A Methodology for Transforming Sequential Flowcharts into Parallel Ones Using FIFO Nets

Annie Choquet-Geniet, L.I.S.I., E.N.S.M.A., 86960 Futuroscope Cedex, France
Tel 33 49 49 80 68 - Fax 33 49 49 80 64 - E Mail ageniet@diane.univ-poitiers.fr

Dominique Geniet, L.I.S.I., E.N.S.M.A., 86960 Futuroscope Cedex, France
Tel 33 49 49 80 62 - Fax 33 49 49 80 64 - E Mail dgeniet@diane.univ-poitiers.fr

René Schott, C.R.I.N., Université de Nancy 1, 54506 Vandœuvre les Nancy Cedex, France
Tel 33 83 59 30 41 - Fax 33 83 27 83 19 - E Mail schott@loria.fr

0 Abstract

We present a methodology based on Fifo nets, for the transformation of flowcharts modeling sequential programs into flowcharts integrating all semantically equivalent behaviours, both sequential and parallel. The model of fifo nets is appropriate since fifo nets model the test to zero problem (required for the modeling of loops of not predined size), and integrate scheduling (required since some actions must occur in a given order). We describe the complete transformation, and show that the fifo nets got provides a concise representation of the set of the behaviours semantically equivalent to the behaviours of the sequential flowchart.

Proc. of 15th International Conference on Computing and Information
Petersborough, Ontario, Canada, 7/1995
pp 177-190

1 Introduction

The design of a parallel program can be got in two ways: either one specifies directly the parallelism, using a parallel language; or one first writes a sequential program, and then transforms it into a parallel one, with the help of appropriate tools. Since the design of a sequential program is much easier as the design of a parallel one, the second approach appears as more attractive, and efficient tools are still needed.

Our investigation field corresponds to this approach. We present a methodology for transforming the flowchart modeling a sequential program into a flowchart which collects all its semantically equivalent behaviours (sequential as well as parallel). This approach is based on Fifo nets.

Others approaches have been developed, e.g. [C91][C92][CP93][F92]: in all of them, the first step consists of the construction of a semantic graph of precedence of the tasks; in addition, [C91] uses scheduling tools; in [F92], the graph is used in order to transform the control structure of the program, the parallelisation is then performed on the transformed program. [C92] and [CP93] uses the trace monoid. One must notice that the programs concerned by these papers are composed of basic statements and *for* loops (i.e. of pre-defined size). Finally, the goal of these approachs is the production of one particular parallel version of the initial sequential program.

We are interested in an exhaustive modeling of all the behaviours generated by a sequential program, i.e. all the behaviours which are semantically equivalent.

Several tools for modeling parallel programs have been developed. Among them, let us mention synchronised automata [AN82], Petri nets [P62], and Fifo nets [M83].

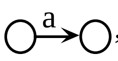
Our aim is to provide a tool for transforming the sequential flowchart of a program into a parallel one, for programs composed of elementary statements and *while* loops (the number of iterations is not pre-defined). This model must collect both sequential and parallel behaviours of the program. Once built, it can be studied *off-line* (for example, the marking graph can be built) for the generation of code; or *on-line*, for driving the software. We have chosen the model of fifo nets, for they integrate the notion of scheduling (respect of the order) and the test to zero. If we use an operational semantics which allows one firing at once, the model generates the set of sequential behaviours of the program. A vectorial operational semantics gives the set of parallel behaviours. This approach goes back to [R86]. We extend and precise it.

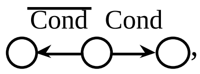
This paper is organized as follows: in section 2, we present the modeling of sequential programs, and we defined the set of semi-equivalent programs, with the help of a conflict relation; in section 3 two examples are proposed; in section 4 we present the model of fifo net, and describe the construction of the fifo net modeling the set of semi-equivalent behaviours. Finally, section 5 gives the main properties of the model.

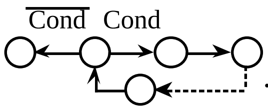
2. Sequential programs

1. Modeling

A sequential program can be classically represented by the use of a finite automaton [E76], called *sequential flowchart*. Each transition of the automaton is labelled by an atomic statement.

An imperative statement **a** is represented by the scheme 

, *If Cond Then…Else…* by 

, and a loop by 

.

Example

```
Get (x) ;
Get (y) ;
If x>y
Then
     While x>y
     Loop
          x:=x-1 ;
     End Loop ;
Else
     Put (x+y) ;
```

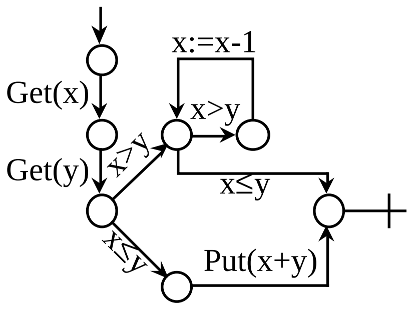


Figure 1: Sequential program

Notations

We denoteby ξ the set of test statements of a program, and by I the set of imperative statements. Each test $\mathbf{p} \in \xi$ generates two transitions, labelled by p_1 and p_2, corresponding respectively to p_{True} and p_{False}. Let ξ_1 and ξ_2 be the sets generated in this way. We define two mappings h_i:$\xi \rightarrow \xi_i$,$p \mapsto p_i$. The alphabet of the flowchart is $\Sigma = I \cup \xi_1 \cup \xi_2$. We denote the fact that ***a*** *is an internal statement of loop B* by $\mathbf{a} \in$ B (the test **p** of B is considered as an internal statement too).

2. The conflict relation

We are interested in whether the temporal order of two statements in the execution of the algorithm is essential for the correctness of the program. If not, the statements can be performed in parallel, or their order can be commuted. We express this by relations on the statements of the program.

