INFORMATION AND CONTROL 40, 241–257 (1979)

Top-Down Design in the Context of Parallel Programs*

N. D. JOTWANI

Department of Mathematical and Computer Sciences, Michigan Technological University, Houghton, Michigan 49931

AND

J. ROBERT JUMP

Department of Electrical Engineering, Rice University, Houston, Texas 77001

A class of parallel programs, based on Free Choice Petri nets, is modeled by associating operators and predicates with vertices of the net. The model, called a *formal parallel program* (FPP), forms a natural extension of flow-chart notation to parallel programs. Definitions are made of the *behaviour* of an FPP, and the *simulation* of one FPP by another. A class of *top-down* FPPs is next defined, by requiring program graphs to be obtained through successive refinement steps, using a restricted set of control structures. Using the above definitions, it is shown that there exists an FPP $\mathscr{F}$ satisfying the property that for any top-down FPP $\mathscr{F}'$ simulating $\mathscr{F}$, the degree of parallelism attainable in $\mathscr{F}'$ is smaller than that in $\mathscr{F}$. The measure of parallelism used is the number of different ways of carrying out a computation. In the case of parallel programs, this phenomenon of *loss of parallelism* therefore uncovers a performance factor which may offset some of the advantages of using top-down design.

1. INTRODUCTION

A formal model of parallel programs is defined, using as the basis the class of Free Choice Petri nets (FCP nets), which is a proper sub-class of the class of Petri nets [Hack (1972)]. The model, called a *formal parallel program* (FPP), is obtained by associating operators and predicates with vertices of an FCP net. The model can be viewed as a natural extension of the familiar flow-chart notation to parallel programs. Using this model, some of the implications of using top-down design procedures for parallel programs are investigated.

The advantages of design by step-wise refinement, using a restricted set of control structures, are well-known. Program clarity, ease of modification,

* Work done under NSF Grant GJ-750.

0019-9958/79/030241–17$02.00/0

verification, etc. are some of the advantages of these design methods [Dijkstra, Mills]. In the case of parallel programs, especially, the complexity of unstructured programs rises enormously with program size unless the rules of step-wise refinement are followed. Also, top-down design for parallel programs ensures that the resulting programs are well-behaved—i.e. free from deadlocks [Coffman et al], and having well-defined loci of control, or 'live' and 'safe' in Petri net terms. Since the problem of liveness and safeness does not arise in sequential programs, this is an added advantage of using top-down design for parallel programs.

It will be shown in this paper than in some cases the use of top-down design for parallel programs may restrict the degree of parallelism attainable. The measure of parallelism used here is the number of different ways of carrying out a computation. In the case of parallel programs, therefore, a performance factor (program speed) will be shown to exist which may offset some of the advantages of top-down design.

The model used here may be compared with other proposed models of parallel computations [Rodriguez, Karp & Miller]. Note that the present model does not make an explicit representation of synchronization primitives [eg. Habermann]. In this connection, however, it may be noted that: (a) assigning the same operator (or predicate) to more than one vertex of the FCP net indicates a type of implicit synchronization, and (b) for some purposes of analysis, we do not require an explicit representation of the synchronization schemes.

Sections 2–4 below present the basic definitions related to the model. Section 5 describes formally the notion of top-down design, while section 6 develops the main result of this paper.

2. Definition of the Model

A. Mathematical Preliminaries

We present first the mathematical preliminaries needed for the definition of *formal parallel programs*.

A (finite) *Petri net* G is a triple $G = (T, P, E)$ where: T is a finite, nonempty set of *transitions*, P is a finite, non-empty set of *places*, and $E \subseteq (T \times P) \cup (P \times T)$ is the set of *edges* defining a directed, bi-partite graph. A *marking* M on G is a function $M: P \to \mathcal{N}$, where $\mathcal{N}$ is the set of non-negative integers. A *firing function* δ^t associated with a transition t is a function on the set of all markings on G. δ^t is defined at marking M iff $\forall p \in P, (p, t) \in E \Rightarrow M(p) > 0$, and is given as $\delta^t(M) = M'$ where

$$\forall p \in P,\ M'(P) = \begin{cases} M(P) + 1, & \text{if } (t, p) \in E \wedge (p, t) \notin E \\ M(P) - 1, & \text{if } (p, t) \in E \wedge (t, p) \notin E \\ M(P), & \text{otherwise} \end{cases}$$

We denote by δ the union of all the firing functions δ^t, $t \in T$, for a net G, and we say $M' = \delta(M)$ if $M' = \delta^t(M)$ for some transition t.

We say transition t is *firable* under marking M if δ^t is defined at M. A *firing sequence* σ of G is a string $\sigma = t_1 t_2 \cdots t_m$, $t_i \in T$ for $1 \leqslant i \leqslant m$, s.t. for a sequence $M^0, M^1, \ldots, M^m$ of markings on G, $M^i = \delta^{t_i}(M^{i-1})$ for $1 \leqslant i \leqslant m$. M^0 is the *initial marking* on G. We extend the notation to say $M^m = \delta(M^0, \sigma)$. Let $F(G)$ denote the set of all firing sequences of net G under initial marking M^0.

As an example, consider the net in Fig. 1. Here t_1, t_2, t_3 are transitions, and p_1, p_2 are places.

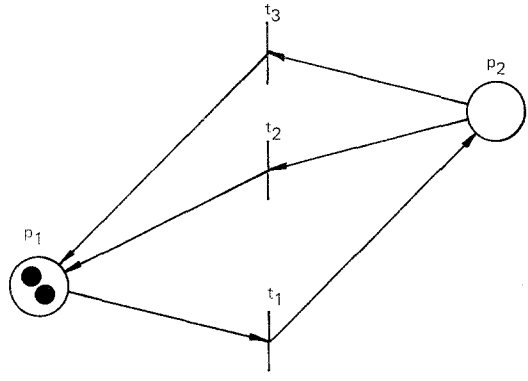

FIG. 1. Example of a Petri net.

