数理解析研究所講究録
511巻 1984年 120-143

120

Analysis and transformation of concurrent processes connected by streams

Kazushi Kuse 久世 和資
Masataka Sassa 佐々 政孝
Ikuo Nakata 中田 育男

University of Tsukuba

1 Introduction

Networks of concurrent processes communicating with each other are widely used in recent programming systems [Hoa][Kah][Occ]. A network of concurrent processes connected by **streams** is worth notice as a simple method for representing nontrivial problems by combining simple modules [Nak]. A stream is a possibly infinite sequence of values and is receiving attention in research on data flow languages [Den][Arv], functional programming [Bur][Hen] and logic programming [Cla]. In this note, we analyze a stream-connected network based on the theory of Petri nets [Pet]. The analysis will be based on the framework of the language designed in [Nak], but the results can be applied to more general networks connected using streams.

Problems analyzed are:

(1) Properties of the network system in itself, such as deadlock, livelock, and proper execution.

(2) Decision of minimum buffer sizes for each stream, assuming a limited resource for buffers.

(3) Formal treatment of in-line expansion algorithms, which is an interesting method of implementation realizing maximum run-time efficiency. In-line expansion is a transformation of concurrent processes into a sequential process.

As a result of this analysis, general characteristic features of networks connected by streams are shown. First, it is shown that only a limited type of deadlock can occur in such a network. Second, it is shown that no livelock (starvation) occurs in such a network. These properties seem to be a consequence of the disciplined use of concurrency by streams.

2 Streams and networks of concurrent processes connected by streams

A stream is a sequence of values of a certain fixed type. For example

<1,4,9,16,25,...>

is of the type

stream of INTEGER.

As a simple example, let us consider the problem of calculating

$U = 1^2+2^2+3^2+...+n^2$.

The structure of a program to calculate this can be best expressed by means of a stream, as shown in Fig 1. The process SQR produces the stream <1,4,9,16,...>, and the process SUM consumes it and calculates the sum of its elements. The final program is shown in Fig 2. Statements(2) and (4) put or get one element to/from a stream. The language used is designed as an extension of Pascal, and has been implemented. Multiple output and input streams are allowed for each process, and a network of concurrent processes connected by streams can be realized in general.

Termination is one of the main problems of concurrent processes communicating with each other. In [Nak], termination is dealt with by exception handling, namely the '**on** statement'. When SQR terminates first, the status of the stream becomes 'eos (end of stream)'. On the other hand , when SUM terminates first, its status becomes 'blocked'. When a get or put operation is attempted on a stream with eos or blocked status, control is

passed to the corresponding **'on** statement'. Statement(5) of Fig 2 is an example of an exception handling statement.

For a detailed explanation, see [Nak].

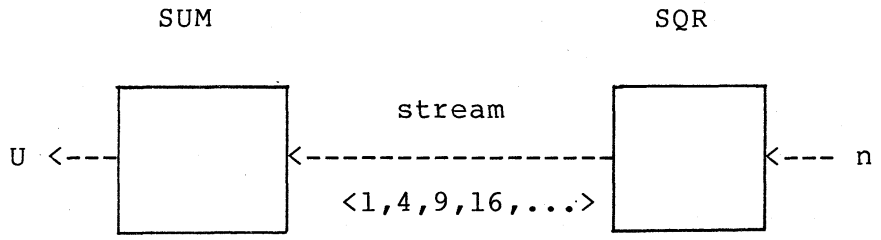


Figure 1. Module structure for the computation of $U=1^2+2^2+3^2+...+n^2$.

```
function SQR(N : INTEGER) S: stream of  INTEGER;          .....(1)
     var I: INTEGER;
     begin
          for I=:1 to N do
             next S:=I*I     {put I**I to a stream}       .....(2)
     end;

function SUM(T: stream of INTEGER) R: INTEGER;            .....(3)
     var X: INTEGER;
     begin
          X:=0;
          loop
             X:=X+next T     {get one element from the stream}..(4)
          end;
          on eos => R:=X     {return X as the function value}...(5)
     end;

  ...
U:=SUM(SQR(10));            {connect two processes by the stream}
  ...
```

Figure 2. A program for Figure 1.

The following assumption are made in our network of processes connected by streams.

Assumptions

(1) There are no isolated processes, consequently all processes must be connected by streams.

