VNU Journal of Science, Mathematics - Physics 23 (2007) 113-121

Some methods for transforming sequential processes into concurrent ones

Hoang Chi Thanh[*]

Department of Mathematic Mechanics Informatics, College of Science, VNU
334 Nguyen Trai, Thanh Xuan, Hanoi, Vietnam

Received 31 October 2006; received in revised form 2 August 2007

Abstract: In this paper we investigate and build up three methods for transforming sequential processes of a net system into concurrent ones. These methods are based on: trace languages, shift-left and case graphs. They are also presented by detail algorithms and can be applied to other models of concurrent systems.

Keywords: net system, reliance alphabet, trace language, maximal concurrent steps, case graph.

1. Introduction

The behaviour of a system often is represented by the set of the system's processes. The behaviour shows what the system can do in the way of performing processes. It is easy to formalize sequential behaviour of a given system. The behaviour usually is presented by the language generated by the system. Concurrent process with maximal concurrent steps points out an optimal way to perform the corresponding sequential process. Therefore, constructing methods to transform sequential processes of a system into concurrent processes is still a meaningful problem in system controls.

This paper concentrates in building up three methods for transforming sequential processes into concurrent ones.

The first method is based on trace languages and the normal form of traces [1-3]. The trace language generated by a system can be determined by the language generated by a system and the detached relation over events of the system. The algorithm finding the normal form of a trace shows us how to find maximal concurrent steps on each trace. After normalizing, the trace language generated by a system gives us the whole concurrent behaviour of the system.

The second method is based on the following observation [4]: an event in a sequence of events can be attemped on any concurrent step on the head of the sequence if it is concurrent of each its left-side event. So we construct a "shift-left" algorithm to transform a sequence of events into a sequence of maximal concurrent steps.

The third method is constructed by case graphs. The case graph is a geometrical representation of the behaviour of a net system [5,6]. In [6] we applied edge adding technique to obtain the complete case graph. Here, we are interested in maximal concurrent steps. So we merge edges after some rules to obtain a reduced case graph, whose labels on paths point out sequences of maximal concurrent steps.

[*] Tel.: 84-4-5585840
E-mail: thanhhc@vnu.edu.vn

A popular model to represent concurrent systems is a net system. We recall some important notions.

1.1. Net systems

Based on a simple Petri net, a net system is defined in [2,3,5] as follows.

Definition 1: A triple N = (B, E; F) is called a *Petri net* iff:

1. B, E are two disjoint sets,
2. $F \subseteq (B \times E) \cup (E \times B)$ is a binary relation, the *flow relation* of N.

Elements of the set B present *conditions* and elements of the set E present *events* of the system. The flow relation F shows the relationship between two these objects.

Let N = (B, E; F) be a net. For $x \in X_N = B \cup E$,

$^\bullet x = \{y \mid (y,x) \in F\}$ is called the *preset* of *x*,

$x^\bullet = \{y \mid (x,y) \in F\}$ is called the *postset* of *x*.

The net N is called *simple* iff distinct elements do not have the same pre- and postset.

Each subset $c \subseteq B$ is called a *case* of the net N.

Let $e \in E$ and $c \subseteq B$. The event *e* is *c-enabled* iff $^\bullet e \subseteq c \wedge e^\bullet \cap c = \emptyset$. Then, $c' = (c \setminus {}^\bullet e) \cup e^\bullet$ is called the *follower case* of *c* under *e* (*c'* results from the occurrence of *e* in the case *c*) and we write: $c[\, e > c'$.

The occurrence of events on the net N forms a forward reachability relation $r_N \subseteq 2^B \times 2^B$, defined as follows:

$$\forall\, c, c' \in 2^B \ :\ (c, c') \in r_N \ \Leftrightarrow\ \exists\, e \in E,\ c[\, e > c'.$$

The reflexive and transitive closure of the forward and backward reachability relation, $R_N = (r_N \cup r_N^{-1})^*$ gives us the *reachability relation* over the net N. It is an equivalence relation over the 2^B.

Definition 2: A quadruple $\Sigma = (B, E; F, c_0)$ is called a *net system* iff:

1. N = (B, E, F) is a simple net without isolated elements.
2. $c_0 \subseteq B$ is a case (the *initial case*) of the net N.