Figure 1 indicates a marking M for which $M(p_1) = 2$ and $M(p_2) = 0$. In this case, we may also say that p_1 has two *tokens* (or *markers*) on it, while p_2 has none. Transition t_1 above is firable, and on *firing* t_1 one of the two tokens on p_1 will move to p_2.

A net $G = (T, P, E)$ is said to be *live* if, for any $t \in T$, and for any firing sequence σ of G, a sequence σ_t can be found s.t. t is firable under the marking $\delta(M^0, \sigma \cdot \sigma_t)$[1]. The net G is *safe* if, for any $p \in P$, and for any firing sequence σ of G, $M'(p) \leqslant 1$ where $M' = \delta(M^0, \sigma)$.

A *subnet* (or *subgraph*) $G' = (T', P', E')$ of G is a net satisfying (a) $T' \subseteq T$, (b) $P' \subseteq P$, and (c) E' is the restriction of E to T' and P'. We shall denote by $\pi(u, v)$ a path in G from vertex u to vertex v[1], and the terms cycle, initial end-point of π, terminal end-point of π, etc., will have the usual meaning. We shall denote by ${}^{\cdot}x$ and $x^{\cdot}$ respectively the sets $\{y \mid (y, x) \in E\}$ and $\{y \mid (x, y) \in E\}$, for any vertex x of G.

B. Free Choice Petri nets

A Petri net $G = (T, P, E)$ is *Free Coice* iff $\forall p \in P$, $| p^{\cdot} | > 1 \Rightarrow {}^{\cdot}(p^{\cdot}) = \{p\}$. The implication of this restriction is that whenever a place p is marked

[1] '$\cdot$' will denote concatenation of firing sequences of G, as well as that of paths in the net G.

(i.e. has one or more tokens) which has more than one output transition ($|\dot{p}| > 1$), then any one of these transitions may fire independently of the state of the other places of G. The net in Fig. 2 is Free choice, because p is the only place in it satisfying $|\dot{p}| > 1$, and $\dot{(}\dot{p}) = \{p\}$.

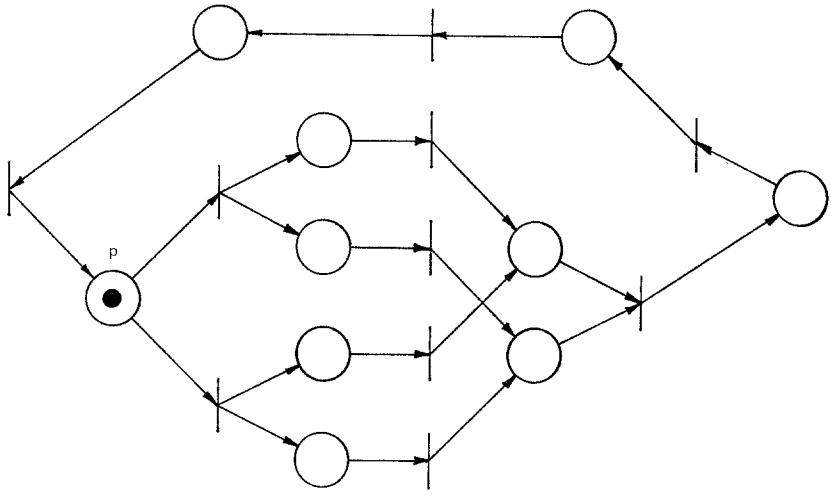

FIG. 2. Example of a Free Choice place.

We know that conditional branches in programs have the form:

$$\textit{if}\ (\text{condition} = \text{TRUE})\ \textit{then goto}\ \text{label}$$

The branch is taken independently of the state of the (parallel) program at other points, i.e. independently of the values of all other program counters. We claim that a natural model of parallel programs results if we restrict the decision nodes in it to have this *Free Choice* property. The two way branch is then shown in our model in Fig. 3.

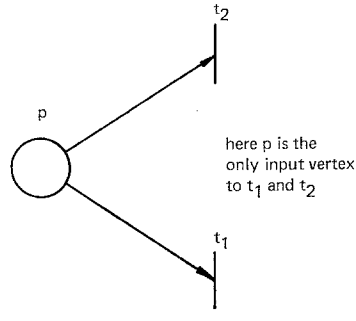

FIG. 3. Free Choice property.

We shall denote by FC(G) the set of all *Free Choice places* of FCP net $G = (T, P, E)$, i.e.

$$\mathrm{FC}(G) = \{p \in P \mid |p^{\cdot}| > 1\}.$$

The class of FCP nets was first analyzed in [Hack (1972)]. Hack determined necessary and sufficient conditions for the liveness and safeness of FCP nets. Essentially, he showed that is is possible to perform a pair of dual reductions on a live and safe FCP net, yielding strongly connected components of two basic types. A detailed analysis of any FCP net based model must necessarily use as starting points conditions similar to those in [Hack (1972)], but for the purposes of this paper a full statement of these conditions is not required.

The two basic types of components which result from the reductions mentioned above are state machines and marked graphs. A Petri net $G = (T, P, E)$ is a *state machine* if every $t \in T$ satisfies $|t^{\cdot}| \leqslant 1$ and $|{}^{\cdot}t| \leqslant 1$; and a Petri net $G = (T, P, E)$ is a *marked graph* if every $p \in P$ satisfies $|p^{\cdot}| \leqslant 1$ and $|{}^{\cdot}p| \leqslant 1$. The classes of state machines and marked graphs are obtained by excluding from FCP nets the features of parallel action and decision nodes respectively. A state machine corresponds to the familiar sequential finite state system. A marked graph represents, in a sense, the simplest possible parallel system—one in which there are no decision nodes. Marked graphs have been analyzed fully [Commoner et al], and have been employed to model asynchronous parallel control structures [Jump & Thiagarajan]. We shall use the two classes of state machines and marked graphs in order to formalize the notion of top-down design for our model of parallel programs.

