A parallel process definition and control system

by DAN COHEN
Harvard University
Cambridge, Massachusetts

INTRODUCTION

(1) General

The purpose of this work is to supply a simple method for definition and efficient control of a network for asynchronous parallel processing.

The system is able to compile a definition of a network of processes which run partially in parallel and partially sequentially, into a set of control instructions for some monitoring process, to be executed at run time.

The defined network is composed of a set of interrelated processes and a monitoring process which initiates processes according to some dynamic conditions. Typical resources for processes are processors, I/O devices, memory banks, and bulk storage.

The system discussed below is concerned with two tasks, the handling of raw input statements (the network definition language and its parsing algorithms), and the network control process, which initiates and inhibits processes as they become needed or unnecessary.

(2) The states of the processes

Each process at any time can be in one of the following states:

(a) reset (not initiated)
(b) initiated
(c) successfully completed
(d) failed
(e) not needed

However there is no need to distinguish between states (d) and (e). Each process has three binary variables f, g and h to indicate its state.

f = 1 : the process was initiated.

h = 1 : the process failed or was found not needed.

The f variable is set externally to initiate a process. The g variable is set internally by the process to indicate its completion. The h variable may be set internally to indicate failure, or externally to indicate that this process is not needed.

The combinations of f, g, h may be interpreted as:

<table>
<thead>
<tr>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(3) The control language

Each process, p, has two propositions associated with it:

b(p) which is 'TRUE' if p has already started, and

c(p) which is 'TRUE' if p was successfully completed.

A network is defined by a set of statements \( \{ S_i \} \) of the form:

\[ S_i : f_i(c(p_1), c(p_2), c(p_3), ...) \Rightarrow b(p_h) \]

where the \( \{ f_i \} \) are propositional functions, which contain \( AND \) and \( OR \).

The meaning of \( S_i \) above is that when \( f_i \) is ‘TRUE’ then \( b(p_h) \) is set to ‘TRUE’ (which initiates the process \( p \), if it was not already initiated). For a more convenient notation, we write \( p_a \) for \( c(p_a) \) on the left hand side, and write \( p_b \) for \( b(p_b) \) on the right hand side.

Statements with identical propositional functions are combined by writing their propositional functions on the left, and writing the set of all their right hand sides, on the right. Example:

\[ 1 \Rightarrow \text{implies } y = 1 \]
\[ c(A) + c(B) & c(C) = b(D) \]

is written as:

\[ A + B & C = D, E \]

An example for definition of a network

\[ s \Rightarrow A, B \]

read: start with processes A and B (s for start)

\[ B & C \Rightarrow D, E \]

read: When B and C are successfully completed initiate D and E

\[ A + B \Rightarrow C, D \]

read: When A or B is successfully completed initiate C and D

\[ C & D + E \Rightarrow e \]

read: When (C and D) or E is successfully completed, end (e for end).

Let L be the left hand side of a statement (i.e., the propositional function) and let R be the set of the processes which are initiated when L is 'TRUE.' Hence each statement has the form:

\[ S_i : L_i \Rightarrow R_i \]

(read: \( L_i \) implies \( R_i \))

We say that p is used in \( L \Rightarrow R \), if \( L \) contains p.

Any process which is used by the network, but is not initiated by it, like s, is called a source-process, and its beginning is a source of the network.

Any process which is initiated by the network, but not used by it, like e, is called a sink-process, and its end is a sink of the network.

Each network must have at least one source. A circuit-free network has at least one source and one sink.

### (3.1) The control language processor

The input statements are compiled into a set of control instructions to be executed at run time. This compilation has two stages. The first stage checks validity (and outputs diagnostics) and reduces the input statements to some “graphable” form. The second stage produces control instructions according to the reduced statements.

(a) check validity (3.2.1)

(b) simplification (I), if possible (3.2.2)

(c) transform the network so that each process is initiated only once (3.2.3)

(d) separate AND from OR (3.2.4)

(e) transform the network so that each process is used only once (3.2.5)

(f) simplification (II), if possible (3.2.6)

Each simplification (I) step is applied as soon as it applicable.

This ordering of the steps insures that no reduction steps except simplifications are needed as a result of a later step.

After this procedure the input statements are reduced to a “graphable” form, where the definition statements correspond to vertices, and the processes correspond to edges. Because of the AND/OR separation (3.2.4) each node can be represented by an AND or OR gate.