The dependence relation *D* (Bernstein relations)

Within a program, some actions act on common variables and may not therefore occur simultaneously, or be permuted. Meanwhile, some others may, since they act on separate variables. This is represented by the dependency relation (which is symmetric) defined on $(\xi \cup I) \times (\xi \cup I)$.

Let **s** be a statement, W_s (resp. R_s) the set of variables of the program modified by **s** (resp. read by **s**). **s** and **t** are independent if and only if **s** neither reads or writes on any variable modified by **t**, and conversely.

Definition 1 : $\mathbf{s}\ D\ \mathbf{t} \Leftrightarrow \big(W_{\mathbf{s}} \cap (W_{\mathbf{t}} \cup R_{\mathbf{t}}) \big) \cup \big(W_{\mathbf{t}} \cap (W_{\mathbf{s}} \cup R_{\mathbf{s}}) \big) \neq \emptyset$

The dependency graph represents the dependency relation combined with the effective order of occurrence of conflicting actions : $\mathbf{a} \rightarrow \mathbf{b}$ means that **a** and **b** are conflicting and **a** occurs before **b** in the sequential program. The dependency relation is the symmetric closure of the non symmetric relation represented by the graph.

Example

```
(a)   Get ( x ) ;
(b)   Get ( y ) ;
(p)   While x > 0
      Loop
(c)        x : = x - 1 ;
(d)        y : = y + 1 ;
      End Loop ;
(e)   Put ( y ) ;
```

For this program, the flochart is given in figure 2a. The dependency relation D is {(**a**, **p**), (**p**,**a**), (**a**, **c**), (**c**, **a**), (**b**, **d**), (**d**, **b**), (**b**, **e**), (**e**, **b**), (**c**, **p**), (**p**, **c**), (**d**, **e**), (**e**, **d**)}. The associated dependency graph is given in Figure 2b.

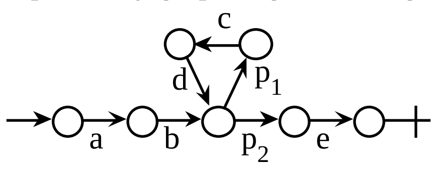


Figure 2a
Sequential flowchart

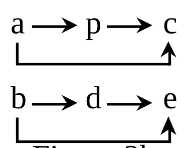


Figure 2b
Dependency graph

Figure 2: Dependency graph

This relation, defined on $\xi \times (\xi \cup I)$, is not symmetric. Let **p** be the test statement of a loop P (we say that P is *governed* **p**), and **a** an internal statement of P. The ith occurrence of **a** cannot be performed before the ith occurrence of **p** has been performed (even if they are independent in the sense of D) ; however, if there is no dependency between **a** and **p** in the sense of D, **a** can be delayed after the jth occurrence of the test **p**, if j ≥ i :

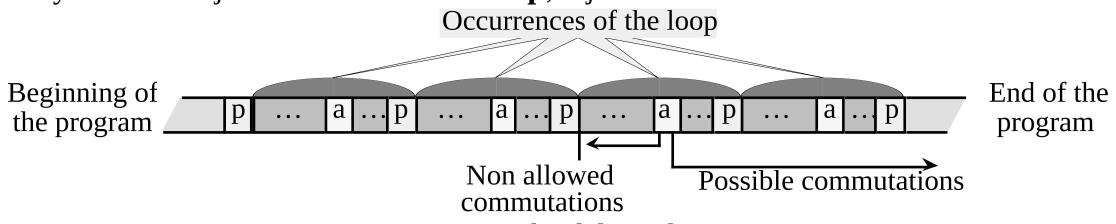


Figure 3: The delay relation

Definition 2 : Let P be a loop governed by **p** and **a** a statement : $\mathbf{p}\ S\ \mathbf{a} \Leftrightarrow \mathbf{a} \in P$

Remark

Since $\mathbf{p} \in P$, we have $\mathbf{p}\ S\ \mathbf{p}$. This corresponds to the fact that every positive occurrences of the test must occur before the negative one.

Example

Again with the program of Figure 2, we have S = {(**p**, **c**), (**p**, **d**), (**p**, **p**)}. Here, **d** can be delayed after the next occurrences of the test **p**.

The precedency relation *P*

This relation can be viewed as an extension of the dependency relation, where loops are seen as meta-actions. Such a point of view enables a modular representation of the program (and thus a modular processing of the flowchart).

1. Let P and Q be two loops respectively governed by **p** and **q**. **p** $\mathcal{P}$ **q** $\Leftrightarrow$ an action of P conflicts with an action of Q (according to $\mathcal{D}$). This means that P and Q cannot commute.

2. Let P be a loop governed by **p** and **a** an imperative statement. (p,a) $\in \mathcal{P} \Leftrightarrow$ an internal action of P conflicts with **a** (according to $\mathcal{D}$): P and **a** cannot commute.

Definition 3 : Let **p** and **q** be two test statements governing P and Q, and **a** an imperative statement. P is defined by **p** $\mathcal{P}$ **q** $\Leftrightarrow \exists$ **a** $\in$ P, **b** $\in$ Q | **a** $\mathcal{D}$ **b**, and

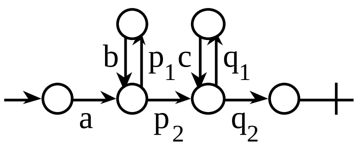
p $\mathcal{P}$ **a** $\Leftrightarrow \exists$ **b** $\in$ P | **a** $\mathcal{D}$ **b**

Here again, we represent by a graph, called *precedency graph*, the relation P, combined the the order of occurrence of the loops in the program.

Example

Let us consider the following program :

```
(a)    Get ( x ) ;
(p)    While Even ( x )
       Loop
(b)         Get ( x ) ;
       End Loop ;
(q)    While Odd ( x )
       Loop
(c)         x : = x / 2 ;
       End Loop ;
```



$\mathcal{P}$ = {(**p**, **q**), (**p**, **c**), (**q**,**p**), (**c**,**p**), (**b**, **q**), (**q**, **b**)}

Figure 4 : The precedence relation

The complete conflict relation is $\mathcal{D} \cup \mathcal{S} \cup \mathcal{P}$, denoted $\mathcal{DSP}$. The commutation relation is $(\mathcal{DSP})^c$, where $.^c$ is the complementary operator.