C. Formal Parallel Programs

It can easily be shown, using the definitions of liveness and safeness, or using the criteria developed in [Hack (1972)], that a live and safe FCP must consist of one or more strongly-connected components. In order to base our parallel program model on FCP nets, we shall modify a live and safe FCP net and draw it in the form shown in Fig. 4. It is implicit in Fig. 4. that (a) t_0^0 is the only transition firable under the initial marking, and (b) addition of the 'return link' (Fig. 5) to the net yields a live and safe FCP net.

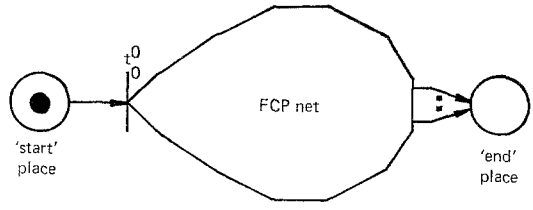

FIG. 4. Modified form of a live and safe FCP net.

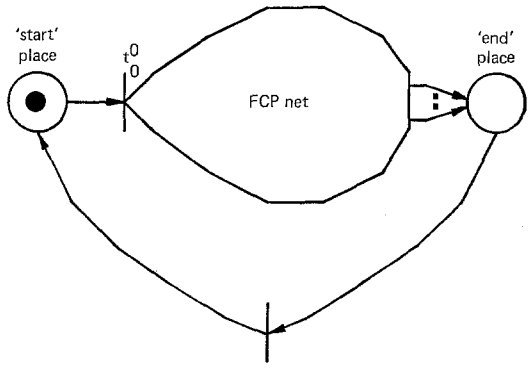

FIG. 5. 'Return link' restored in Fig. 4.

By this means we obtain from a live and safe FCP net a parallel program flow-graph with unique initial and terminal end-points ('start' and 'end' places above). The FCP net shown in Fig. 4 above will be given the name *linear FCP net* in the following paragraphs. Using the linear FCP net, we now formally define the parallel program model.

DEFINITION (Parallel program model)

A *formal parallel program* (FPP) is a 5-tuple $\mathscr{F} = (G, S_{\text{op}}, f_{\text{op}}, S_{\text{pr}}, f_{\text{pr}})$ where

(a) $G = (T, P, E)$ is a linear FCP net under initial marking M^0,

(b) S_{op} is a set of *operators* and f_{op} is a total function f_{op}: $T \to S_{\text{op}} \cup \{\lambda\}$, and λ is the *null-operator*, $\lambda \notin f_{\text{op}}[\text{FC}(G)^*]$,

(c) S_{pr} is a set of *predicates* and f_{pr} is a total function f_{pr}: $\text{FC}(G) \to S_{\text{pr}}$.

The 4-tuple $(S_{\text{op}}, f_{\text{op}}, S_{\text{pr}}, f_{\text{pr}})$ is an *interpretation* on G yielding $\mathscr{F}$. ∎

Note that the assignment of 'range' and 'domain' cells to vertices of G is not made here, since it is not central to the analysis of this paper. Also, the 'interpretation' used here is similar to that in [Keller], and therefore an FPP may be thought of as a realization of a parallel program schema. Moreover, an FPP is necessarily a finite-state realization of a compact, commutative schema.

Comparing our model with that of [Rodriguez], we see that we have a much smaller number of vertex types, and consequently simpler enabling rules. Also, the model of [Rodriguez] is of a 'data-flow' type, i.e. the function nodes receive data at the input edges and the results of the computation are placed on the output edges.

In the analysis which follows, we shall make use of a convention for naming the vertices of the linear FCP net G which will simplify the resulting notation. In the remaining paragraphs of this section we describe this convention.

A. Elements of S_{op} are named $f_0, f_1, \ldots, f_{|S_{op}|-1}$. Elements of T are then named according to the scheme below:

(a) $f_{op}^{-1}(\lambda) \subseteq T$ is given by $f_{op}^{-1}(\lambda) = \{\lambda^1, \lambda^2, \ldots, \lambda^{m_\lambda}\}$ where $m_\lambda = |f_{op}^{-1}(\lambda)|$,

(b) t_0^0 is the single transition at the output of 'start', and

(c) $f_{op}^{-1}(f_i) \subseteq T$ is given by $f_{op}^{-1}(f_i) = \{t_i^0, t_i^1, \ldots, t_i^{m_i}\}$, where $m_i = |f_{op}^{-1}(f_i)| - 1$, for $0 \leqslant i \leqslant |S_{op}| - 1$. We say there are $m_i + 1$ *occurrences* of the operator f_i (or, the ith operator) in $\mathcal{F}$, for $0 \leqslant i \leqslant |S_{op}| - 1$.

B. Elements of S_{pr} are named $g_0, g_1, \ldots, g_{|S_{op}|-1}$. Elements of FC($G$) are then named so that $f_{pr}^{-1}(g_i)$ is given by $f_{pr}^{-1}(g_i) = \{p_i^0, p_i^1, \ldots, p_i^{n_i}\}$, where $n_i = |f_{pr}^{-1}(g_i)| - 1$, for $0 \leqslant i \leqslant |S_{pr}| - 1$. We say there are $n_i + 1$ *occurrences* of the predicate g_i (or, the ith predicate) in $\mathcal{F}$, for $0 \leqslant g_i \leqslant |S_{pr}| - 1$.

