t, t, t, t, t, t, f, t, t)
*	(t, t, t, t, t, t, t, t, t)

Note: the above vector is

$$\phi(\gamma_1),\ \phi(\gamma_2),\ \phi(\gamma_3),\ \phi(\gamma_4),\ \phi(\gamma_5),\ \phi(\gamma_6),\ \phi(\gamma_1),\ \phi(\gamma_2),\ \phi(\gamma_3))$$

(2) We can now state several obvious theorems.

Theorem 1. If there exists a correct simulation sequence there will be a node in the graph which we label * such that all entries of * are true.

Theorem 2. Any path from 0 to * represents a correct simulation sequence.

Theorem 3. If the length of each arc is the cost of performing the associated simulation then the shortest path from ϕ to * represents a minimal cost correct simulation sequence.

Proof: The theorems follow directly from the definitions.

It follows from the above, that the combination of a shortest path algorithm with the graph generation algorithm described above would generate solutions to our problem. There are, however, several theorems which allow the simplification of the graph and result in greater efficiency.

Theorem 4. No minimal path will contain an arc originating and terminating on the same node (loop).

Proof: One always obtains a shorter path by deleting the loop.

Lemma 1. If all loops are removed from G then G is not cyclic.

Proof: It follows from our definitions that the simulation of a process never changes a true entry to false. If the state changes, at least one entry is changed from false to true, and it is never possible to return to the previous state.

Theorem 5. For any node in G from which originate two arcs labeled δ_1 and δ_2, (1) the node reached by the arc labeled δ_1 will have an arc labeled δ_2 originating there, and vice versa; (2) the node reached by taking the arc labeled δ_1 and then the arc labeled δ_2 will be identical to the node reached by taking the arc labeled δ_2 and then the arc labeled δ_1.

Proof: The existence of both arcs at the original node implies that all inputs to both processes are marked true. Each process changes a fixed set of entries in the state vector, and those sets are disjoint. The order of simulation is, therefore, irrelevant.

Corollary: If at the node described above, minimal path includes the arc labeled δ_1 followed by the arc labeled δ_2, then there is another minimal path using the arc labeled δ_2 followed by the arc labeled δ_1.

Theorem 6. If a node n in G is on a minimal path from ϕ to * and an arc with a label of the form $\langle\Delta\rangle$ passes through that node then there is a minimal path which includes that arc.

Proof: By the previous theorem and corollary we need not consider any other arcs with labels of the form $\langle\Delta\rangle$. Let us consider an arc labeled δ_1 and leading to node m, and an arc labeled $\sim \delta_2$ and leading to node m'.

It is clear that an arc labeled δ_1 will originate at m', while an arc labeled either δ_2 or $\sim \delta_2$ will originate at m. Let the first arc lead to p', the second lead to p. It is clear every entry marked <u>true</u> in the label of p' will also be marked <u>true</u> in the label of p since all entries determined by δ_1 will be <u>true</u> at both and at least as many of the entries associated with δ_2 will be <u>true</u> at p (possibly more since some of the inputs of δ_2 may have been determined by outputs of δ_1). Let us call two paths which have the same sequence of arc labels <u>parallel</u>. If there is a minimal path leading through p' to * then there must be a parallel path leading through p to *. The theorem now follows by noting that both paths have the same length. Figure 9 illustrates the various arcs and nodes mentioned in this proof.

Illustration for Proof of Theorem 6

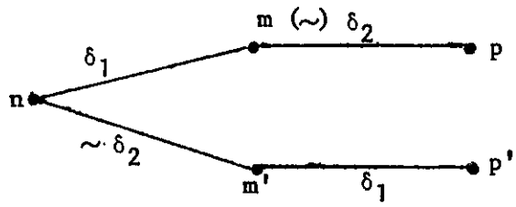
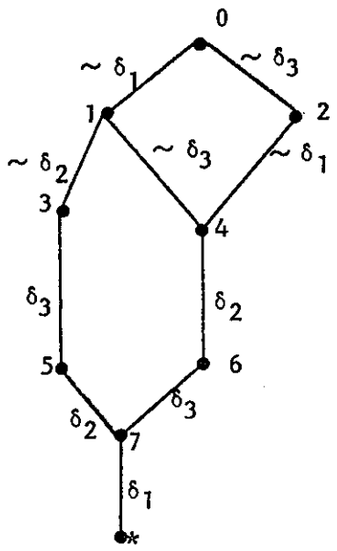


Figure 9

<u>Corollary</u>: If we obtain from G a graph G' by (1) deleting all loops, and (2) deleting all arcs except one with a label of the form <Δ> from nodes which have an arc so labeled, the resulting graph will have a minimal path no longer than the minimal path of G, and it will represent a correct simulation sequence.