3. Class of semi - equivalence of a program

We are interested in all sequences of statements that can be derived from some executions of the sequential program by the commutation (or parallelisation) of statements whose order is not essential.

For technical reasons, we consider not only complete, but also partial executions of the sequential program, i.e. each state of the flowchart is terminal.

We extend in a natural way $\mathcal{DSP}$ to a relation $\mathcal{DSP}_{\text{extended}}$, defined on $\Sigma = I \cup \xi_1 \cup \xi_2$: let h be the morphism defined by h: $I \cup T_1 \cup T_2 \rightarrow I \cup T$, $\begin{cases} p_i \in T_i \rightarrow p \\ a \in I \rightarrow a \end{cases}$: (**a**, **b**) $\in \mathcal{DSP}_{\text{ext}} \Leftrightarrow$ (h(**a**), h(**b**)) $\in \mathcal{DSP}$. We still denote $\mathcal{DSP}_{\text{extended}}$ by $\mathcal{DSP}$.

We denote by L(A) the language of the flowchart.

Let w be a word of L(A), and w' $\in \Sigma^*$. w and w' are semi-equivalent $\Leftrightarrow$ w' is obtained from w by a sequence of allowed commutations :

Definition 4 : Two words w and w' are semi-equivalent if there exists a sequence $w_o = w, w_1, \dots, w_n = w'$ such that $w_i = v_i\mathbf{ab}v'_i$, $w_{i+1} = v_i\mathbf{ba}v'_i$, with $(\mathbf{a}, \mathbf{b}) \notin DSP$. We denote it by $w \equiv w'$

The set of allowed behaviours of the program, denoted by B(A), is the set of words which are semi-equivalent to a word of L (A). B(A) is a semi partially commutative monoid.

<u>Remark</u>

The set B(A) contains all the programs which have the same denotational semantics as the sequential program.

3 Examples

1. Let us consider the following program

```
(a)   x := 0
(p)   While odd(Random(100))
      Loop
(b)       x := x+1 ;
      End Loop ;
(c)   Put(x) ;
```

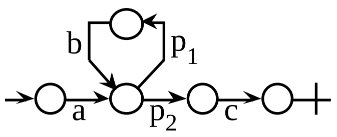


Figure 5a
Sequential flowchart

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \\ p - & & \end{tikzcd}$$

Figure 5b
Dependency graph

Figure 5

The language of the sequential flowchart (Fig. 5 a) is L(A)=Pref(a.$(p_1.b)^*p_2$c).

We have

$$B(A)=\mathrm{Pref}\left(p_1^{\,n_1}.a.b^{q_1}.p_1^{\,q_2}.b^{q_2}\dots p_1^{\,n_i}.b^{q_i}.p_2.b^{q_{i+1}}.c, \begin{cases} \forall\ j\in \left[1\,,i\right], \sum_{k=1}^{k=j} n_k < \sum_{k=1}^{k=j} q_k \\ \sum_{k=1}^{k=i} n_k = \sum_{k=1}^{k=i+1} q_k \end{cases} \right)$$

We can notice that this language is not a language of Petri net.

2. Let us consider the following program, which computes x +y [R82]. We get :

```
(a)   Get ( x ) ;
(b)   Get ( y ) ;
(p)   While x > 0 Loop
(c)         x : = x - 1 ;
(d)         y : = y + 1 ;
      End Loop ;
(e)   Put ( y ) ;
```

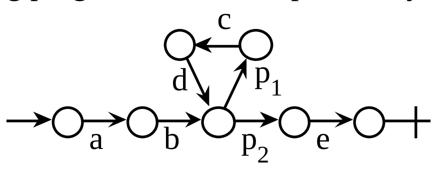


Figure 6a
Sequential flowchart

$$\begin{tikzcd} a \arrow[d] \arrow[dr] & & b \arrow[d] \arrow[dr] & \\ p \arrow[r] & c & d \arrow[r] & e \end{tikzcd}$$

Figure 6b
Dependency graph

Figure 6

4 Fifo nets

We want to find a concise representation of *B*(A). The choice of fifo nets is motivated by the fact that they integrate the notion of order and the test to zero.

The requirement of respect of the order is yielded by the relation *DP*. The test to zero appears because of the relation *S*: in the example of Figure 6, the occurrences of **d** could be delayed, but the remaining number of occurrences of **d** must be null before **e** occurs.

1. Definition

Fifo nets [F82][M83][CF88][VC92] are extensions of Petri nets : places are replaced by fifos (queues running in the *First In First Out* mode) and integers by words.

Definition 5 : A marked fifo net is defined by (N, M_o), with N = (F, T, A, W), where F is a finite set of fifos, T a finite set of transitions, A a finite alphabet,

$W : F \times T \cup T \times F \rightarrow A^*$ the valuation function and $M_o : F \rightarrow A^*$ the initial marking.

The firing rules for fifo nets are very similar to the firing rules of Petri nets : the order relation on integers is replaced by the left factor (or prefix) relation, denoted by $<_g$, the addition is replaced by the concatenation (denoted by.) of a right factor, and the subtraction by the deletion of a left factor.

Definition 6 : Let (N, M_o) be a marked fifo net and t a transition. t can occur (or is fireable) from M_o iff :

$\forall$ f $\in$ F, W(f, t) $<_g$ M_o(f). The firing of t leads to the marking M defined by :

$\forall$ f $\in$ F, W(f, t). M(f) = M_o(f). W(t,f). We denote it by M_o (t > M.

The notion of occurrence can, as for Petri nets, be extended in a quite natural way to sequences of transitions.