The naming convention is chosen so as to provide us with an easy translation from an FPP to the corresponding FCP net, and vice versa. In the case that f_{op} and f_{pr} are one-to-one, we shall use f_i, g_j etc. as vertex names in G, and we shall refer to a transition as an operator, or to an FC place as a predicate.

The definition of the parallel program model is now complete. In the next two sections we present two formal definitions—the *behaviour* of an FPP, and the *simulation* of one FPP by another—which are needed in establishing the main result of the paper.

3. Behaviour of a Parallel Program

The behaviour of an FPP, the parallel program model of the previous section, will now be defined along the lines of the 'computation sequences' of other models [Karp & Miller, Rodriguez]. Specifically, the set of all possible sequences of operators and predicates that may be observed will define the bahaviour of an FPP. Along similar lines are the definitions of [Hack (1976)] and [Peterson].

Let $\mathcal{F} = (G, S_{op}, f_{op}, S_{pr}, f_{pr})$ be an FPP. Let α be any firing sequence of G. The *behaviour sequence* α' of $\mathcal{F}$ corresponding to α is obtained from α by means of the following sequence of steps:

(a) delete any instances of λ^i, $1 \leqslant i \leqslant m_\lambda$, from α,

(b) insert p_j^i immediately to the left of any instances of an $x \in (p_j^i)^\cdot$ in the resulting string, where $0 \leqslant i \leqslant n_j$ and $0 \leqslant j \leqslant |S_{pr}| - 1$,

(c) replace any instances of t_j^i in the resulting string by the operator f_j, where $0 \leqslant i \leqslant m_j$ and $0 \leqslant j \leqslant |S_{op}| - 1$,

(d) replace any instances of p_j^i in the resulting string by the predicate g_j, where $0 \leqslant i \leqslant n_j$ and $0 \leqslant j \leqslant |S_{pr}| - 1$.

Example. Consider the FPP shown in Fig. 6. Two of the firing sequences of this net are $t_0^0 t_0^1 t_2^0$ and $t_0^0 t_1^0 t_2^0$. The corresponding two behaviour sequences

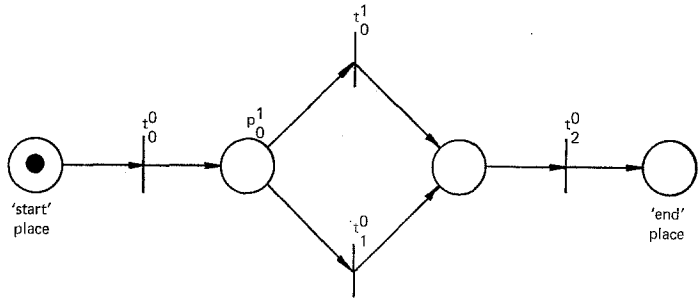

FIG. 6. Example of a Formal Parallel Program.

are $f_0g_0f_0f_2$ and $f_0g_0f_1f_2$. We see that a behaviour sequence preserves the information about the decisions made at FC places, as well as the order in which the various operations and decisions were carried out. ∎

We denote by $\mathscr{B}(\mathscr{F})$ the set of all behaviour sequences of $\mathscr{F}$. We then denote by β the onto function[2] $\beta: F(G) \to \mathscr{B}(\mathscr{F})$ which is defined by steps i–iv of the above definition.

DEFINITION. (Behaviour of an FPP)
The *behaviour* of an FPP $\mathscr{F}$ is the set $\mathscr{B}(\mathscr{F})$ of all the bahaviour sequences of $\mathscr{F}$. ∎

Based on the above definition of the bahaviour of an FPP, in the following Section we define the simulation of an FPP $\mathscr{F}$ by another FPP $\mathscr{F}'$.

4. SIMULATION

Using the above definition of the bahaviour of an FPP, we shall formalize the idea of simulation between two FPP's.

Recall that the presence of parallelism, in general, permits a computation to be carried out in more than one way. The following definition states the conditions under which two behaviour sequences of an FPP represent the same computation.

Let α and α' be any two behaviour sequences of an FPP $\mathscr{F} = (G, S_{\text{op}}, f_{\text{op}}, S_{\text{pr}}, f_{\text{pr}})$. α and α' are said to be *similar* if the following conditions are satisfied:

(a) $\#(f_i \mid \alpha) = \#(f_i \mid \alpha'), \forall f_i \in S_{\text{op}}$[3]
$\#(g_i \mid \alpha) = \#(g_i \mid \alpha'), \forall g_i \in S_{\text{pr}}$

[2] Recall that $F(G)$ is the set of all firing sequences of G.
[3] Here $\#(x \mid \alpha)$ denotes the number of occurrences of x in the string α.

(b) if the kth occurrences of any $g_i \in S_{\text{pr}}$ are followed in α and α' by f_j and $f_{j'}$ respectively, then $j = j'$; here $g_i \in S_{\text{pr}}$, f_j, $f_{j'} \in S_{\text{op}}$ and $1 \leqslant k \leqslant \#(g_i \mid \alpha) = \#(g_i \mid \alpha')$.

The second part of the definition above states that the pattern of decisions made to obtain the two sequences α and α' is identical. We say $(\alpha, \alpha') \in sim$ iff α and α' are similar. Clearly sim is then an equivalence relation on $\mathscr{B}(\mathscr{F})$.