The equivalence class $C = [c_0]_R$ is called the *state space* of Σ.

An event *e* may occur in a case *c* if and only if the preconditions of *e* belong to *c* and the postconditions of *e* do not belong to *c*. When *e* occurs, the preconditions of *e* cease to hold and the postconditions of *e* begin to hold. After the occurrence of an event, the obtained follower case must belong to the state space. Other events may be enabled by this case. A system's state space is the environment for occurrence of steps on the system. The net system is used to model concurrent systems, such as communicating systems, operation systems, industrial production systems, concurrent programs ...

Let $\Sigma = (B, E; F, c_0)$ be a net system and there are a sequence of cases $c_1, c_2, \dots, c_n, c_{n+1}$ belonging to C and a sequence of events $e_1, e_2, \dots, e_n$ belonging to E such that: $c_i[\, e_i > c_{i+1}$, $i = 1, 2, \dots, n$.

Then the sequence:

$c_1[\, e_1 > c_2[\, e_2 > c_3 \dots c_n[\, e_n > c_{n+1}$

presents a sequential process occurred on the system.

Definition 3: The *language* generated by Σ is defined as follows:

$$L(\Sigma) = \{e_1e_2 \dots e_n \mid \exists\, c_1, c_2, \dots, c_n, c_{n+1} \in C,\ \exists\, e_1, e_2, \dots, e_n \in E\ :\ c_i[\, e_i > c_{i+1},\ i = 1, 2, \dots, n\}.$$

The language generated by a net system describes all sequences of events occurred on the system. It is used to represent the system's behaviour. But the language shows us only the sequences of events, which occurr sequentially on the system. So how to detect the system's behaviour, in which some events may occurr concurrently? Futhermore, can we result it from the sequential language of the system?

In the next sections, we build three methods for transforming sequential processes into concurrent ones.

First of all, we formalize concurrent steps on the net system.

1.2. Concurrent steps on a net system

Let $\Sigma = (B, E; F, c_0)$ be a net system.

Definition 4:

1. A relation $q \subseteq E \times E$ is called a *detached relation* iff:

$$\forall e_1, e_2 \in E : \quad (e_1, e_2) \in q \iff e_1 \neq e_2 \wedge {}^\bullet e_1 \cap {}^\bullet e_2 = {}^\bullet e_1 \cap e_2{}^\bullet = e_1{}^\bullet \cap {}^\bullet e_2 = e_1{}^\bullet \cap e_2{}^\bullet = \varnothing.$$

2. A subset $G \subseteq E$ is called a *detached set* iff:

$$\forall e_1, e_2 \in G, \ e_1 \neq e_2 \Rightarrow (e_1, e_2) \in q.$$

3. Let c, c' be cases and G be a detached set. G is called a *step* on Σ from c to c' iff each $e \in G$ is c-enabled and $c' = (c \setminus {}^\bullet G) \cup G^\bullet$.

Then we denote: $c[\, G > c'$.

Of course, the detached relation is symmetrical and irreflexive (sir-relation). If G is a step on the net system Σ then the events belonging G can occurr concurrently. Therefore, quick determining concurrent steps on a net system becomes a meaningful problem. Usually, we are interested in, as big as possible, concurrent steps.

Theorem 1 [4]: Let G be a detached set on a net system Σ and let c, c' be two cases of Σ. Then:

$c\,[\, G > c' \iff (c \setminus c' = {}^\bullet G) \wedge (c' \setminus c = G^\bullet)$.

The above theorem can be used to detect concurrent steps on a net system. But the complexity of the corresponding algorithm will be very high.

2. Concurrency transformation by trace languages

The theory of traces was originated by A. Marzurkiewicz in [2] as an attempt to provide a good mathematical description of the behaviour of concurrent systems. The normal form of a trace gives an optimal way to perform a process represented by the trace. So we apply trace languages to net systems.

Definition 5: Let A be an alphabet.

1. *An independence relation* over A is a symmetric and irreflexive binary relation p over A.

$\forall a, b \in A : \ (a,b) \in p \iff (b,a) \in p \Rightarrow a \neq b.$

2. *A reliance alphabet* is a couple (A, p), where A is an alphabet and p is an independence relation over A.