Figure 10 shows a reduced graph for the example of Figures 1 and 6. Figure 11 shows correct simulation sequences for that example.

REDUCED GRAPH for Example of Figure 6



Labels are given in Table 1.

Figure 10

Correct Simulation Sequences for Example of Figures 1, 6.

$\sim \delta_1, \sim \delta_2, \delta_3, \delta_1, \delta_2$ (not found in reduced graph)

$\sim \delta_1, \sim \delta_2, \delta_3, \delta_2, \delta_1$

$\sim \delta_1, \sim \delta_3, \delta_2, \delta_3, \delta_1$

$\sim \delta_3, \sim \delta_1, \delta_2, \delta_3, \delta_1$

Figure 11

It is of interest to define the class of networks for which the above technique is useful. This is clearly the class of networks for which the graph G contains a node * as defined earlier.

Definition 10

We shall write $\gamma_1 \leftarrow\!\cdot\, \gamma_2$ if there exists a process δ with input σ and output ω such that $\sigma \leftarrow \gamma_2$; $\omega \leftarrow\!\cdot\, \sigma$; $\omega \rightarrow \gamma_1$.

Example: In Figure 6 $\gamma_5 \leftarrow\cdot\ \gamma_3$

Definition 11

We shall write $\gamma_i \lessdot \gamma_j$ if there is a sequence of interconnectors $\gamma_0, \gamma_1 \cdots \gamma_n$ such that $\gamma_0 = \gamma_i$ and $\gamma_n = \gamma_j$ and $\gamma_i \leftarrow\cdot \gamma_{i+1}$ for $i = 0 \ldots n$. This relation is clearly transitive.

Example: In Figure 6 $\gamma_4 \lessdot \gamma_3$

Theorem 7. For any network $\gamma = (\Delta, \Xi, \Gamma, \Re)$, there is no γ in Γ such that $\gamma \lessdot \gamma$ if and only if the graph $G(\nu)$ contains a node * as defined earlier, and conversely.

Proof: If at node n, $\phi(\gamma)$ is true then, at the origin of all arcs terminating at n, it is true that $\forall\ \gamma \in \Gamma \mid \gamma' \lessdot \gamma\ [\phi(\gamma)]$. If there exists γ such that $\gamma \lessdot \gamma$ then the node 0 had $\phi(\gamma)$, because the graph was built by drawing nodes outward from 0 and is non cyclic. It follows that the existence of the node * implies that there is no γ such that $\gamma \lessdot \gamma$. To show the converse, we assume that there is no node * in G. This implies that there is at least one γ such that there is no node where $\phi(\gamma)$. This can only be so if there is some γ' for which this is true and for which $\gamma \leftarrow\cdot \gamma$. If $\gamma' = \gamma$ then we are done; if not, we repeat the process finding still another γ'' such that there is no node where $\phi(\gamma'')$ and $\gamma' \leftarrow\cdot \gamma''$. Since there are only a finite number of elements in Γ we must either find a γ previously found or one such that $\gamma \lessdot \gamma$. In either case we are done, having found a γ such that $\gamma \lessdot \gamma$.

The Second Characterization of the Problem

It is possible to substitute the following for Definition 1.

Definition 1a

A process δ, is a quintuple $(\Sigma, \Omega, \Psi, \Upsilon_1, \Upsilon_2)$ where Σ is a set of input lines, Ω is a set of output lines, Ψ is a set of internal state variables, Υ_1 is an algorithm such that each execution of Υ_1 will determine new values for Ω based on the values of Ψ and Σ, and Υ_2 is an algorithm such that each execution of Υ_2 determines new values for Ψ in terms of Σ and previously determined values of Ψ.

In other words, we can decompose the algorithm Υ into two algorithms, one computing the output values of the device, the other computing its internal state. If the costs of executing each of the sub-algorithms is not as high as the cost of executing the whole algorithm (i.e., if the algorithm can be decomposed in some meaningful way) then it is usually possible to find cheaper simulation sequences because it is never necessary to evaluate the internal state incorrectly and then restore it.