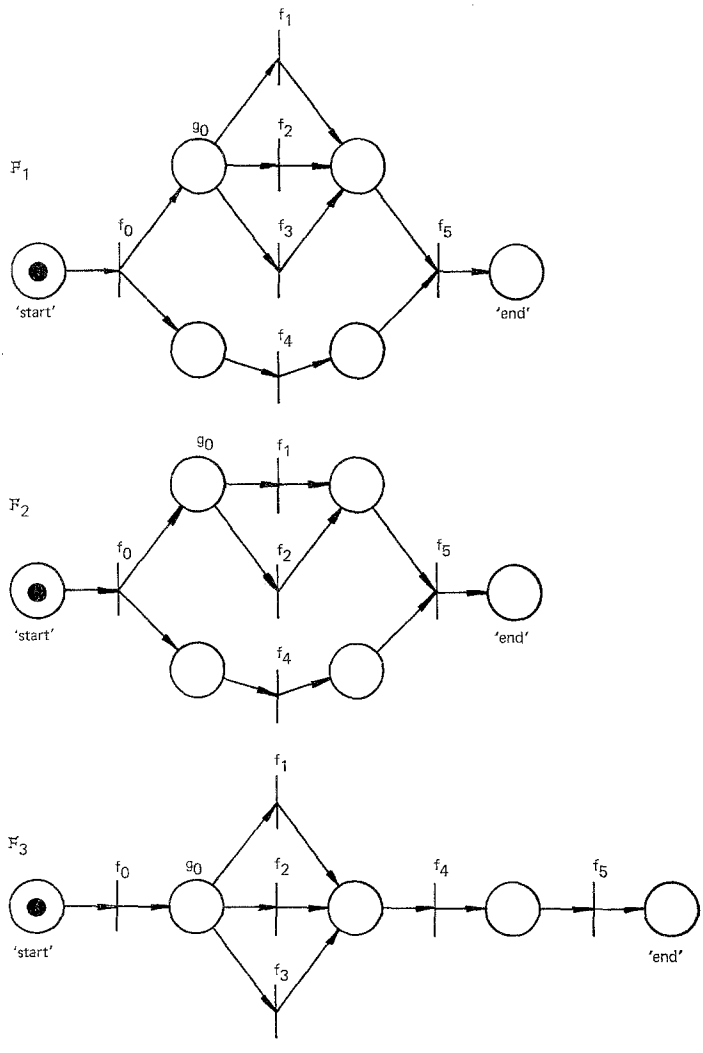

FIG. 7. Examples of simulation of FPPs.

DEFINITION. (Simulation of one FPP by another)

Let $\mathscr{F}$ and $\mathscr{F}'$ be two FPP's. We say $\mathscr{F}'$ simulates $\mathscr{F}$ iff the following conditions are satisfied:

(i) $\mathscr{B}(\mathscr{F}') \subseteq \mathscr{B}(\mathscr{F})$,

(ii) no equivalence class of the relation *sim* on $\mathscr{B}(\mathscr{F})$ is disjoint with $\mathscr{B}(\mathscr{F}')$.

We say that $\mathscr{F}$ and $\mathscr{F}'$ are *equivalent*, or $\mathscr{F}'$ simulates $\mathscr{F}$ *without loss of parallelism*, if $\mathscr{B}(\mathscr{F}) = \mathscr{B}(\mathscr{F}')$. ▮

We show an example illustrating the above definitions.

Consider the three FPP's shown in Fig. 7. It can easily be verified that $\mathscr{F}_3$ simulates $\mathscr{F}_1$ but is not equivalent to it, because f_4 has been placed in sequence with g_0, f_1, f_2, f_3. $\mathscr{F}_2$ does not simulate $\mathscr{F}_1$ because part ii of above definition is violated—there is no behaviour sequence in $\mathscr{F}_2$ similar to any of the sequences $f_0g_0f_3$, $f_0f_4g_0f_3$, $f_0g_0f_3f_4$, $f_0g_0f_3f_4f_5$ and $f_0f_4g_0f_3f_5$ of $\mathscr{F}_1$.

This completes the definitions related to our model which are essential for the analysis in the latter sections.

5. TOP-DOWN DESIGN

The main aim of this paper is to describe one implication of using top-down design techniques, i.e. design by successive refinement steps, for parallel programs. We shall prove in section 6 that under certain conditions an FPP has no top-down equivalent, even though it can be shown that any FPP can be simulated by a top-down FPP [Jotwani].

In this section we shall formally define the class of top-down FPP's, and we shall briefly outline the relationship between this class and the classes of interval-reducible [Hecht & Ullman] and structured [Mills] sequential programs.

The following few preliminary definitions will lead up to the central definition of this section:

A *proper state machine* (proper marked graph, resp.) is a linear FCP net which is a state machine (marked graph, resp.) from which the place 'start' has been deleted, and to which a terminal transition has been added as diagrammed in Fig. 8. The *substitution* of net G' *into* a linear FCP net G is defined iff G' is either a proper state machine (psm) or a proper marked graph (pmg). The substitution *at transition* t of G consists in replacing t in G by G', as shown below,

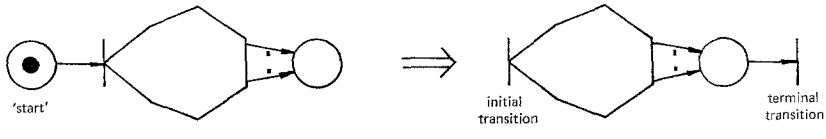

FIG. 8. Obtaining 'proper state machines' and 'proper marked graphs.'

to yield another linear FCP net G'' (Fig. 9). We use the terms S-substitution and M-substitution respectively to denote that the net G' is a psm or a pmg in a particular instance of substitution.