Considering adjacent independent symbols in a string to be commuting, one can relate different strings as follows.

Let $\mathcal{A} = (A, p)$ be a reliance alphabet.

1. The relation $\sim_p \subseteq A^* \times A^*$ is defined as follow: for $x, y \in A^*$, $x \sim_p y$ iff there exist x_1, $x_2 \in A^*$ and $(a,b) \in p$ such that $x = x_1abx_2$ and $y = x_1bax_2$.

2. The relation $\approx_p \subseteq A^* \times A^*$ is defined as the least equivalence relation over A^* containing $\sim_p$. The relation $\approx_p$ is indeed the reflexive and transitive closure of $\sim_p$.

Thus, $\sim_p$ identifies 'commutatively similar' strings – each such group of strings is called a trace.

Definition 6: Let $\mathcal{A} = (A, p)$ be a reliance alphabet.

1. For an $x \in A^*$, *the trace of* x, denoted by $[x]_p$, is the equivalence class of $\approx_p$ containing x.
2. A set of traces over $\mathcal{A}$ is called *a trace language* over $\mathcal{A}$.

For traces t_1, t_2 over $\mathcal{A}$, *the trace composition* of t_1 and t_2, denoted by $t_1.t_2$, is defined by $t_1 \cdot t_2 = [x_1.x_2]_p$, where $x_1, x_2 \in A^*$ are representatives of t_1 and t_2 respectively.

In general, a trace can be obtained as a trace composition of other traces, but trace decompositions of the given trace do not have to be unique. Hence it is desirable to have 'normal form' decomposition of traces.

Definition 7: Let $\mathcal{A} = (A, p)$ be a reliance alphabet and let t be a trace over $\mathcal{A}$. The following decomposition $t = t_1.t_2. \ldots t_m$, such that:

1. for all $1 \leq i \leq m$, $t_i \neq \Lambda$;
2. for all $1 \leq i \leq m$, t_i can be written as $[u_i]_p$, where $u_i \in A^*$, $\#_a(u_i) = 1$ for each $a \in \text{alph}(u_i)$, and $(a,b) \in p$ for all $a, b \in \text{alph}(u_i)$ such that $a \neq b$;
3. for all $1 \leq i \leq m$-1, if $t_i = [u_i]_p$ and $t_{i+1} = [u_{i+1}]_p$ then, for each $a \in \text{alph}(u_{i+1})$, there exists $b \in \text{alph}(u_i)$, such that $(a,b) \notin p$;

is called *the normal form* of the trace t.

In [1] J. I. Aalbersberg and G. Rozenberg pointed out that every trace can be uniquely decomposed into a normal form, i.e. a minimal number of 'maximal independent' parts. They have built two algorithms for finding the normal form of a trace. The first algorithm is based on integer pointers and the second one is based on dependence graphs. We recall the first one.

Algorithm 2 [1]:

Input: A reliance alphabet $\mathcal{A} = (A, p)$ and a string $w \in A^*$.

Declaration: Let $l = |w|$, $n = \#A$ and $A = \{a_1, a_2, \ldots, a_n\}$; let k be a variable over $\{0, 1, \ldots, l\}$ and let d be an array of length n over $\{1, 2, \ldots, l+1\}$; let i, j be variables over $\{1, 2, \ldots, n\}$; let u be an array of length l over $(A \cup \{\lambda\})^*$; let m be a variable over $\{1, 2, \ldots, l\}$.

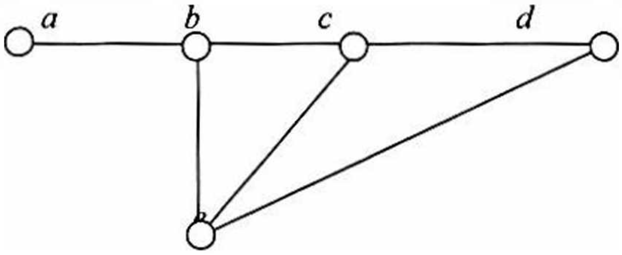
Computation:

```
1.  For all $i = 1, 2, \ldots, n$, $d(i) := 1$
2.  For all $k = 1, 2, \ldots, l$, $u(k) := \lambda$
3.  $k := 0$
4.  $k := k + 1$
5.  set $j$ such that $w(k) = a_j$
6.  $u(d(j)) := u(d(j))a_j$
7.  For all $i = 1, 2, \ldots, n$ such that $i \neq j$, $d(i) \leq d(j)$ and $(a_i, a_j) \notin p$,   $d(i) := d(j)+1$
8.  $d(j) := d(j) + 1$
9.  if $k < l$ then goto 4)
10. set $m$ such that $u(m) \neq \lambda$ and either $u(m+1) = \lambda$ or $m = l$
11. stop.
```

Output: The strings $u(1), u(2), \ldots, u(m)$.