During the reduction dummy processes are created, which can be recognized as such, thereafter. The role of these dummy processes is similar to the role of temporaries in program compilation. All dummy processes are created and named only by the compiler. We will name all dummy processes hereafter by q. Dummy processes have no duration, and do not need any time or resources to be executed. They serve to eliminate ambiguities in the network definition. The nature of these processes will be made clear in (3.2.3), (3.2.4) and (3.2.5) below.

Note that this reduction is not a complete optimization.

### (3.2) The reduction procedure

- **(3.2.1)** validity checking.
  - There must be at least one source to any network.

- **(3.2.2)** simplification (I)
  - The simplification is not essential, however it may save time and space and therefore is desirable.

  (a) is s is the only source of the network then replace s & p by p, and s + p by s, in any L. (Note that we drop subscripts for processes, for simplicity.)

  (b) If e is the only sink of the network, then if \{e,p\} \in R, delete p from R. If p is initiated only in this statement, replace c(p) by ‘false’ wherever p is used.

  (c) Apply any available method of propositional calculus simplification, to L. (i.e., replace A + 

\[ a \pm \text{ and } \& \text{ mean logical OR and AND. The AND takes precedence over the OR.} \]

\[ D, E \text{ means the set } \{D, E\}. \]

\[ \text{i.e., c(p) appears in the propositional function, L.} \]

\[ \text{The meaning of a “graphable” form is explained later, in (3.2)} \]
Parallel Process Definition and Control System

A by A, and A + A & B by A.)

(d) Delete L \Rightarrow \emptyset.
(e) Replace L \Rightarrow R_1 and L \Rightarrow R_2 by L \Rightarrow R_1 \lor R_2.
(f) Delete q \Rightarrow R (where q is a dummy-process), and replace q by R, where q was initiated. See Figure 1.
(g) Delete p =\Rightarrow q (where q is a dummy-process, and p a process) and replace q by p wherever q is used. See Figure 2.
(h) Delete statements of the form ‘FALSE’ \Rightarrow R. If some process p \in R is initiated only in this statement, replace c(p) by ‘false’ wherever p is used.

(3.2.3): Uniqueness of initiation

If any process is initiated more than once then there are L_1 \Rightarrow R_1 and L_2 \Rightarrow R_2 such that R_1 \land R_2 \neq \emptyset, i.e. there are common terms in the sets R_1 and R_2. In this case replace L_1 \Rightarrow R_1 and L_2 \Rightarrow R_2 by:

\[
\begin{align*}
L_0 &\Rightarrow q_1, R_1 - (R_1 \land R_2) \\
L_0 &\Rightarrow q_2, R - (R_1 \land R_2)
\end{align*}
\]

\[
\begin{align*}
\pi_1 &\Rightarrow q_1 \\
\pi_2 &\Rightarrow q_2 \\
\vdots \\
\pi_n &\Rightarrow q_n \\
\sum q_n &\Rightarrow R
\end{align*}
\]

where q_1 and q_2 are new dummy processes.

Example: L_1 \Rightarrow A, B, C and L_2 \Rightarrow B, C, D are replaced by:

\[
\begin{align*}
L_1 &\Rightarrow q_1, A \\
L_2 &\Rightarrow q_2, D \\
q_1 &\Rightarrow q_2, B, C
\end{align*}
\]

This rule is illustrated in Figure 3.

(3.2.4): OR/AND separation

Any statement which contains both \textit{AND} and \textit{OR} is converted to several statements which use only \textit{AND} or \textit{OR}. Some dummy processes are used to represent subcombinations of processes (the same way compilers use temporaries). A simple precedence reduction analysis suffices. For simplicity here, let us assume that L is converted to, or is given as a sum of products: L = \sum \pi_i. Sum of products may be simplified by using the idempotent and absorption rules. 7 In this case replace L \Rightarrow R by:

\[
\begin{align*}
\pi_1 &\Rightarrow q_1 \\
\pi_2 &\Rightarrow q_2 \\
\vdots \\
\pi_n &\Rightarrow q_n \\
\sum q_n &\Rightarrow R
\end{align*}
\]

7 The absorption rule: x + x & y = x and x & x = x

FIGURE 1—An example for rule (3.2.2.f)

is replaced by

\[
\begin{align*}
L_1 &\Rightarrow A \\
L_2 &\Rightarrow B \\
L_3 &\Rightarrow C \\
L_4 &\Rightarrow D
\end{align*}
\]

FIGURE 2—An example for rule (3.2.2.g)

is replaced by

\[
\begin{align*}
L_1 &\Rightarrow A \\
L_2 &\Rightarrow B \\
L_3 &\Rightarrow C \\
L_4 &\Rightarrow D
\end{align*}
\]

FIGURE 3—An example for rule (3.2.3)
where the \{q_1, \ldots, q_m\} are new dummy processes. Repeat this step as necessary, until all the statements are AND-statements or OR-statements.