The reader should have no difficulty in extending the definitions and techniques used in the first characterization to this second characterization of the problem. It is still possible in this characterization for multiple simulations of Υ_1, the output algorithm, to be needed. The state graph technique is again applicable to finding minimal solutions and can, in fact, be used to find minimal solutions for networks containing a mixture of devices described both by single and double algorithms.

Other Characterizations

If there are separate algorithms for each device output, then each algorithm could be evaluated once and only once. Each algorithm would be evaluated once all the inputs on which it depended were determined. In other words the

more information about the internal structure of a process available, the more efficient the simulation possible.

Efficient Handling of Idle and Waiting Devices

One of the soundest objections to the method of simulating parallel processors described above, is that it is inherently more costly than the "predictive" methods used in such simulation systems as SIMSCRIPT, SOL, and GPSS. Whereas the method described here executes each device at least once whenever an event might occur, and often more than once, the other systems predict future events and will often "skip" long periods of time during which the devices in the system have been predicted to be inactive. This often results in substantial economies. It is the purpose of this section to show that the efficiency of such systems can be approached by combining event list techniques with the method of this paper.

The methods used in the prediction based simulation systems do not resolve the problem of simultaneous events. To assure proper simulation of networks in which the effects discussed in this paper are important, a structural analysis such as is used here, is replaced by great effort on the programmer's part.

The ability to take advantage of inactive or waiting processes can be added to the methods discussed in this paper. It must be remembered that the processes themselves can be described in any language, including languages which allow statements such as SOL's wait and wait until. (SOL itself could be used for description of processes within the network). This information can be communicated to the structural analysis algorithm, which will respond by using as the initial node in $G(\nu)$ a node at which δ and all its outputs are marked true. The rest of the system can be simulated in an optimal manner. In effect this process and its output will be considered external to the system. If all the processes in a system are predicted to be inactive for a length of time, the time can be skipped just as it is in the predictive systems. When-

ever these are simultaneous events, correct resolution of timing conflicts is guaranteed.

Asynchronous Networks of Processes

In the discussion above it was assumed that all of the processes in the network were synchronously controlled. Many systems of interest consist of asynchronous processes. The simplest case, a process whose clock rate is lower than that of other devices in the network as a whole can be considered to be inactive for certain clockpulses, and can be handled just as the "waiting" devices discussed above. For more complex cases the following is relevant.

Time of Sequence Determination

The method of sequence determination developed in this paper is fairly involved and could involve considerable computation for large highly connected networks. It is usually desirable that the number of times that this is done be minimized. In the simplest case, that in which every process must be simulated at every clock pulse, the ordering can be just once at "compile" time. In networks of processes that are controlled by individual clock rates as discussed above, it is possible to compute "activity patterns" at compile time (i.e., before any simulation is performed) if the clock rates are fixed. In such cases one can compile a set of simulation sequences, one for each activity pattern, and execute them as the activity patterns occur. In networks where the inactive periods of the processes are not constant, but the length of the inactive period is determined at the start of the inactive period, the event list techniques become appropriate. It will probably prove economical to compute a simulation sequence as if all processes were active when simulating an event, the inactive processes can be eliminated, and the simulation sequence simplified. The resulting sequence may not be optimal but the suboptimal solution will often be preferable to the expensive search for an optimal solution wherever simultaneous events occur. It is clear that the relative costs of such approach-

es depend on the nature of the network being simulated.

In the event that some devices in the network are inactive for indefinite periods (i.e., until some external event occurs), the technique used to implement the wait until statement in SOL [3], [4] becomes necessary.

Dynamic Networks

Implicit in the above discussion has been the assumption that the number of processes and their interconnection is fixed over time. Such an assumption does not hold in languages such as SOL (or GPSS) where transactions may enter or leave the system during a simulation. Further, one can conceive of the interconnections changing in some systems being simulated. There is, however, no reason why the above conflict resolution technique cannot be applied to such networks during the simulation provided that the necessary information about the processes is available. An instantaneous view of a SOL simulated system is a network as defined earlier. The active transactions and facilities correspond to processes, the global variables to interconnectors.[7]

Resolution of "Race Conditions"

Figure 12 shows a simple example of a "cheat path" or permissable race condition in a logic network. Inspection will show that $\gamma_4 \lessdot \gamma_5$ and $\gamma_5 \lessdot \gamma_4$.