(2) Each stream has a single producer process and a single consumer process, where a producer puts elements into the stream and a consumer gets elements from it. Case(a) of Fig 3 is not allowed, but case(b) is allowed. Output of the same stream to many processes can be realized by case(b).
(3) The connection of a stream is static, and is not changed at run-time.
(4) No recursive calles of processes is allowed.
(although the implementation of [Nak] allows recursive calls)

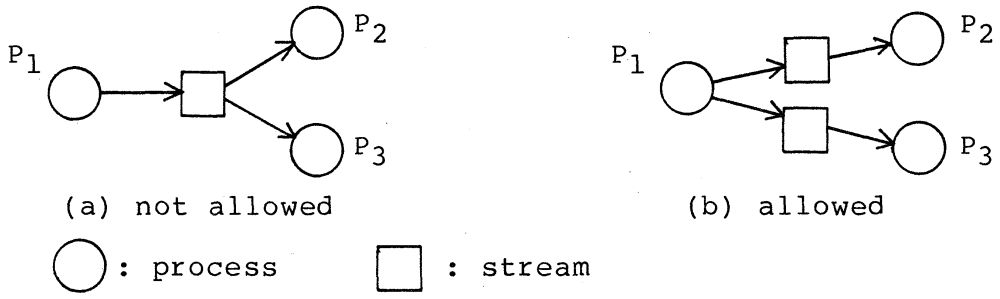


Figure 3. Assumption(2) (processes connected by streams).

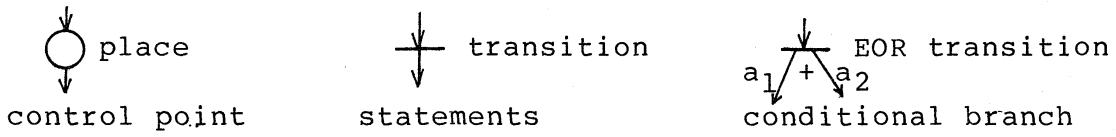
3 Modelling

3.1 Modelling by Petri Net

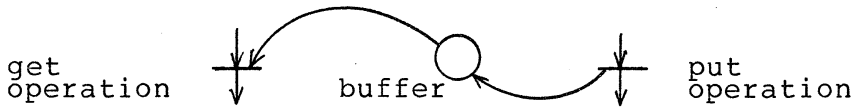
A network of processes is modelled as a Petri Net as follows.
(1) For each process, a Petri net is constructed by using transitions to represent statements and places to represent control points. More precisely, a sequence of general statements (those which are not put/get operations of streams) possibly with conditional branches or joins, but having one entry point and one exit point, is represented by a transition. Each put/get operation also is represented by a transition. A conditional branch is represented by a special EOR (exclusive or) transition [Bae]. (A token output from this EOR transition can only go through an arc.) (Fig 4(a))
(2) Transitions for put and get operations on a stream are

connected by a place modelling a buffer between them. The place may be with capacity k (for finite buffer) or without capacity constraint (for infinite buffer). This place capacity does not add essential modelling power to a Petri net, since a place of finite capacity can be represented using only places of infinite capacity for general Petri nets [Pet].(Fig 4(b))

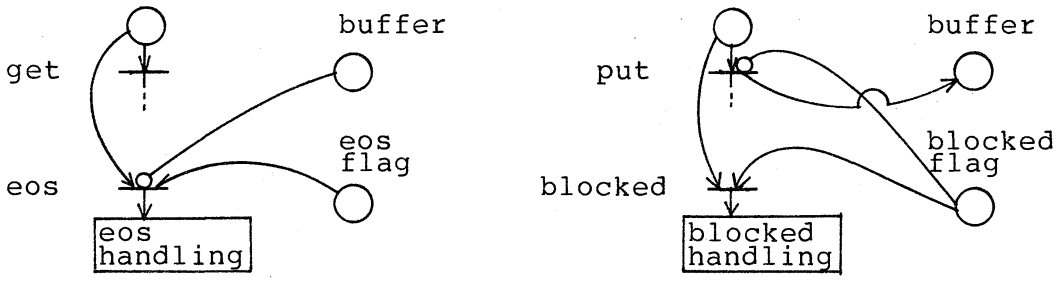
(3) Eos and blocked exception handling are modelled as in Fig 4(c). For each buffer, places with capacity 1 for an eos flag and a blocked flag are supplied. In order that eos/blocked exception handling arises only when another process terminates, transitions for put or get operations and places for buffers and flags are connected using an inhibitor arcs. If the size of the buffer is finite, the introduction of inhibitor arc does not add new modelling power to the Petri net [Age].