The similarity between dummy processes and temporaries in program compilation may be illustrated by the following example. Consider the statement:

\[
A + B \cdot C \cdot (D + E) \Rightarrow F
\]

As an arithmetic statement it may be compiled as:

\[
\begin{align*}
D + E & \Rightarrow T_1 \\
B \cdot C \cdot T_1 & \Rightarrow T_2 \\
A + T_2 & \Rightarrow F
\end{align*}
\]

\[T_1 \text{ and } T_2 \text{ are temporaries}\]

As a process statement it may be compiled as:

\[
\begin{align*}
D + E & \Rightarrow q_1 \\
B \& C \& q_1 & \Rightarrow q_2 \\
A + q_2 & \Rightarrow F
\end{align*}
\]

\[q_1 \text{ and } q_2 \text{ are dummy processes}\]

See Figure 4.

\[(3.2.5): \text{ uniqueness of usage}\]

If some process, \(p\), is used \(m > 1\) times, then add \(p \Rightarrow \{q_1, q_2, \ldots, q_m\}\), and replace \(p\) in its \(n\)th usage by \(q_n\) (these \(\{q_n\}\) are new dummy processes). See Figure 5.

\[(3.2.6): \text{ simplification (II)}\]

Simplification (II) like simplification (I) is not essential, but may save time and space. It may be done only after (3.2.3) OR/AND separation, and (3.2.4) uniqueness of initiation, are completed.

---

**Figure 4**—An example for rule (3.2.4)

---

**Figure 5**—An example for rule (3.2.5)

(a) \(p + E_1 \Rightarrow q\) and \(q + E_2 \Rightarrow R\), where \(q\) is a dummy process is replaced by \(p + E_1 + E_2 \Rightarrow R\). See Figure 6.

(b) \(p \& E_1 \Rightarrow q\) and \(q \& E_2 \Rightarrow R\), where \(q\) is a dummy process is replaced by \(p \& E_1 \& E_2 \Rightarrow R\). See Figure 7.

\[(3.3): \text{ Example}\]

(a) The input statements:

\[(S_1) s \Rightarrow A, B\]
\[(S_2) A + B \Rightarrow C, D\]
\[(S_3) B \Rightarrow D, E\]
\[(S_4) C \& D + E \Rightarrow e\]

(b) \(D\) is initiated twice, in \(S_2\) and in \(S_3\). Apply (3.2.3):

\[(S_1) s \Rightarrow A, B\]
\[(S_2) A + B \Rightarrow C, q_1\]
\[(S_3) B \Rightarrow q_1, E\]
\[(S_4) C \& D + E \Rightarrow e\]
\[(S_5) q_1 + q_2 \Rightarrow D\]

(c) \(S_4\) contains both AND and OR. Apply (3.2.4):

\[(S_1) s \Rightarrow A, B\]
\[(S_2) A + B \Rightarrow C, q_1\]
\[(S_3) B \Rightarrow q_1, E\]
\[(S_4) q_1 + q_2 \Rightarrow e\]
\[(S_5) q_1 + q_2 \Rightarrow D\]
\[(S_6) C \& D \Rightarrow q_1\]
\[(S_7) E \Rightarrow q_4\]
(e) B is used twice, in $S_2$ and in $S_3$. Apply (3.2.5)

$$
(S_1) \quad s \Rightarrow A, B \\
(S_2) \quad A + q_0 \Rightarrow C, q_1 \\
(S_3) \quad q_5 \Rightarrow q_3, E \\
(S_4) \quad q_4 + E \Rightarrow e \\
(S_5) \quad q_1 + q_3 \Rightarrow D \\
(S_6) \quad C \& D \Rightarrow q_3 \\
(S_7) \quad B \Rightarrow q_9, p_5
$$

(f) step (f) of simplification I, can be applied to $S_3$, and $S_4$

$$
(S_1) \quad s \Rightarrow A, B \\
(S_2) \quad A + q_0 \Rightarrow C, q_1 \\
(S_3) \quad q_4 + q_9 \Rightarrow e \\
(S_4) \quad q_1 + q_9 \Rightarrow D \\
(S_5) \quad C \& D \Rightarrow q_3 \\
(S_6) \quad B \Rightarrow q_9, q_7, E
$$

No more reduction rules can be applied. The network is now in a “graphable” form, as shown in Figure 8.