Example

Here, F = {f_1,f_2,f_3}, T = {x,y,z,t,u}. For example, we have W(y,f_3) = ab.

The initial marking is M_o = (*a*,*b*,ε), and we have M_o(xu> M = (ε,ba,a).

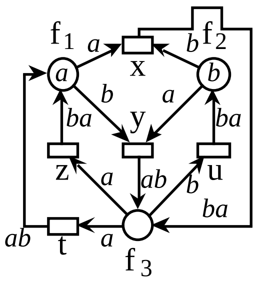


Figure 7: A fifo net

2. From the flowchart to the fifo net

We suppose that A is a flowchart modeling a sequential program and ∑ its alphabet.

We suppose the relation *DSP* computed. For simplicity, we reduce statements to atomic actions and loops only, i.e. we consider the conditional statements as meta-actions. The construction presented here is modular: loops are first considered as meta-actions, then, they are unfolded.

The first step, before the effective construction of the net, consists of a slight modification of the dependency graph, and of the definition of two functions defined on the set of tests statements, which express quantitatively, in a modular way, the relation *G*.

1. Reduction of the dependency graph and of the precedence graph

Let G_1 be the dependency graph. We denote by G the reduced dependency graph, which is a smallest graph having the same transitive closure than G_1. We procede is the same way with the precedence graph.

2. Meta-dependence and meta-precedence relations

For the purpose of modularity, we consider the general frame of the program $u_1B_1u_2B_2\ldots u_nB_nB_{n+1}$, where B_i are loops governed by tests $\mathbf{p_i}$, and u_i sequences of imperative statements. The loops B_i are of same level and we call meta-precedence graph the restriction of the reduced precedence graph to $\{\mathbf{p}_1, \ldots, \mathbf{p_n}\}$. In the same way, we associated a meta-precedence graph to each inner program of each loop.

The meta-dependence graph is the reduced restriction of the relation $\mathcal{DP}$ to the set

$$R = \{\mathbf{a} \in \mathcal{I}, \exists\ i \in \{1,\ldots, n+1\}, |u_i|_{\mathbf{a}} \neq 0\} \cup \{\mathbf{pi}, i=1,\ldots, n\}$$

(again combined with the order of occurrences).

3. Weights of an action

Let B be a loop governed by **p.**

We define the past-weight of **p** as the number of predecessors of **p** in the meta-precedence graph containing p. This is the number of loops of the same level as B, whose termination enables B to start. These loops are independent, and can be performed in parallel.

The inner-weight of an action is determined from the meta-precedence graph of the inner program of the loop. It is the number of vertices without successor in this graph. This is the number of independent loops whose termination is followed by the next occurrence of the loop.

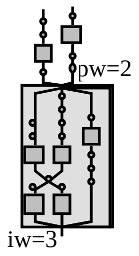


Figure 8: Weighs

Remark

Since we deel with undefined and thus possibly infinite loops, we will not delay inner loops after the termination of the including loop, even if the Bernstein relation would allow it. Indeed, the next occurrence of the including loop is conditioned by the termination of every inner ones.

We extend the past-weight function to the set of imperative statements: pw(**a**) is the number of tests such that $\mathbf{p_i} \to^* \mathbf{a}$ in the meta-dependency graph, and there is no other test along the path. It is the number of independent loops whose termination is followed by an imperative segment containing **a**.

Example

```
(a)   get(x) ;
(p)   while odd(x)
      loop
(b)         get(y) ;
(c)         get(z) ;
(q)         while odd(y)
            loop
(d)               get(u) ;
(t)               while y mod 3 = 0
                  loop
(e)                     get(y) ;
                  end loop ;
(u)               while u mod 5 /= 0
                  loop
(f)                     get(u) ;
                  end loop ;
            end loop ;
(r)         while (z < 0 ) or (z > 1000)
            loop
(g)               get(v) ;
(v)               while prime(z)
                  loop
(h)                     get(z) ;
                  end loop ;
(w)               while not prime(v)
                  loop
(i)                     get(v) ;
                  end loop ;
            end loop ;
(s)         while z mod y /= 0
            loop
(x)               while y < 0
                  loop
(j)                     get(y) ;
                  end loop ;
(y)               while z mod 3 = 0
                  loop
(k)                     get(z) ;
                  end loop ;
(z)               while not odd (u + v)
                  loop
(l)                     get(u) ;
(m)                     get(v) ;
                  end loop ;
            end loop ;
      end loop ;
```

The program consists of only one loop. The inner program is composed of 3 loops, their dependence graph is

$$\begin{tikzcd} q \arrow[dr] & & r \arrow[dl] \\ & s & \end{tikzcd}$$

The loop (**q**) is composed of 2 independent loops, the same holds for the loop (**r**). And the loop (**s**) consists of 3 independent loops. The past and the inner weight functions are :

pw(**p**)=0, iw(**p**)=1 ; pw(**q**)=0,iw(**q**)=2 ; pw(**t**)=0, iw(**t**)=1, pw(**u**)=0, iw(**u**)=1 ; pw(**r**) = 0, iw(**r**) = 2 ; pw(**v**)=0, iw(**t**)=1, pw(**w**)=0, iw(**w**)=1 ; pw(**s**)=2,iw(**s**) =3 ; pw(**x**)=0, iw(**x**) = 1 ; pw(**y**)=0, iw(**y**)=1, pw(**z**)=0, i(**z**)=1 ;

Figure 9: Past-weight and Inner weight

We can now present the effective structure of the net

a. The transitions

The set of transitions corresponds to the set of actions of the flowchart, i.e. $T \approx \Sigma$. For each statement **a** of the flowchart, there is exactly one transition labelled by a.

b.Places and fifos

A place C_p is associated to each test **p**.

A place C_a is associated to each imperative statement **a**, of past-weight not equal to zero.