Each $w \in A^*$ gives the corresponding trace $t = [w]_p$, applying the above algorithm we obtain the normal form of $t = [u(1)]_p.[u(2)]_p \dots [u(m)]_p$, where the steps $\overline{u(i)}$ are maximal independent.

Example 8: Let $\mathcal{A}$ = (A, p) be the reliance alphabet given by the following undirected graph.



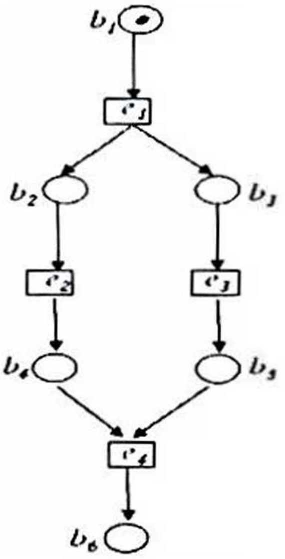
Let w = *aecbbed*. Computing by the algorithm 2, we obtain $u(1) = ab$, $u(2) = bce$, $u(3) = de$. This means the normal form of the trace $t = [w]_p$ is $[ab]_p.[bce]_p.[de]_p$. So, instead of performing the sequential process in 7 steps, one can perform it concurrently in 3 steps.

Now we apply the above technique to net systems. Let $\Sigma = (B, E; F, c_0)$ be a net system. The reliance alphabet defined on the net system is $\mathcal{E}$ = (E, q), where E is the set of events and q is the detached relation over E. Then, the trace language $V(\Sigma) = L(\Sigma) / \approx_q$ represents all concurrent processes of the net system.

Using the algorithm 2 to find normal form for each trace, we get concurrent processes with maximal concurrent steps on the given net system.

Note that the input of the algorithm 2 is a representative of the trace. Hence, the algorithm 2 transforms directly strings of the language L(Σ) into the set of normal forms of traces. This is just the set of all sequences of maximal concurrent steps on Σ.

Example 8: Consider the net system Σ given by the following directed graph.



The set of events $E = \{e_1, e_2, e_3, e_4\}$ and the detached relation $q = \{(e_1, e_4), (e_2, e_3)\}$. The language generated by Σ is $L(\Sigma) = \{e_1e_2e_3e_4, e_1e_3e_2e_4\}$. It is just the trace language V(Σ) generated by the net system Σ.

Applying the algorithm 2 to strings of L(Σ) we get the sequence of maximal concurrent steps $\{e_1\}, \{e_2, e_3\}, \{e_4\}$. Thus, events e_2 and e_3 may be performed concurrently.

In this technique, we recognize that the cases c_i were hidden. But the following result ensures the finding normal form of the trace t on the net system is correct.

Theorem 3: Let $c_1 [e_1 > c_2 [e_2 > c_3 \ldots c_i [e_i > c_{i+1} [e_{i+1} > c_{i+2} \ldots c_n [e_n > c_{n+1}$ be a sequence of activities on a net system Σ. If $(e_i, e_{i+1}) \in q$ then e_{i+1} is c_i -enabled. In other words, $c_i [\{e_i, e_{i+1}\} > c_{i+2}$ and $\{e_i, e_{i+1}\}$ becomes a concurrent step on Σ.

Proof:

By the definition of the reachability, $c_{i+1} = (c_i \setminus {}^\bullet e_i) \cup e_i^\bullet$.