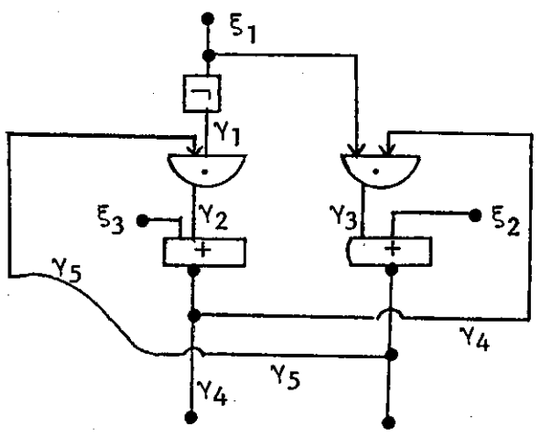


Figure 12

[7] The network may not be complete, however; this is usually either easily corrected or a programming error in the SOL program.

Although this is a race condition in the sense of the above discussion (delayless loop) it does not result in undefined behavior because $\gamma_5 \lessdot \gamma_4$ only if $\xi_1 = 1$ and $\gamma_4 \lessdot \gamma_5$ only if $\xi_1 = 0$. For logic circuits it is possible to derive the conditions on the immediate dependency rules, Λ, but in the more general situation under disucssion this is not possible. If we are given conditional statements as elements of Λ for each device, we can resolve the race and produce a correct simulation sequence at run time. A restricted class of networks (including the example) could be resolved at compile time by noting that the conditions were mutually exclusive. Another example in which such loops are dealt with is given later.

Implications to Logic Circuit Simulation

Two recent papers [6], [7] have discussed the simulation of logic networks described by structural descriptions similar to the network descriptions used in this paper. It should be clear that such networks are a special case of the networks discussed in this paper. Both papers referenced described methods which involved considerable analysis of the structural description during the running of the simulation. The first of these papers [6] is oriented towards gaining efficiency by simulating only the active elements. The other [7] gains efficiency by suppressing simulation of redundant logic. In both cases the overhead involved in suppressing the unnecessary simulations is sufficient to suggest that a cost analysis might disclose a wide class of networks in which it was more efficient to simulate every element every time. However, by applying the results of this paper it is possible to achieve the desired effects at a lower run time overhead than the methods described in either paper. Both techniques perform much of the structural analysis at run time. The nature of the logic elements is such that the multiple algorithm representation of the devices is simple and natural. It is then quite easy, at compile time, to derive a simulation sequence (on the assumption that all

devices are active) which involves execution of no algorithm twice. Further, one can derive, for each output, the set of algorithms which use its value. Compiled with each algorithm is a scheduling routine which enters the affected algorithms on the event list. For each event time the event list consists of a Boolean vector with one entry for each algorithm, that entry indicating that the corresponding algorithm is to be executed if true. Note that if times when the network is completely inactive are rare, the event list should not be a list at all but a table such as that used in [6]. At each event one then proceeds down the simulation sequence executing only those algorithms scheduled for that time. This method involves no structural analysis of any sort during the simulation run. Further, in such a system, it would be quite easy to provide an option allowing the user to select either selective simulation (simulation of only elements which might change) or simulation of every element every time according to the economies of the situation. It is conceivable that the system might make such a choice on the basis of its experience with the circuit being simulated. The time required to simulate every element every time is easily calculated. If the average simulation time using selecting simulation rises above this, the system reverts to the simpler method.

Compatability of Device Descriptions

Since the algorithm for determining proper simulation sequences is independent of the language in which the processes are described (except for the set Λ), it is possible to simulate the interaction of processes which have been described in different languages. We are currently developing a simulation system, SODAS, in which it will be possible to describe processes in a version of SOL, BOOLE [8], SFD-ALGOL, [9], [10], [11], a register transfer language for computer description, a version of Continuous System Simulation Language [12], and the SODAS language itself. Each of these languages has definite advantages for certain types of systems, and we hope to make the features of all of them

available to those interested in describing and simulating systems containing subsystems of more than one type.

An Example - Picture Processing

A recent paper by Rosenfeld and Pfaltz [13] discusses picture processing by parallel and sequential computers. The authors are concerned with picture transformations which can be performed by a large number of local[8] operators. These operators could conceivably be performed either in a definite sequence by a conventional computer or in parallel by a network of parallel processors (e.g., Solomon type units [15] or units such as PAU on ILLIAC III [14].

The authors point out that while picture processing work is often described in terms of parallel local operations, much of the work is actually done by simulating the parallel computer systems by a sequence of local operations performed on a sequential computer.