To each pair (**a**,**b**) of the reduced depedency relation G, we associate a fifo $F_{a,b}$.

c.Valuation function and initial marking

Lcm denotes an extension of the classical definition : lcm(m,n) is the least common multiple if m and n are positive, lcm(0,m)=m if m≠0, lcm(0,0)=1.

The control structure

p_1 and p_2 are output transitions of C_p, with $W(C_p,p_1)=W(C_p,p_2)=p^{lcm(iw(p),pw(p))}$. This insures the synchronisation both with the precedent loops and with the inner loops. If the control place C_p contains lcm(pw(**p**), iw(**p**)) occurrences of the letter p, then, either all the past loops have terminated and the first occurrence of the test can occur, or all the inner loops have terminated, and the next occurrence of the test may occur. Thus, we do not need to distinguish between the different messages send by the different loops. If B contain no internal loop, then p_1 is an input transition of C_p : $W(p_1, C_p) = p^{W(C_p, p_1)} = p^{pw(\mathbf{p})}$. The output transitions of C_p are determined from the different loops of the program, according to the following rules:

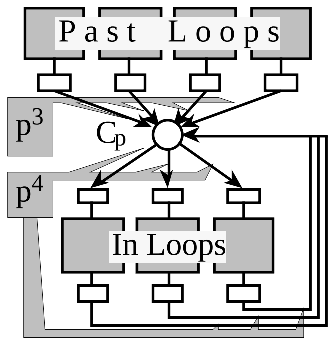


Figure 10

Connexions between loops of the same level

Let $\{B_1, B_2, \ldots, B_n\}$ be the set of loops of the same level, governed by $p_1, p_2, \ldots, p_n$. If **pj** is a predecessor of **pi** in the precedence graph,

$$W\left(pj_2, C_{pi}\right) = pi^{\frac{W(C_{pi}, pi_1)}{pw(\mathbf{pi})}} = pi^{\frac{lcm(iw(\mathbf{pi})\ pw(\mathbf{pi}))}{pw(\mathbf{pi})}}.$$

Role of internal loops

Let P be a loop governed by **p**, and $L_1, L_2, \ldots, L_m$ the internal loops of P (which may contain themselves some other loops), governed respectively by **q1**, **q2**, …, **qm**.

1. For each L_i such that **qi** has no predecessor in the meta-precedence graph defined on the set {**q1**,**q2**,…,**qm**}, $W(p_1, C_{qi}) = qi^{lcm(pw(\mathbf{qi}),iw(\mathbf{qi}))}$ (here, pw(**qi**) = 0).

2. If **qi** has no successor in the precedence graph, then $W\left(qi_2, C_p\right) = p^{\frac{W(C_p, p_1)}{iw(\mathbf{p})}} = p^{\frac{lcm(iw(\mathbf{p}), pw(\mathbf{p}))}{iw(\mathbf{p})}}$.

<u>Examples</u>

The next figures illustrate these different cases :

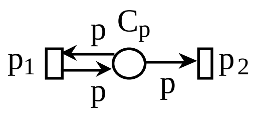
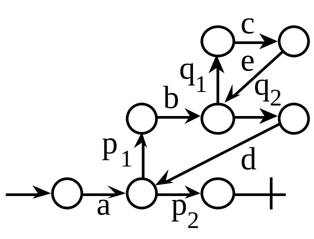
1. For the program of Figure 1, we get : 

Figure 11

2. A program with a loop included in an other :

```
(a)  c : = 3 ;
(p)  While c ≠ 25
     Loop
(b)       d : = c + 2 ;
(q)       While d ≠ Random
          Loop
(c)            Put ( d ) ;
(d)            Skip_Line ;
          End Loop ;
(e)       c : = Random ;
     End Loop ;
```

Program



Flowchart

$$\begin{tikzcd}
a \arrow[d] \arrow[dr] & \\
p \arrow[d] & \\
d & b \arrow[l] \arrow[d] \arrow[dr] & \\
& q & c
\end{tikzcd}$$

Reduced dependency graph

The associated control net is

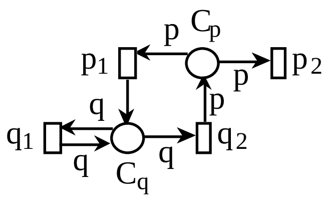
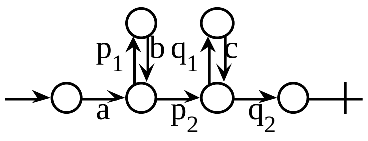


Figure 12

3. Two consecutive dependent loops

```
(a)  Get ( x ) ;
(p)  While Even ( x )
     Loop
(b)       Get ( x ) ;
     End Loop ;
(q)  While Odd ( x )
     Loop
(c)       x : = x / 2 ;
     End Loop ;
```

Program



Flowchart

$$\begin{tikzcd}
a \arrow[r] & p \arrow[r] & b \arrow[r] & q \arrow[r] & c
\end{tikzcd}$$

Reduced dependency graph

We have **p** P **q**. We get the control net

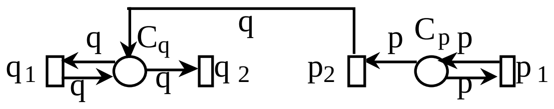
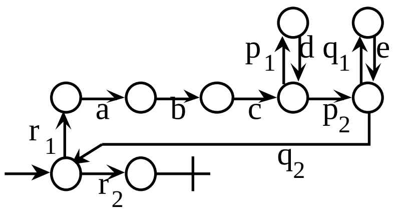


Figure 13

4. A program with a loop containing two consecutive independent loops :

```
(r)     While x > 0
        Loop
(a)         Get ( x ) ;
(b)         y : = Random ;
(c)         z : = Random ;
(p)         While x ≠ y
            Loop
(d)             y := Random ;
            End Loop ;
(q)         While x ≠ z
            Loop
(e)             z:=Random;
            EndLoop;
        EndLoop ;
```



$$\begin{tikzcd}
c \arrow[r] & q \arrow[r] & e \\
a \arrow[ur] \arrow[dr] & & \\
b \arrow[r] & p \arrow[r] & d
\end{tikzcd}$$