Futhermore, c_{i+1} enables e_{i+1} then ${}^\bullet e_{i+1} \subseteq c_{i+1} \wedge e_{i+1}^\bullet \cap c_{i+1} = \varnothing$. So, ${}^\bullet e_{i+1} \subseteq ((c_i \setminus {}^\bullet e_i) \cup e_i^\bullet) \wedge e_{i+1}^\bullet \cap ((c_i \setminus {}^\bullet e_i) \cup e_i^\bullet) = \varnothing$. Because e_i and e_{i+1} are detached then ${}^\bullet e_{i+1} \subseteq c_i \wedge e_{i+1}^\bullet \cap c_i = \varnothing$. Thus e_{i+1} is c_i -enabled and $\{e_i, e_{i+1}\}$ is a concurrent step on the net system Σ.

The complexity of the algorithm: The transformation is executed on strings of the language generated by a net system Σ. The length of each string (or its cycle) belonging to L(Σ) is not greater than |C|. So, by [1] the complexity is $O(|E|.|C|)$.

Each case can enable at most |E| events. Thus, the language generated by Σ consists of at most $|E|^{|C|}$ strings. Then the total complexity of the above algorithm is $O(|C|.|E|^{|C|+1})$.

3. "Shift-left" algorithm

Let $c_1 [G_1 > c_2 [G_2 > c_3 \ldots c_i [G_i > c_{i+1} [G_{i+1} > c_{i+2} \ldots c_n [G_n > c_{n+1}$ be a concurrent process of the net system Σ.

If there are some events belonging to G_{i+1}, when adding them to the previous step G_i , the obtained set is still detached and is c_i-enabled, then we shift these events from the step G_{i+1} into the step G_i and change the case c_{i+1} appropriately. After shifting, if the step G_{i+1} becomes empty then we ignore both c_{i+1} and G_{i+1}. Repeat this until no event can be shifted.

When the algorithm terminates we get a sequence of maximal concurrent steps of the given net system Σ.

The input of the algorithm is a sequence of single events. This sequence may be considered as a sequence of concurrent steps, where each step consists of only one event.

Algorithm 4 ("Shift-left" algorithm):

Input: A sequence of activities $c_1 [e_1 > c_2 [e_2 > c_3 \ldots c_n [e_n > c_{n+1}$ on the net system Σ.

Computation:

```
for $i := 1$ to $n$ do $G_i := \{e_i\}$ ;
  for $j := 1$ to $n-1$ do
    begin
      $i := j$ ;
      while $i \geq 1$ do
        begin
        for every event $e \in G_{i+1}$ do
          if $G_i \cup \{e\}$ is detached $\wedge$ $c_i$ enables $(G_i \cup \{e\})$ then
            begin
        $c_{i+1} := (c_{i+1} \setminus {}^\bullet e) \cup e^\bullet$;
         $G_i := G_i \cup \{e\}$ , $G_{i+1} := G_{i+1} \setminus \{e\}$; {shift the event e from $G_{i+1}$ into $G_i$}
               if $G_{i+1} = \varnothing$ then ignore both $c_{i+1}$ and $G_{i+1}$ ;
        end ;
            $i := i - 1$
          end ;
      end .
```

Output: A sequence of maximal concurrent steps on Σ.

The algorithm goes back because we want not only making steps maximal concurrent but also reducing the number of steps.

The algorithm's complexity: The algorithm transforms a sequential process into a concurrent one with the complexity of $O(|C|^2)$.

Therefore, the total complexity of the algorithm is $O(|C|^2.|E|^{|C|})$.

Example 9: Applying the "shift-left" algorithm to the net system given in Example 8, we get the following sequence of maximal concurrent steps $\{e_1\},\{e_2, e_3\},\{e_4\}$.

4. Reducing case graphs

To view broadly all processes happened on a net system we costruct a case graph of the system. It is a directed graph, whose nodes are cases of the net system and labels on edges describe the occurrence of events on the net system.

Definition 10: Let Σ be a net system. Construct the following set of labelled edges.

$$P = \{ (c, e, c') \in C \times E \times C \mid c[e > c' \}$$

Labelled directed graph Φ = (C, P) is called a *case graph* of the net system Σ.

In the case graph Φ, each edge (c, e, c') can be considered as the edge (c, G, c') with the single step $G = \{e\}$. Therefore, the labels on edges are steps. In order to find maximal concurrent steps on Σ, we reduce the case graph of the net system. The reducing is based on the following results.

Theorem 5: Let Σ be a net system and let $c_1, c_2, c_3 \in C$; $G_1, G_2 \subseteq E$.