They go on to describe the method used to simulate a parallel computer on a conventional machine. They then show sequential local operations which will perform the same picture transformations much more efficiently than the simulated parallel method.

In the following we shall examine the distance transforms of [13] and it will be shown that the method of this paper yield a method simulating the parallel processing situation which is slightly more efficient than that invented by the authors of [13]. The following quote from [13] describes the transform, and the method used in [13] to produce it.

† "4.1 Distance

> Let P and Q be any two distinct points in a digitized picture, and let d*(P,Q) be the smallest positive integer such that there exists a sequence of distinct points $P = P_0, P_1, \ldots, P_n = Q$ with

[8] A local operation is one which calculates a new element of a picture using only the value of that element and the values of a small number of neighbors.

† Reprinted with the permission of ACM.

P_i a neighbor of P_{i-1}, $1 \leqq i \leqq n$. This d* is called the distance from P to Q; if $P = Q$, the distance between them is defined as zero. The distance from P to a given subset S of the picture is defined as the smallest of the distances from P to the points in S.

Like connectivity, the distance concept is defined by iterating the property of being a neighbor. Here, however, the minimum number of iterations required to 'reach' Q from P is of interest, whereas in the case of connectivity, the question considered was whether Q could be reached at all from P using only points in a given subset as intermediate points. As was pointed out for connectivity in Section 3.1, a distance can also be defined using only horizontal and vertical neighbors as 'steps.' If d(P,Q) is the distance from P to Q using this more restricted definition, is is clear that $d \geqq d^*$. For simplicity, the restricted definition is used in the remainder of this paper.

Evidently, d(P,Q) (and similarly for d*) has all the properties of a metric[4]. It should be emphasized, however, that d is not even approximately the Euclidean distance. In fact, the locus of points at a given distance $d > 0$ from a given point P is a diagonally oriented square of side D+1 centered at P, rather than a circle[5].

4.2 Distance transformation

Given a digitized picture whose elements have only the values 0 and 1, it is desired to construct a distance transform of the picture in which each element has an integer value equal to its distance from the set of 0's. (It is assumed that the set of 0's is nonempty.) Thus in particular, the 0's remain unchanged, since they are at zero distance from themselves; the 1's which are horizontal or vertical neighbors of 0's also remain unchanged; the 1's which are horizontal or vertical neighbors of such 1's become 2's; and so on.

This transform can be performed using just two sequentially applied local operations as follows. Let

$$f_1(a_{i,j}) = 0 \qquad \text{if } a_{i,j} = 0$$
$$= \min (a_{i-1,j} + 1,\ a_{i,j-1} + 1) \qquad \text{if } (i,j) \neq (1,1) \text{ and } a_{i,j} = 1,$$
$$= m + n \qquad \text{if } (i,j) = (1,1) \text{ and } a_{1,1} = 1,$$
$$f_2(a_{i,j}) = \min (a_{i,j},\ a_{i+1,j} + 1,\ a_{i,j+1} + 1).$$

Since no two points of the picture can be distance m+n apart, we know that $a_{1,1}$ is at a distance less than m+n from the set of 0's, if this set is nonempty; thus the final value of $a_{1,1}$ (or of any other element labeled m+n by f_1) will be the value assigned to it by f_2.

THEOREM. Let $C = (c_{i,j})$ be the picture which results when f_1 is applied to the picture $A = (a_{i,j})$ in forward raster sequence, followed by f_2 in reverse raster sequence. Then C is the distance transform of A.

PROOF. Note first that if $a_{i,j} = 1$ and a horizontal or vertical

---

[4] It is positive definite by definition, and is clearly symmetric (the reversal of a sequence from P to Q is a sequence from Q to P and vice versa). Moreover, since any two sequences from P to Q and Q to R, respectively, can be put end to end to give a sequence from P to R, it evidently satisfies the triangle inequality.

[5] For the metric d*, the corresponding locus is an upright square of side 2d+1 centered at P.