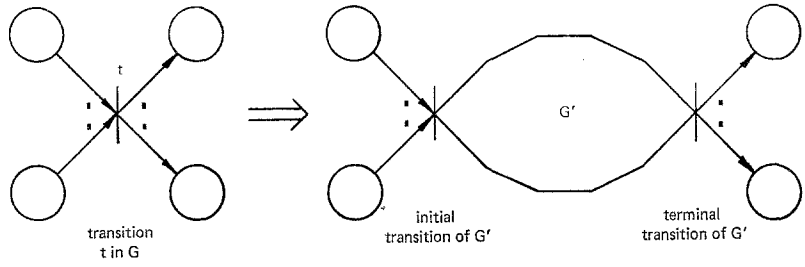

FIG. 9. Substitution.

DEFINITION. (Top-down programs)

The class of *top-down* FCP nets, strictly contained in the class of linear FCP nets, is defined inductively as follows:

basis step—the net G_0^0 shown in Fig. 10 is a top-down FCP net.

induction step—if linear FCP net G is a top-down net, and if net G'' is obtained from G by means of a single substitution step, then G'' is a top-down FCP net.

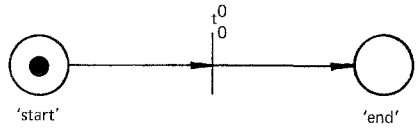

FIG. 10. Basis top–down FCP net.

An FPP $\mathscr{F} = (G, S_{op}, f_{op}, S_{pr}, f_{pr})$ is a *top-down program* (TDP) if G is a top-down FCP net. ▮

Essentially the definition of TDP's states that modules representing parallelism and those representing control-flow logic (i.e. pmg and psm nets respectively) should be introduced separately into the program, one at a time. The definition is a very natural one in this context, since it requires that the two orthogonal features of a parallel program, parallelism and control flow branches, be introduced separately into a top-down program.

Note that we have made no restrictions on the type of proper state machine used in order to obtain a top-down program. Two restricted classes of sequential programs very widely used are the class of interval--reducible programs [Hecht & Ullman], and the class of structured programs [Mills]. In the following section we show that under certain conditions the use of top-down parallel programs may involve a loss in the degree of parallelism attainable. Now it is known that

any sequential program has an interval-reducible equivalent. Also, under a definition of program equivalence which is based on the functional behaviour of programs (as opposed to a definition based on behaviour sequences) it can be shown that any sequential program has a structured equivalent [Mills]. In view of these known properties of sequential programs, it can be shown that restricting the psm nets (in the above definition of top-down FPP's) to be interval-reducible, or structured, does not affect the main result of this paper (in section 6 below) concerning loss of parallelism.

A more complete discussion of the implications of using interval-reducible and structured state machines can be found in [Jotwani]. Also in it is a formal proof of the intuitive result that any FPP can be simulated by a TDP, which may be thought of as a Structure Theorem for this class of parallel programs.

6. Loss of Parallelism

Making use of the above definitions of FPP's, behaviour, TDP's, etc., we show in this section that there exist FPP's for which there are no TDP equivalents. As mentioned in section 5 above, it can be formally proved, by making use of an appropriate 'structure algorithm', that any FPP can be *simulated* by a TDP. Therefore the result of this section, in effect, is that the use of top-down design procedures for parallel programs may entail a loss in the degree of parallelism attainable. Note that in view of our definitions of 'simulates' and 'equivalence', the measure of parallelism used here is the number of different behaviour sequences of an FPP $\mathscr{F}$, i.e. the number of different ways in which the parallel computation represented by $\mathscr{F}$ may be carried out.

In Fig. 11 below we show the FPP $\mathscr{F}$ for which we shall prove that there is no top-down equivalent. The relevant proofs follow, in Lemmas 6.1–6.3, leading up to Theorem 6.1.

Therefore let $\mathscr{F} = (G, S_{\text{op}}, f_{\text{op}}, S_{\text{pr}}, f_{\text{pr}})$ be the FPP shown in Fig. 11. Note that here f_{op} and f_{pr} are one-to-one.

There is exactly one FC place in G, which corresponds to the predicate g above. The two output transitions of this FC place correspond to operators f_1 and f_2.

Now let $\mathscr{F}' = (G', S_{\text{op}}, f'_{\text{op}}, S_{\text{pr}}, f'_{\text{pr}})$ be any TDP simulating $\mathscr{F}$.

Lemma 6.1. *With $\mathscr{F}'$ as defined above, the net G' contains exactly one* FC *place, which corresponds to predicate g.*

Proof. That G' contains an FC place p corresponding to the predicate g (i.e. $f'_{\text{pr}}(p) = g$) follows at once when we consider the bahaviour sequence $f_0 g f_1$ of $\mathscr{F}$, which must be observed in $\mathscr{F}'$. To see that p is the only FC place in G', assume the opposite, i.e. let $p' \in \text{FC}(G')$ s.t. $p' \neq p$ and $f'_{\text{pr}}(p') = g'$. But then

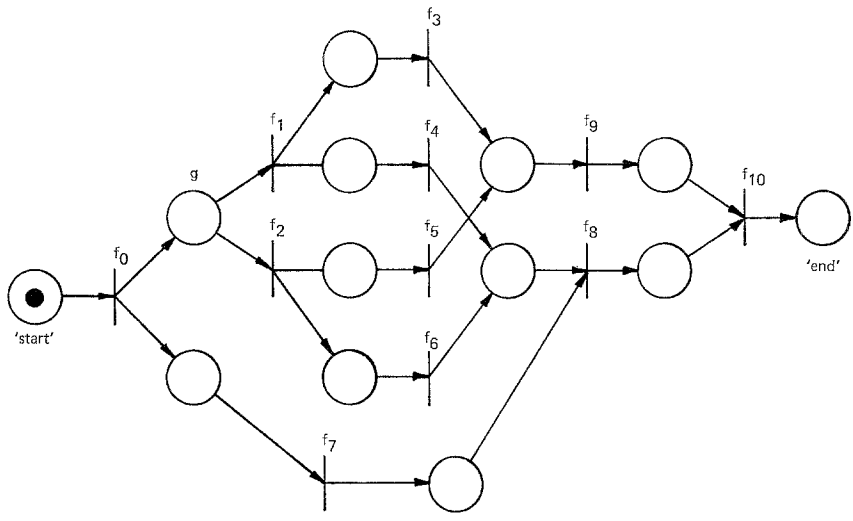

FIG. 11. FPP to which there is no TDP equivalent.