If (c_1, G_1, c_2) and (c_1, G_2, c_3) are two edges of the case graph Φ, $G_1 \cap G_2 = \varnothing$ then (c_2, G_2, c_4) and (c_3, G_1, c_4) with $c_4 = (c_1 \setminus {}^\bullet G) \cup G^\bullet$ are just two edges of the graph Φ and if $G = G_1 \cup G_2$ is detached then $c_1 [G_2 > c_4$.

Proof:

Since $c_1[G_1 > c_2$ then ${}^\bullet G_1 \subseteq c_1$, $G_1^\bullet \cap c_1 = \varnothing$ and $c_2 = (c_1 \setminus {}^\bullet G_1) \cup G_1^\bullet$.

Similarly, since $c_1[G_2 > c_3$ then ${}^\bullet G_2 \subseteq c_1$, $G_2^\bullet \cap c_1 = \varnothing$ and $c_3 = (c_1 \setminus {}^\bullet G_2) \cup G_2^\bullet$. Futher, because G_1, G_2 are detached and $G_1 \cap G_2 = \varnothing$ then we have ${}^\bullet G_2 \subseteq c_2$ and $G_2^\bullet \cap c_2 = \varnothing$. Thus $c_2[G_2 > c_4$ with $c_4 = (c_1 \setminus {}^\bullet G) \cup G^\bullet$. Analogously, we prove that $c_3[G_1 > c_4$.

Because the set G is detached then $c_1[G > c_4$.

Theorem 6: Let Σ be a net system and let $c_1, c_2, c_3 \in C$; $G_1, G_2 \subseteq E$.

If (c_1, G_1, c_2) and (c_2, G_2, c_3) are two edges on the case graph Φ then $G_1 \cap G_2 = \varnothing$ and (c_1, G_2, c_4), (c_4, G_1, c_3) with $c_4 = (c_1 \setminus {}^\bullet G) \cup G^\bullet$ are two edges of the graph Φ and if the set $G = G_1 \cup G_2$ is detached then $c_1[G > c_3$.

Proof:

Assume that $e \in G_1$. Thus ${}^\bullet e \cap c_2 = \varnothing$. So e is not c_2-enabled. It means $e \notin G_2$. We have $G_1 \cap G_2 = \varnothing$.

Since ${}^\bullet G_1 \subseteq c_1$ and ${}^\bullet G_2 \subseteq c_1$ we get ${}^\bullet G_1 \cup {}^\bullet G_2 \subseteq c_1$. Futher, because $G = G_1 \cup G_2$ is detached then ${}^\bullet G = {}^\bullet G_1 \cup {}^\bullet G_2 \subseteq c_1$. Analogously, because $G_1{}^\bullet \cap c_1 = \emptyset$, $G_2{}^\bullet \cap c_1 = \emptyset$ we have $(G_1{}^\bullet \cup G_2{}^\bullet) \cap c_1 = \emptyset$.

Hence G_2 is c_1-enabled and we get $c_1[\ G_2 > c_4$ with $c_4 = (c_1 \setminus {}^\bullet G_2) \cup G_2{}^\bullet$ and $c_4[\ G_1 > c_3$. Sinee $G = G_1 \cup G_2$ is detached then $G^\bullet = G_1{}^\bullet \cup G_2{}^\bullet$. We have $G^\bullet \cap c_1 = \emptyset$. It means $c_1\ [G > c_3$.

Using Theorem 5 and Theorem 6 above, we build an algorithm to reduce case graphs as follows.

***Algorithm** 7*: Construct the case graph $\Phi = (C, P)$ of the net system Σ, whose label on each edge is a single step.

If (c_1, G_1, c_2) and (c_1, G_2, c_3) are two edges of the graph, $G_1 \cap G_2 = \emptyset$ and $G = G_1 \cup G_2$ is detached then we remove 4 edges (c_1, G_1, c_2), (c_1, G_2, c_3), (c_2, G_2, c_4) and (c_3, G_1, c_4) with $c_4 = (c_1 \setminus {}^\bullet G) \cup G^\bullet$ and add a new edge (c_1, G, c_4).