Program Flowchart Reduced dependency graph

The control net associated is

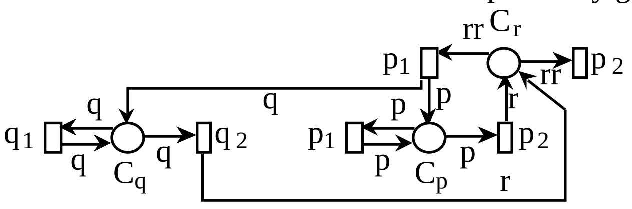


Figure 14

5. With synchronisation. For the program of Figure 8, we get the control structure

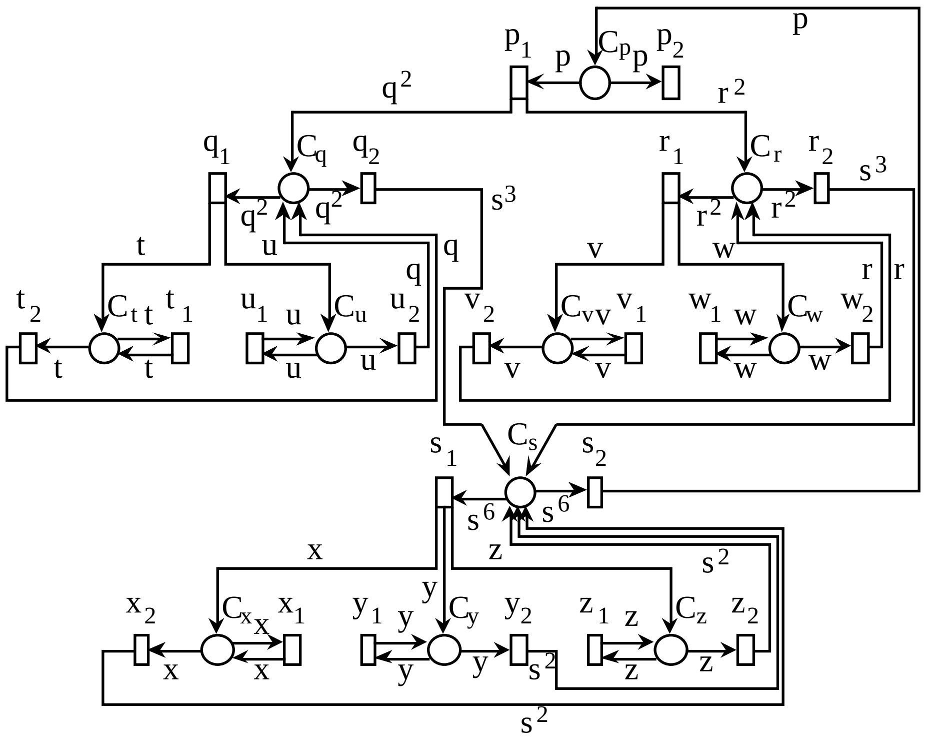
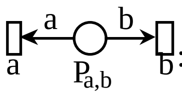


Figure 15

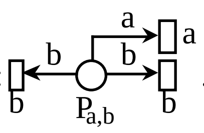
The scheduling structure

Let $F_{a,b}$ be a fifo. Two configurations can be met:

1. **a** and **b** are imperative statements, $(\mathbf{a}, \mathbf{b}) \in D$. The scheduling pattern is 

: a and b are the only output transitions of $P_{a,b}$.

2. **a** is an imperative statement, **b** is a test (associated with the transitions b_1 and b_2), $(\mathbf{a}, \mathbf{b}) \in D$.

In this case, the scheduling pattern contains three transitions: 

.

Remark

There is no fifo $P_{a,b}$ with **a** and **b** test statements, since two test statements cannot be conflicting in the sense of the relation D.

The input transitions of $F_{a,b}$, their valuation and the initial marking of $F_{a,b}$ are described by the following algorithm:

Let $\mathbf{p_a}$ denotes the deepest loop (i.e. its test action) governing an action $\mathbf{a} \in \xi \cup I$, i.e. $\mathbf{p_a} \neq \mathbf{a}$, and let $p_a=0$ if **a** belongs to the main program $u_1 p_1 \ldots u_n p_n u_{n+1}$. Furthermore, let $\mathbf{p_a} > \mathbf{p_b}$ denote the fact that the loop governed by $\mathbf{p_a}$ is an inner loop of the one governed by $\mathbf{p_b}$.

1st Case : $(\mathbf{a},\mathbf{b}) \in I^2$.

We makes the distinction between the following situations (where $W(p,P_{a,b})=x$ has to be replaced by $M_O(P_{a,b})=x$, if p=0):

1. $\mathbf{p_a} > \mathbf{p_b}$: Let **p** be such that $\mathbf{p_a} \geq \mathbf{p} > \mathbf{p_b}$, and $\mathbf{p} \geq \mathbf{p'} > \mathbf{p_b}$ implies $\mathbf{p}=\mathbf{p'}$. Then, we add the backwards arcs $W(p_{a_2},P_{a,b})=b$, and $W(p_{a_1}, P_{a,b})=a$.
2. $\mathbf{p_a}=\mathbf{p_b}$: Let $W(p_{a_1}, P_{a,b})=ab$.
3. $\mathbf{p_b} > \mathbf{p_a}$, or $\mathbf{p_a}, \mathbf{p_b}$ incomparable: Let $W(p_{a_1}, P_{a,b})=a$, and $W(p_{b_1}, P_{a,b})=b$.

2ndcase : $\mathbf{a} \in I$ and $\mathbf{b} \in \xi$

Itcorresponds exactly to the first case extended to an additional backword arc $W(b_1, P_{a,b})=b$.