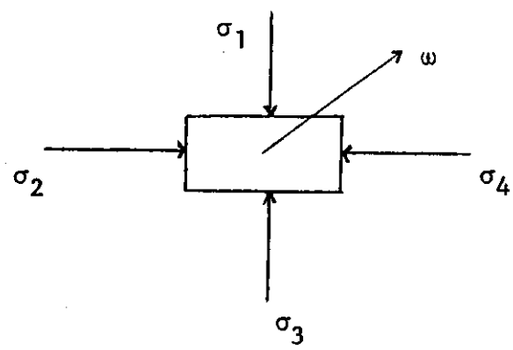
neighbor of $a_{i,j}$ is zero, evidently $c_{i,j} = 1$, and conversely. Suppose now that $c_{i,j}$ is equal to the distance from the (i,j) element to the closest zero element in A for all (i,j) such that this distance is less than k. Let $B = (b_{i,j})$ be the picture which results from applying f_1 in forward raster sequence to A. If $c_{i,j} = k$, by the induction hypothesis the distance from the (i,j) element to the nearest zero must be at least k. If it is greater than k, by definition of distance it must be at least k for each of the (i,j) element's horizontal and vertical neighbors. In particular, $c_{i+1,j}$ and $c_{i,j+1}$ each are greater than or equal to k, so that $c_{i,j} = k$ implies $b_{i,j} = k$ by definition of f_2. But then $b_{i-1,j}$ or $b_{i,j-1}$, say the former, must be k-1 by definition of f_1, so that $c_{i-1,j} \leq k-1$; contradiction.

The distance transforms for a circle, two rectangles[6] and regions F, J and K of Figure 3 are shown as Figure 4. These transforms illustrate the output of an IBM 7090/94 program, written in FORTRAN, which accepts input digital picture data as described in Section 3.3. For simplicity, only the odd distance values are printed out modulo 10, while the even values are left blank; the points with value zero are printed as X's.

---

[6] The two rectangles actually have the same proportions; the difference between their shapes in the figure results from the unequal horizontal and vertical size of a character space. The circle appears distorted for the same reason."

The parallel computer in [13] is a set of devices each one of which is assumed to introduce a one unit time delay between input (from neighboring points) and output. Such a parallel computer requires m+n execution cycles to produce the transform (m,n) are the dimensions of the picture). Under these conditions a simulation of the parallel computer will require m+n executions of a simulation sequence which consists of one simulation of each of the m×n computing elements. We can, however, assume that the computer consists of idealized delayless devices defined by the following:



$\delta = (\Sigma, \Omega, \Psi, Y, \Lambda)$

$\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$

$\Omega = \{\omega\}$

$\Psi = \{\psi\}$ (initial value of picture) Ψ=0 or m+n+1

$Y = \omega \leftarrow \min(\psi, \sigma_1+1, \sigma_2+1, \sigma_3+1, \sigma_4+1)$

$\Lambda = \{\omega \leftarrow\!\cdot\, \sigma_1, \omega \leftarrow\!\cdot\, \sigma_2, \omega \leftarrow\!\cdot\, \sigma_3, \omega \leftarrow\!\cdot\, \sigma_4\}$

Figure 13

It is clear that an array of such devices as described would not satisfy the conditions of Theorem 7 because of "race" conditions. We can, however, attach "conditions" to the elements of Λ as follows:

$\Lambda = \{\omega \leftarrow\!\cdot\, \sigma_1$ if $\sigma_1 < \omega$, $\omega \leftarrow\!\cdot\, \sigma_2$ if $\sigma_2 < \omega$, $\omega \leftarrow\!\cdot\, \sigma_3$ if $\sigma_3 < \omega$, $\omega \leftarrow\!\cdot\, \sigma_4$ if $\sigma_4 < \omega\}$.

A typical element of the array would be connected as shown in Figure 14.

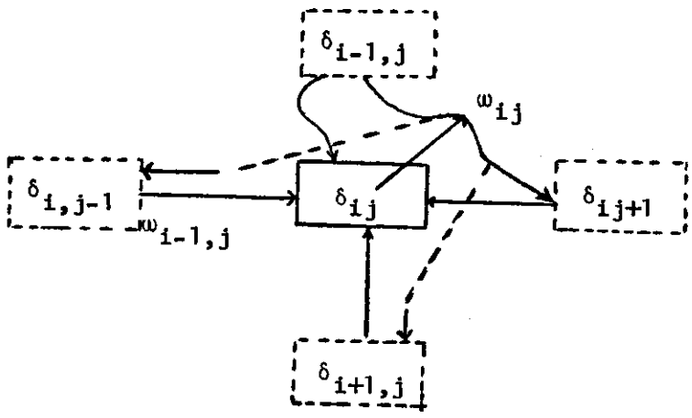


Figure 14

The conditions on a race involving the (i,j) element and the (i,j-1) element would be

$$\omega_{i-1,j} \leftarrow\cdot \omega_{i,j} \text{ if } \omega_{i-1,j} > \omega_{i,j}$$

$$\omega_{i,j} \leftarrow\cdot \omega_{i-1,j} > \omega_{i-1,j}$$

$$\text{race} = (\omega_{i-1,j} \leftarrow\cdot \omega_{i,j}) \wedge (\omega_{i,j} \leftarrow\cdot \omega_{i-1,j}) = \underline{\text{false}}$$

Similar arguments can be made for any pair of adjacent elements. There remains the possibility of a delayless loop of greater length as sketched in Figure 15 (double arrows indicate immediate dependence).