from basic properties of live and safe FCP nets we can easily show that a bahaviour sequence α of the type $\alpha = \cdot\cdot\, g^{(1)} \cdots g^{(2)} \cdot\cdot$ is observed in $\mathscr{F}'$, i.e. α has two occurrences of predicates in it. Since it can be verified for $\mathscr{F}$ that no behaviour sequence of the type of α is observed in $\mathscr{F}$, and since we have assumed that $\mathscr{F}'$ simulates $\mathscr{F}$, we reach a contradiction. The result follows at once. ∎

LEMMA 6.2. *There is exactly one transition in G' corresponding to each of the operators f_1, f_2, and f_3, i.e.* $|f'^{-1}_{\mathrm{pr}}(f_1)| = |f'^{-1}_{\mathrm{pr}}(f_2)| = |f'^{-1}_{\mathrm{pr}}(f_3)| = 1$.

Proof. Follows from an argument similar to that of Lemma 6.1, i.e. by showing that the opposite would imply a behaviour sequence α of $\mathscr{F}'$ which is not observed in $\mathscr{F}$. ∎

LEMMA 6.3. *At least one of the following two statements must be true in $\mathscr{F}'$:*

A: the operators f_1 and f_7 cannot be enabled in parallel in $\mathscr{F}'$, or

B: the operators f_3 and f_8 cannot be enabled in parallel in $\mathscr{F}'$.

Proof. Assume that A is false, i.e. f_1 and f_7 can be enabled in parallel in $\mathscr{F}'$. In view of the fact that there is only one occurrence in $\mathscr{F}'$ of g and f_1, the paths shown in Fig. 12 then exist in G', for some transition t.[4] Further, since G' is a top-down net and g is an FC place, the configuration shown in Fig. 13 exists in G', for some transition t'. Here I and II are the two alternate regions corre-

[4] In all the diagrams of this section, a straight line between two vertices represents an edge, while a 'curved' line represents a path of length $\geqslant 1$.

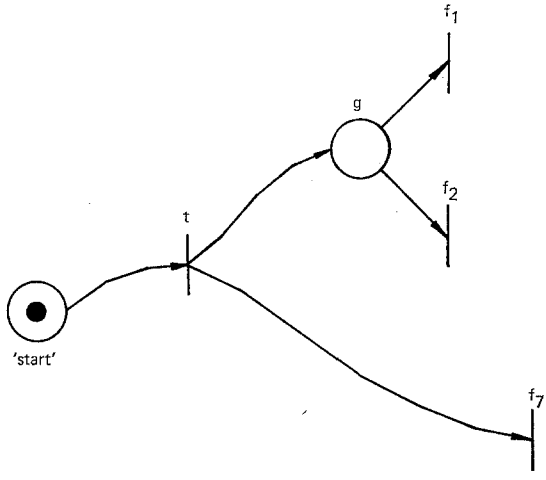

FIG. 12. Diagram 1 for Lemma 6.3.

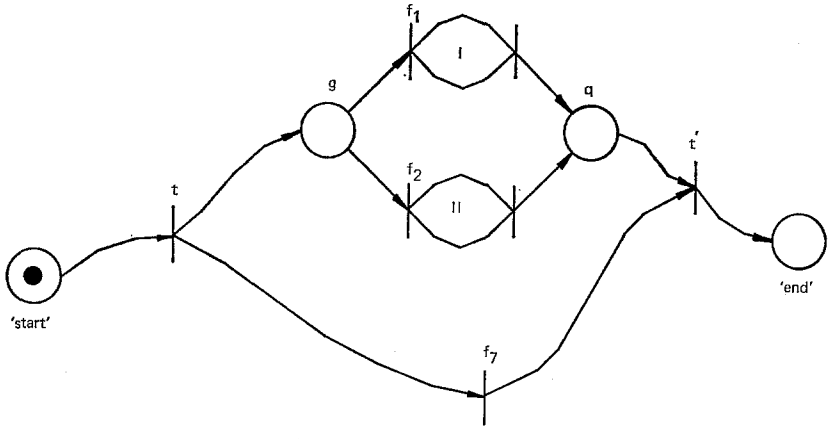

FIG. 13. Diagram 2 for Lemma 6.3.

sponding to the 2-way branch at g. Note that regions marked I and II in the above figure are disjoint, since the FC place g must be introduced in an S-substitution, and we can always choose q (in the corresponding psm) s.t. I and II are disjoint.

Now note the following arguments:

(i) f_3 is in region I above, since it must be reached only if the decision made at g is f_1, and

(ii) f_8, f_{10} are *not* in region I since they are constrained to fire only after f_7 fires.

[Both (i) and (ii) must hold in G' in view of the corresponding properties in $\mathscr{F}$.]

Now note the following arguments:

(a) since f_3 is constrained to fire only after f_1 fires, there is a blank elementary path π_1 in region I from f_1 to f_3 ; similarly there is a blank elementary path π_2 in G' from f_1 to f_8 , and a blank elementary path π_3 from f_3 to f_{10} , and

(b) both π_2 and π_3 given by (a) above contain the place q, in view of the argument (ii) above.