3rd case: $\mathbf{a} \in \xi$ and $\mathbf{b} \in I$

We consider five subcases, including in each case the arc $W(p_{a_1}, P_{a,b})=a$ (recall the special case p=0):

1. $\mathbf{p_a} \geq \mathbf{p} > \mathbf{p_b}$ (with **p** as above): Let $W(a_1, P_{a,b})=a$, and $W(p_2, P_{a,b})=b$.
2. $\mathbf{p_a}=\mathbf{p_b}$: Let $W(a_1, P_{a,b})=a$, and $W(a_2, P_{a,b})=b$.
3. ($\mathbf{p_b} > \mathbf{p_a}$, and $\mathbf{p_b} < \mathbf{a}$), or $\mathbf{p_a},\mathbf{p_b}$ incomparable: Let $W(a_1, P_{a,b})=a$, and $W(p_{b_1}, P_{a,b})=b$.
4. $\mathbf{p_b} \geq \ldots \geq \mathbf{p} > \mathbf{a}$ (with $\mathbf{p} \geq \mathbf{p'} > \mathbf{a}$, implying $\mathbf{p}=\mathbf{p'}$): Let $W(b_1, P_{a,b})=b$, and $W(p_2, P_{a,b})=a$
5. $\mathbf{p_b}=\mathbf{a}$: Let $W(a_1, P_{a,b})=ba$.

Synchronisation structure

Each place C_a, with **a** imperative statement, admits a as only output transition. The place C_a takes the synchronisation into account, i.e. it enables the firing of transition a only if every loop which directly precedes **a** (in the sense of *DSP*) has occurred. We have : $W(C_a, a) = a^{lcm(0,pw(\mathbf{a}))}$.

The control place of **a** is filled by the output transitions of the loops which directly precede **a**, and emptied by transition a. The content of such places determine the behaviour of the net. In the following, we describe how to fill these places.

Starting sequence

We first extract a subset of actions of the program, containing the actions (imperative or test statements) which can occur first, i.e. which do not use any result provided by any loop. We then order these actions according to their order of appearance in the program, so we get a word called the **initial segment**, denoted s_o :

$$\mathbf{a} \text{ occurs in } s_o \Leftrightarrow \mathbf{a} \text{ occurs in } u_1\mathbf{p1}u_2\mathbf{p2}\ldots\mathbf{pn}u_{n+1} \text{ and } pw(\mathbf{a})=0$$

Remarks

The sequence u_1**p1** is a prefix of s_o.
The set of actions which occur in s_o can be deduced from the meta-dependency graph: **a** occurs in s_o iff there is a source **r** of the graph (i.e. a vertex without predecessor) such that there exists a path from **r** to **a** which contains no test statement (except **a** eventually).

Contextual sequences

Each occurrence of a test gives rise to a sequence of actions. We associate to each test **p** two segments, denoted by p^+ and p^- , which correspond respectively to the positive and to the negative alternative of **p**.

1. In order to get the negative segment p^- , we proceed in the same way as for s_o, with the restriction of the meta-dependency graph to the sub-tree with source **p**. This segment contains the actions which can occur just after the loop governed by **p** has terminated.

2. In order to get the positive segment p^+ and the segments associated to the inner tests of the loop B governed by **p**, we unfold B, and we proceed in the same way as described above with the inner program of the loop.

Remarks

If an action must occur after the termination of several loops, i.e. it depends of internal actions of several loops, then the past-weight of this action is the number of such loops, and this action appears in the negative segment of each test.

We get in this way 2m + 1 segments, if m is the number of loops of the program.

Examples

1. Let us consider again the program of Figure 2. There is one loop, so we get 3 segments:
s_o = **abp**, p^+ = **cd**, p^- = **e**.

2. If we consider the program of Figure 12, we get 5 segments:

s_o = **ap**, p^+ = **bqe**, p^- = ε, q^+ = **cd**, q^- = ε.

3.- If we consider the program if Figure 13, we get 5 segments:

s_o = **ap**, p^+ = **b**, p^- = **q**, q^+ = **c**, q^- = ε.

4. If we consider the program of Figure 14, we get 5 segments:

s_0 = **r**, r^+ = **abcpq**, r^- = ε, p^+ = **d**, p^- = ε, q^+ = **e**, q^- = ε.

5. Let us consider the next program:

```
(a)  get(x) ;
(b)  get(y) ;
(c)  get(z) ;
(p)  while x < 0 loop
(d)         x : = x + 4 ;  end loop ;
(q)  while y < 0 loop
(e)         y : = y + 2 ;  end loop ;
(r)  while odd(z) loop
(f)         z : = z/2 ;  end loop ;
(g)  x : = x + y ;
(h)  x : = x + z ;
```

We get 7 segments :

s_0 = **abcpqr**, p^+ = **d**, p^- = **gh**,

q^+ = **e**, q^- = **gh**, r^+ = **f**, r^- = **h**,

and we have pw(**g**) = 2, pw(**h**) = 3.

We can now define the initial marking as well as the missing valuations.

Remark

$\forall$ **a** $\in$ I, if **a** occurs in s_0, there is no control place C_a.

The initial marking is defined by $M_0(C_a) = a^{|s_0|_a}$, where $|s_0|_a$ denotes the number of occurrences of **a** in s_0. So, we have $M_0(C_a) = \begin{cases} \varepsilon \text{ if } a \in I \\ a \text{ if } a \in T \text{ and a occurs in } s_0 . \\ \varepsilon \text{ otherwise} \end{cases}$

Let P be a loop, p_1 and p_2 its two associated actions, p^+ and p^- the segments associated to the test **p**. For each place C_a (**a** imperative action), we have $W(p_1, C_a) = a^{|p^+|_a}$ and $W(p_2, C_a) = a^{|p^-|_a}$.