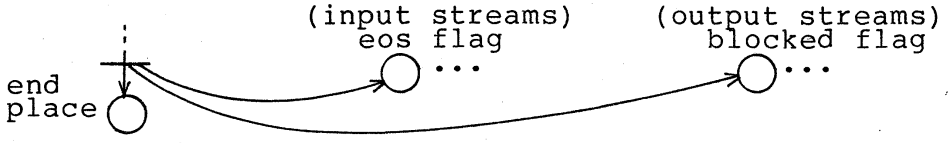
(a) representation of each process



(b) connection of put/get operation by a buffer



(c) representation of eos/blocked exception



(d) connection of end place and flag places

Figure 4. Correspondence between programs and Petri nets.

(4) End places and flag places are connected in such a way that when a process ends, a token is set in each place for eos flags of the output streams of the process and in each place for blocked flags of the input streams of the process. (Fig 4(d))

The part of the Petri net corresponding to each process (including EOR transitions representing conditional branches) is a State Transition Diagram (STD) [Pet]. Its places are 1-bounded and only one token exists for each process.

The program of Fig 2 is modelled as the Petri net in Fig 5. This Petri net is a reduced net of the original in that places and transitions corresponding to dead code are omitted. For example, the blocked flag place and its relevant transitions are omitted.

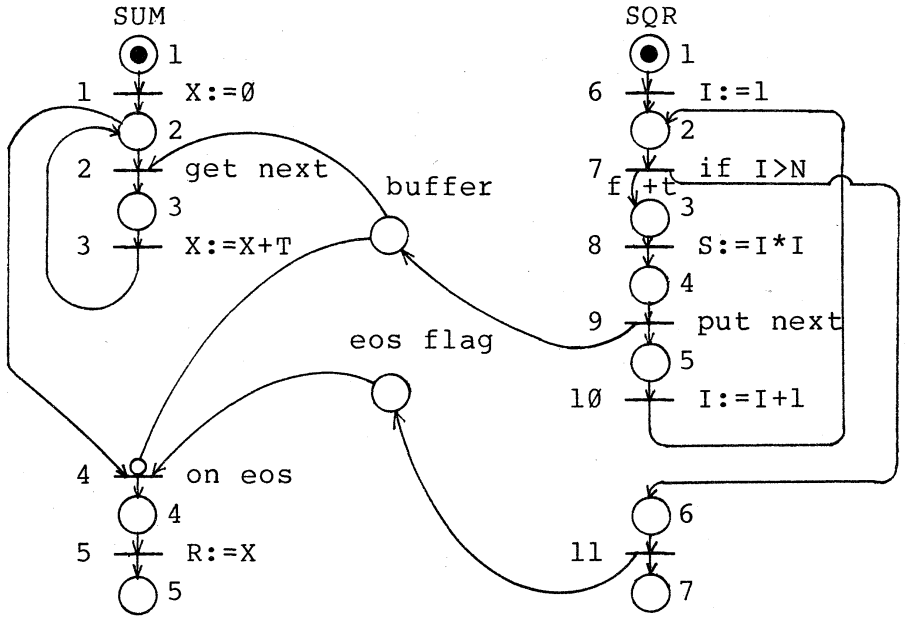


Figure 5. Petri net for the program of Figure 2.

Restriction of the firing rule

Put and get operations on the same stream buffer must be mutually excluded. This can be modelled precisely using a Petri

net. However, since the purpose of this paper is not the modelling of mutual exclusion, which is rather straightforward, we have omitted its modelling and instead have imposed the following

Firing rule: Only one transition can fire at a time.

This rule reduces the complexity of the Petri net, while maintaining the correctness of later analysis.

3.2 Marking graphs

After modelling a network of concurrent processes by a Petri net, we make a variant of a marking graph and use it to make analyses and transformations. Usually, a reachability tree [Pet] is used for analysis, where each node represents a marking of the Petri net. But often the same node may appear many times in a reachability tree. In order to improve on this, we assign a transition set to each node, which is a set of transitions from the root node of the reachability tree to that node. Then, a marking graph is made by uniting two nodes n_1,n_2 in a reachability tree when the following conditions hold:

(1) the marking of n_1 and n_2 are the same,

(2) n_1 and n_2 are in the same level (distance from the root node) in the reachability tree,

(3) n_1 and n_2 have the same transition set.

Condition(3) is for discriminating two nodes which pass through different control flows before reaching a place corresponding to a join of control (more on this in chapter 5) Conditions(2) and (3) are additional restrictions to usual marking graphs.

For simplicity, we made a simple encoding to represent nodes of a marking graph. Each process has the property that it contains a token in only one of its places. Then the marking of places of a process is represented by the place number of the place where the token resides, instead of representing the number of tokens in all places. The marking of places for buffers and

flags is represented as usual by the number of tokens in these places. Thus a marking is represented as follows.

$$(p_1,\ldots,p_n,b_1,\ldots,b_m,ef_1,\ldots ef_m,bf_1,\ldots bf_m)$$

where p_i is the control point of process i
(the location of only one token),
b_j is the number of elements in buffer j (Ø..buffer size),
ef_j is the eos flag of buffer j (Ø or 1),
bf_j is the blocked flag of buffer j (Ø or 1).