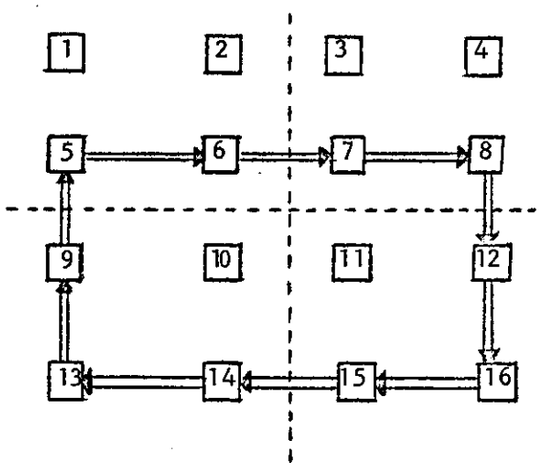


Figure 15

Because $\lessdot$ is a transitive relation such a loop implies race conditions between adjacent elements.

<u>e.g.</u>, $\delta_{12} \lessdot \delta_8 \cdots \lessdot \delta_{15} \lessdot \delta_{16} \Rightarrow \delta_{12} \lessdot \delta_{16}$ (by transitivity)

and $\delta_{16} \lessdot \delta_{12}$. (directly from Figure 15)

Thus, we see that there are no race conditions in the network, If we were to search for a simulation sequence at run time, when the picture values were available and the members of Λ become unconditional the methods developed earlier would apply directly. This is clearly not economical.

The alternative is to search, at compile time, for a procedure which will guarantee correct simulation <u>for any picture</u>.

It is necessary to assume that there is at least one zero element in the picture (as was done in [13]). The distance from the nearest zero element is undefined where there is no such element.

The approach we shall take is to use the conditions on elements of Λ as entries in the state vector. An entry in the state vector will no longer be a Boolean value but a Boolean expression which would yield this value when evaluated for any particular picture. Our assumption of at least one zero allows us to obtain a state in which all entries must be true for any picture. This will be used as the node * in $G(\nu)$.

The conditions for a particular element's output are: The output of ω_{ij} is correct if

(1) the picture element (i,j) is 0

or (2) a neighboring element, e.g., $\{(i-1,j),(i+1,j),(i,j-1),(i,j+1)\}$ is a 0 and the element (i,j) was simulated

or (3) a neighboring element

(a) had its correct value the last time that element (i,j) was simulated, and

(b) that value was equal to the smallest input to (i,j).

Note that (2) is a special case of (3).

The conditions for any neighbor element e' being correct of course are the same as those above. A correct neighbor e' will supply a minimal input to element e if there exists a zero point,z,in the picture such that

(1) z is one of the closest zeroes to e'

and (2) z is one of the closest zeroes to e

and (3) there exists a sequence $E = \{e_1,e_2,e_3 \ldots e_n\}$ of picture elements such that

(a) all have correct values

(b) e_{i+1} is a neighbor of e_i for all i

(c) z is a neighbor of e_1

(d) e_n is e'

(e) (n+1) is the distance from z to e.

The above considerations lead to the following:

(1) In an array of picture elements of size m×n the simulation state vector will consist of m×n Boolean expressions, E_{ij} (one corresponding to each element)

(2) the Boolean expression for each element will be the conjunction m×n Boolean variables, z_{ij} (again, one for each element)

(3) each of the Boolean variables, $z_{i,j}$ in Ei'j' will be <u>true</u> if the element (i',j') would be correct if the element (i,j) were the nearest zero.

An element (i',j') will be considered unconditionally correct when all the entries $z_{i,j}$ in $E_{i',j'}$ are <u>true</u> since at least one element in the array must be the nearest zero.

The network will be considered correctly simulated when all elements are unconditionally correct.