Example

We consider again the program of figure 6. The past-weights of the actions **a**, **b**, **c**, **d** are null (no control place for these actions) and the past-weight of **e** is equal to 1 (there is a control place C_e). The whole net is as follows:

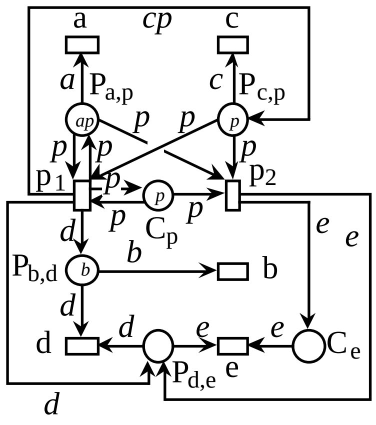


Figure 16: A complete fifo net

5. Properties of the model

In this section, we present the relation between the behaviours of the fifo net and the behaviours of the flowchart. Frst, we do not lose any information in the sense that the fifo net's behaviours contain the flowchart's behaviours. Then, every behaviour which is semi-equivalent to a behaviour of the flowchart is also a behaviour of the fifo net. Conversely, the behaviours of the fifo nets are only these ones. The model meets thus our requirement. The detailed proofs are rather technical, we do not present them here. They can be found in the detailed version of this paper.

From the flowchart to the fifo net

Every word accepted by the automaton (where each state is terminal) is a sequence of the net. Let Prog be a program, represented by the flowchart A, and (N, M_o) be the associated fifo net. The set of behaviours of A is the language of A, $\mathcal{L}$(A), and these of the fifo net the language of the fifo net, L(N, M_o).

Proposition 1 : $\mathcal{L}(A) \subseteq L(N, M_o)$

We prove this result by induction on the length of the sequence of $\mathcal{L}$(A).

Furthermore, every word which is semi-equivallent to a word of $\mathcal{L}$(A) is also a behaviour of the fifo net.

Proposition 2 : $\mathcal{B}(A) \subseteq L(N, M_o)$

We consider two words u$\in \mathcal{L}$(A), and v such that $u \equiv v$; we prove by induction on the length of the commutation chain which leads from u to v, that v also belongs to L(N, M_o). Therefore, we consider a sequence of commutations.

The discussion lays then on the nature of the permuted transitions (imperative or test statements).

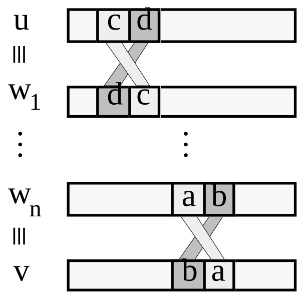


Figure 17

From the fifo net to the flowchart

Conversely, we have the following inclusion, where LF(E) is the set of left factors, or prefix of E:

Proposition 3 : $L(N, M_o) \subseteq LF(\mathcal{B}(A))$

We prove that $\forall$ u $\in$ L(N,M_o), $\exists$ v, w | u.v $\in$ L(N,M_o), w $\in$ $\mathcal{L}$(A) and u.v $\equiv$ w. This result is showed by induction on |u|.

Thus, we have the following theorem :

Theorem : $LF(\mathcal{B}(A)) = L(N, M_o)$

6 Conclusion

We have presented a methodology for transforming the flowchart of a sequential program into a flowchart (named object flowchart in the following) which collects all its possible behaviours. This construction can be applied to programs which contain basic statements, and loops whose iteration numbers can be not pre-defined.

The object flowchart must integrate scheduling (collect *all* the possible behaviours) and the test to zero (not pre-defined number of loop iterations): it is a fifo net. Its construction is based on the conflict relation induced by the semantics of the sequential program. This flowchart collects exactly the set of possible behaviours of the program semantically equivalent to its sequential behaviours.

Two main directions are natural for further developments of this work. The fifo-net obtained can be analyzed by marking graph tools. Such analysis should help the optimization step when compiling onto a parallel object code. On an other hand, the fifo-net contain the whole scheduling control of the program. So, it can be used in *in-line* mode to drive the software, whatever the physical topology of the used machine (time-sharing, parallel) is.

7 References

[AN82] A. Arnold, M. Nivat, *Comportements de processus*, LITP Rapport nr 82-12, 1982

[C91] M.Cosnard, *Parallélisation d'algorithmes: une méthodologie*, LIP tech. report, 1991

[C92] C. Cerin, *Automatic parallelization of programs with tools of trace theory*, IEEE International processing Symposium (IPPS), Beverly Hills, March 23-26 1992, pp 374-379

[CF88] A. Choquet, A. Finkel, *Fifo nets without order deadlock*, Acta Informatica 25, pp 15-36, 1988

[CP93] C. Cerin, A. Petit, *Speedup of recognizable trace languages*, MFCS 93, GDANSK, Poland

[E76] S.Eilenberg, *Automata Languages and machines*, vol. A Academic press 1976

[F92] P.Feautrier, *Techniques de parallélisation*, chap. 17, in Y.Robert Ed., Ecole de printemps du LITP, Masson, 1992

[F82] A. Finkel, *Deux classes de réseaux à files : les réseaux monogènes et les réseaux préfixes* PhD Thesis, L.I.T.P. report nr 83-03, October 1982, Paris

[M83] G. Memmi, *Méthodes d'analyse de réseaux de Petri, réseaux à files et application aux systèmes temps réel*, Thèse d'état, University of Paris 6, June 1983

[P62] C.A. Petri, *Kommunication mit automaten*, Institüte für Instrumentelle Mathematik, Schriften des IMM, nb 2, 1962

[R82] W. Reisig *Petri nets, an introduction*, Springer Verlag, 1982

[R86] G. Roucairol, *Fifo nets*, Advanced course in Petri nets, Badhonnef, 1986, LNCS 254, p 436 - 459

[VC92] G. Vidal-Naquet, A. Choquet-Geniet, *Réseaux de Petri et systèmes parallèles*, Ed. Armand Colin, Paris, 1992