Now, from (a) and (b), it follows at once that the configuration in Fig. 14 exists in G', for some transitions t'', t'''. But then clearly a blank elementary path can be constructed in G' from f_3 to f_8 , yielding statement B of the Lemma. The proof is complete. ▮

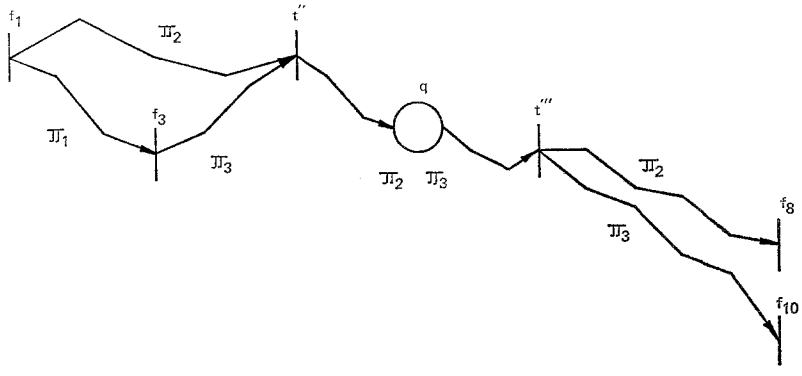

FIG. 14. Diagram 3 for Lemma 6.3.

We are now ready to present formally the main result of this section:

THEOREM 6.1. *The* FPP $\mathscr{F}$ *described above has no* TDP *equivalent.*

Proof. Note that in $\mathscr{F}$, the operators f_1 and f_7 can be enabled in parallel, yielding a behaviour sequence of the type $\cdots f_7 g f_1 \cdots$ and a behaviour sequence of the type $\cdots g f_1 f_7 \cdots$. Similarly, in $\mathscr{F}$ the operators f_3 and f_8 can be enabled in parallel, yielding behaviour sequences of the types $\cdots f_3 f_8 \cdots$ and $\cdots f_8 f_3 \cdots$. However, in view of Lemma 6.3, any TDP $\mathscr{F}'$ simulating $\mathscr{F}$ will not exhibit behaviour sequences of each of the four types above, i.e. $\mathscr{B}(\mathscr{F}') \neq \mathscr{B}(\mathscr{F})$, and the result follows. ▮

The implication of this theorem, of course, is that the computation represented by $\mathscr{F}$, with its implied data-dependencies, cannot be realized using top-down design without sacrificing some of the parallelism attainable in the 'unstructured' version represented by $\mathscr{F}$. $\mathscr{F}$ above illustrates a typical configuration of parallel operations for which, in this sense, there is no top-down equivalent.

This phenomenon of loss of parallelism has been more fully analyzed in [Jotwani], where conditions have been determined which are necessary and sufficient for an FPP to a have a TDP equivalent.

7. Conclusions

We have demonstrated, through the example program of Fig. 11, that in some cases design by top-down refinement may restrict the degree of parallelism attainable in a parallel program. For parallel programs, a performance factor has thus been shown to exist which may offset some of the advantages of using top-down design techniques. For a specific problem, a special-purpose parallel module may be designed which is not any of the basic modules defined in section 5, in order to increase the degree of parallelism attainable. This module (to which there would be no top-down equivalent) may then be incorporated into the top-down design procedures. This would require additional design effort, and clearly the relative weight attached to program speed in the overall performance will be the deciding factor.

Received: April 14, 1978; revised: August 11, 1978

References

Coffman, E. G., *et al.* (1971), System deadlocks, *Comput. Surveys* **3**, 67–78.

Commoner, F., *et al.* (1971), Marked directed graphs, *J. Comput. System Sci.* **5**, 511–523.

Dijkstra, E. W. (1972), Notes on structured programming, *in* "Structured Programming" (E. W. Dijkstra, *et al.*, Eds.), pp. 1–82, Academic Press, New York.

Habermann, A. N. (1972), Synchronization of communicating processes, *Comm. ACM* **15**, 171–176.

Hack, M. H. T. (1972), "Analysis of Production Schemata by Petri Nets," Rept. No. MAC-TR-94, Laboratory for Computer Science, M.I.T., Cambridge, Mass.

Hack, M. H. T. (1976), "Petri Net Languages," Rept. No. MAC-TR-159, Laboratory for Computer Science, M.I.T., Cambridge, Mass.

Hecht, M. S., and Ullman, J. D. (1972), Flow graph reducibility, *SIAM J. Computing* **1**, 188–202.

Jotwani, N. D. (1977), "Study of a Class of Parallel Programs," Ph. D. Thesis, Department of Electrical Engineering, Rice University, Houston, Tex.

Jump, J. R., and Thiagarajan, P. S. (1973), On the equivalence of asynchronous control structures, *SIAM J. Computing* **2**, 67–87.

Jump, J. R., and Thiagarajan, P. S. (1975), On the interconnection of asynchronous control structures, *J. ACM* **22**, 596–612.

Karp, R. M., and Miller, R. E. (1966), Properties of a model for parallel computation: Determinacy, termination, queueing, *SIAM J. App. Math.* **14**, 1390–1411.

Karp, R. M., and Miller, R. E. (1969), Parallel program schemata, *J. Comput. System Sci.* **3**, 147–195.

KELLER, R. M. (1973), Parallel program schemata and maximal parallelism, *J. ACM* **20**, 514–537; 696–710.

MILLS, H. D. (1972), "Mathematical Foundations of Structured Programming," IBM, Gaithersberg, Md.

PETERSON, J. L. (1976), Computation sequence sets, *J. Comput. System Sci.* **13**, 1–24.

RODRIGUEZ, J. E. (1969), "A Graph Model for Parallel Computation," Rept. No. MAC-TR-64, Laboratory for Computer Science, M.I.T., Cambridge, Mass.