(4) Compiling the input statements to control instructions

Now we assume that the input statements have been reduced to their “graphable” form, as described before. The next goal is getting instructions for updating the process variables $f, g$ and $h$. These instructions are executed at run time by some monitoring process. Let $\{\ell_i\}$ be the processes used in $L$, and $\{r_j\}$ the processes in $R$.

(4.1) A set of rules for compiling $\sum \ell_i \Rightarrow |r_j|$ can be written:

(4.1.1) If any $\ell_i$ in $L$ succeeds, initiate $R$ (success forward):

$$
\sum g(\ell_i) \rightarrow \{f(r_j)\}^* \\
^* \text{ means replacing (setting) e.g. } x \rightarrow y \text{ means replace } y \text{ by } x + y.
$$
Example (Figure 9): if A, or B, or C succeeds, initiate D and E.

(4.1.2) If all \( \ell_i \) in L fail, then R is not needed (failure forward):

\[
\prod g(\ell_i) \cdot h(\ell_i) \rightarrow \{ h(\tau_i) \}
\]

Example (Figure 9): if D and E, or C fail, so do D and E.

(4.1.3) If R is not needed, then L is not needed either (failure backward):

\[
\prod h(\tau_i) \rightarrow \{ h(\ell_i) \}
\]

Example (Figure 9): if D and E are not needed, neither are A, B and C.

(4.1.4) If any \( \ell_i \) in L succeeds, then its brothers are not needed (inhibit brothers)

\[
\sum g(\ell_i) \rightarrow \{ h(\ell_i) \}
\]

Example (Figure 9): if A succeeds, B and C are not needed. If B succeeds A and C are not needed, etc.

(4.2) A set of rules for compiling \( \prod \ell_i \rightarrow \{ \tau_i \} \) can be written:

(4.2.1) If all \( \ell_i \) in L succeed, initiate R (success forward):

\[
\prod g(\ell_i) \rightarrow \{ f(\tau_i) \}
\]

Example (Figure 10): if A and B succeed then initiate C, D, and E.

(4.2.2) If any \( \ell_i \) in L fails, then R is not needed (failure forward):

\[
\sum g(\ell_i) \cdot h(\ell_i) \rightarrow \{ h(\tau_i) \}
\]

Example (Figure 10): if A or B fails, so do C, D and E.

(4.2.3) If R is not needed, then L is not needed either (failure backward):

\[
\prod h(\tau_i) \rightarrow \{ h(\ell_i) \}
\]

Example (Figure 10): if C and D and E are not needed neither are A and B.

(4.2.4) If any \( \ell_i \) in L fails, then its brothers are not needed (inhibit brothers):

\[
\sum g(\ell_i) \cdot h(\ell_i) \rightarrow \{ h(\ell_i) \}
\]

Example (Figure 10): if A fails B is not needed. If B fails A is not needed.

The statement \( p \Rightarrow R \) can be compiled according to either rules. However we consider one input node as an AND node. Note that (4.2.4) is not necessary as it is implied by (4.2.2) and (4.2.3).

For each dummy process, \( q \), the setting instruction of \( f(q) \) is replaced by the setting instruction of \( g(q) \).

The purpose of the rules set forth in 4.1 and 4.2 is to inhibit the execution of processes which are not needed, and to initiate the execution of processes which are needed. A network with one sink only, is always inhibited upon arrival at it.

---

9 The overbar means logical complementation, e.g. \( \bar{0} = 1 \) and \( \bar{1} = 0 \).

10 Processes which terminate at a common vertex are called "brothers" here.
Example:

I. Consider the time-slice B–D–C, as marked on the graph in Figure 11. Say that D fails there at that time. Then (4.2.2) inhibits G and H, (4.2.3) or (4.2.4) inhibits E, (4.2.2) inhibits J, and (4.2.3) inhibits F and B. This means that process B, already initiated, is told to quit, since its success is not necessary to produce "e". This leaves only A, C and I not inhibited.

II. Consider process I completed. Process J is now deemed useless, and is turned off, as are its "parents", F and H.