If (c_1, G_1, c_2) and (c_2, G_2, c_3) are two edges of the case graph and $G = G_1 \cup G_2$ is detached then we remove 4 edges (c_1, G_1, c_2), (c_2, G_2, c_3), (c_1, G_2, c_4) and (c_4, G_1, c_3) with $c_4 = (c_1 \setminus {}^\bullet G_2) \cup G_2{}^\bullet$ and add a new edge (c_1, G, c_3).

Repeat the instructions 2) and 3) until no edges can be removed.

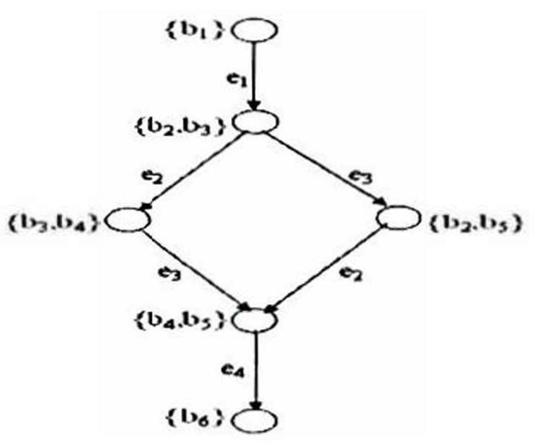
Note that, in the instruction 2) we have to check whether the sets G_1 and G_2 are disjoint, but it is no need for the instruction 3).

Sequences of labels on paths of the reduced case graph point out concurrent processes of the net system Σ with maximal coneurrent steps.

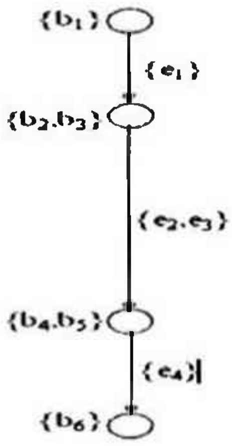
The algorithm's complexity: We can use a searching algorithm (breadth-first search or depth-first search) on the case graph to detect situations as in the instructions 2) and 3). The case graph consists of $|C|$ nodes and the degree of each node is not greater than $|E|$. Thus the complexity of the graph searching is $O(|C|.(|E|+1))$. The times of searching are not grater than $|E|$. At every node, one has to choose two its adjacent edges and check the disjoint of their labels and the detachment of the union of these labels.

Therefore, the total complexity of the algorithm is $O(|C|.|E|^4)$.

Example 11: Consider the net system Σ given in Example 8. The case graph and the reduced case graph of the net system are the following.



a) The case graph



b) The reduced case graph

From the reduced case graph of the net system Σ we get the sequence of labels on a path.

It is just the sequence of maximal concurrent steps $\{e_1\},\{e_2, e_3\},\{e_4\}$.

ones can be applied in system controls. Two first methods begin from the language generated by the given system and use an algorithm to find the normal form of traces or the "shift-left" algorithm. These algorithms have the same complexity, but the second one is much simpler. The complexity of the third algorithm is the least. It is more complicated when computing on computers.

These methods can be applied to other models of distributed systems, especially to models with dynamic environment or timed changing.

Acknowledgements. This paper is based on the talk given at the Conference on Mathematics, Mechanics, and Informatics, Hanoi, 7/10/2006, on the occasion of 50th Anniversary of Department of Mathematics, Mechanics and Informatics, Vietnam National University.

References

[1] J.I. Aalbersbeg, G. Rozenberg, Theory of Traces, *Theoretical Computer Science* 60 (1988) 1.

[2] A. Mazurkiewicz, Concurrent program schemes and their interpretations, *DAIMI Report PB-78*, Aarhus Univ., Denmark, 1977.

[3] Hoang Chi Thanh, Behavioural Synchronization of Net Systems, *IMSc Report 115*, Madras, India (1991) 136.

[4] Hoang Chi Thanh, "Shift-left" Algorithms Transforming Sequential Processes into Concurrent Ones, *Proceedings of the Second National Symposium "Fundamental and Applied Information Technology Research* (FAIR), Science & Technique Press, 2006.

[5] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1985.

[6] Hoang Chi Thanh, Vu Trong Que, Using case graphs in concurrency control on net systems *Proceedings of the 9th National Symposium on Information Technology and Communication*, 2006.