The above procedure has a simple implementation on a binary digital computer. The state vector consists of an m×n array V of vectors of m×n bits. Initially, there is one bit "on" in V[I,J], that corresponds to element (i,j). For each of the m×n elements there is a mask for each neighbor which contains a 1 corresponding to each point (i',j') such that a minimal path from (i,j) to (i',j') includes that neighbor[9]. Simulation of the element (i,j) then is represented by

$$V[I,J] \leftarrow (UP[I] \wedge V[I-1,J]) \vee (DOWN[I] \wedge V[I+1,J])$$
$$\vee (LEFT[J] \wedge V[I,J-1] \vee (RT[J] \wedge V[I,J+1]) \vee V[I,J]$$

[9] inspection will show that there are only 2×(m+n) distinct masks

where UP, DOWN, LEFT, RT are the masks appropriate to each of (i,j)'s four neighbors, and "$\vee$" and "$\wedge$" are vector operators that perform Boolean "or" and "and" on corresponding components of vectors.

Using this method for computing the state vectors one can again generate the graph $G(\nu)$ as discussed earlier and find optimal simulation sequences. The graph, however, is too large for practical considerations. It can be cut down considerably by making use of symmetry within the array.

The results of this technique on very small pictures are quite good. For a 3 x 3 array as shown in Figure 16.

1	2	3
4	5	6
7	8	9

Figure 16

The forward and backward raster sequence [13] requires 18 executions of the functions defined in [13] or 16 executions of the functions given earlier here. However, the sequence

2, 8, 5, 2, 4, 6, 8, 1, 3, 7, 9

requires only 11 applications of those same functions and guarantees correct results.

Similarly, for a 4 x 4 picture (Figure 17)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 17

A sequence of length 22 will replace the 32 operations required by the technique in [13].

The advantage is less for larger pictures and in all likelihood the raster sequences used in [13] are to be preferred since the sequencing control can be executed more efficiently. As the picture gets larger, the number of element simulations approaches the $2\times m\times n$ required by the double raster sequence.

For certain types of pictures there are enough "zeroes" that an approximation may be justified. This can be done by assuming that no point is more than some fixed distance to a zero. To find an optimal sequence under this approximation one need only choose the initial point in the graph $G(\nu)$ so that all points more than that distance away from (i,j) are already marked true in the expression for (i,j).

The main limitations on this technique is that it is not as inventive as the authors of [13]. It can find sequences of the functions used by the parallel elements but cannot recognize situations where some other local operation is appropriate. At best this technique can aid a human picture processor by allowing him to confine his inventiveness to the design of parallel processes.

Conclusions

The aim of this work has been the development of an algorithm which will derive an efficient sequential process equivalent to a given network of interconnected "truly parallel" processes. This has been accomplished for a class of networks; those in which all rules indicating immediate dependency are unconditional and there are no delayless loops. The method can be extended to a broader class of systems; those in which there are no delayless loops, but the immediate dependency rules are conditional. In such cases it is necessary to find a suitable condition which is always true; given this it is possible to use the method derived for the first class by generalizing the notion of "state vector" to allow the entries to be Boolean expressions.

It appears quite easy to generate networks in which the graph used in searching for optimal solutions becomes too large even with the aid provided by Theorems 4 to 7 and the corollaries. For such systems the methods of this paper can provide both a means of verifying the correctness of simulation sequences proposed elsewhere or a means of generating suboptimal correct sequences by making arbitrary or heuristic choices of processes to be simulated in simulation states where several paths could be explored.

The fundamental weakness, if it can be called that, of the method lies in our early assumption that the simulation sequence must consist of executions of the individual process algorithms. Any equivalent sequential processes involving the applications of other algorithms will not be considered. It appears that attempts to correct this depend on further results in the area of "equivalence of computations".

Considerable effort in this paper has been devoted to an application to picture processing. Picture processing was not within the scope of the research reported in this paper. The material included within this paper is included as an illustration of method, not as a contribution to picture process-

ing. Although the simulation sequences which can be derived for the distance transform considered, are theoretically better than those reported earlier, it appears that this improvement will not be significant for a practical picture processing task. It does appear, however, that this paper offers a methodology which should be considered by researchers doing further work along the lines suggested by [13].

Acknowledgement

The author is indebted to his colleagues, A. Ginzburg, A. Newell, A. Perlis, Tim Standish, and Jon Strauss for their patient and helpful comments on early drafts of this paper.