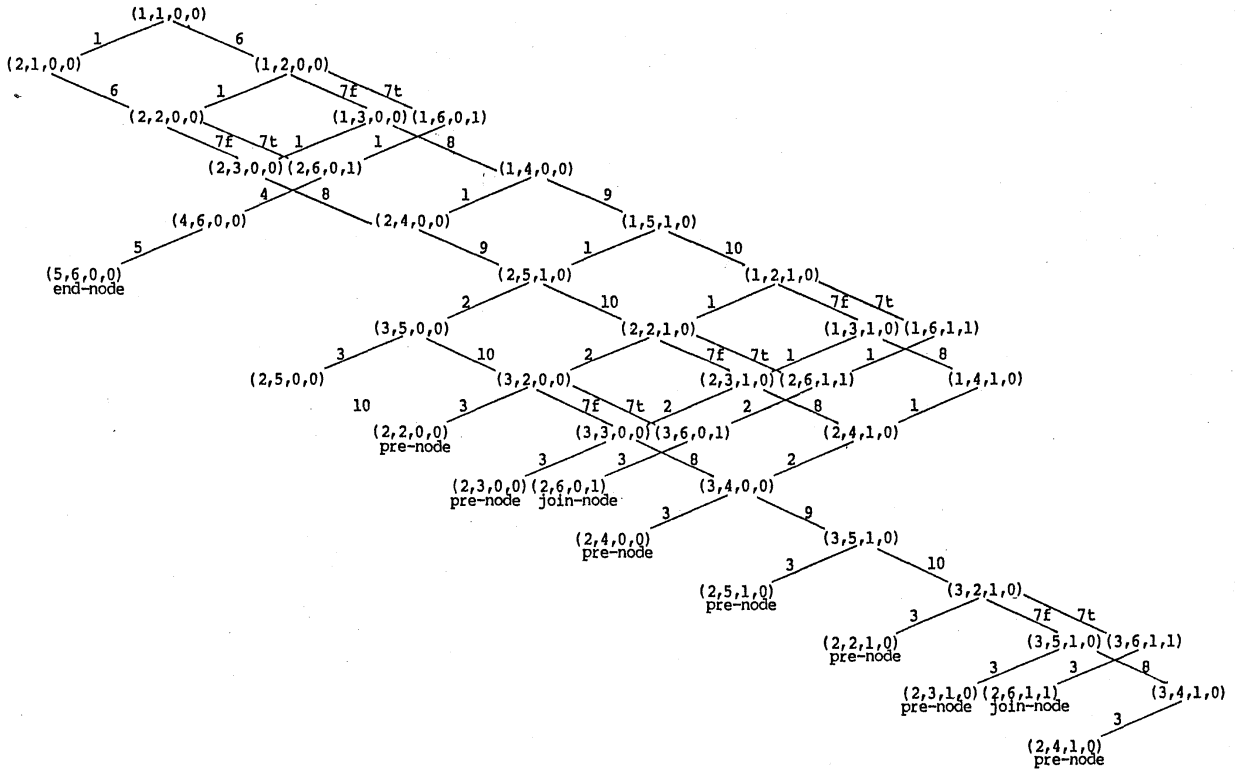
A marking graph may contain the following special type nodes.

(1) root-node : a node where all processes are at their initial state.
(2) end-node : a node where all processes are in the terminated state.(final place)
(3) pre-node : a node n_1 such that there is another node n_2 of the same marking in a lower level of the marking graph and there exists at least one path passing through n_2 from the root-node to n_1.
(4) join-node : a node n_1 such that there is another node n_2 of the same marking in a lower or equal level (already processed) of the marking graph and with no path passing through n_2 from the root-node to n_1.
(5) over-node : a node which is made by firing a node which is not enabled due to the limitation of a buffer size. The firing is made by forced increment of the buffer size.
(6) dead-node : a node not enabled. (except for over-node)

The marking graph corresponding to the Petri net of Fig 5 is shown in Fig 6.

Some transitions of the original Petri net may not appear in

the marking graph. Such transitions can be regarded as dead codes. Since P_i's, ef_j's, and bf_j's are finite, the marking graph becomes finite if all buffer b_j's are finite.



↙:transition of process SUM ↘:transition of process SQR

Figure 6. Marking graph for the Petri net of Figure 5. (buffer size=1)

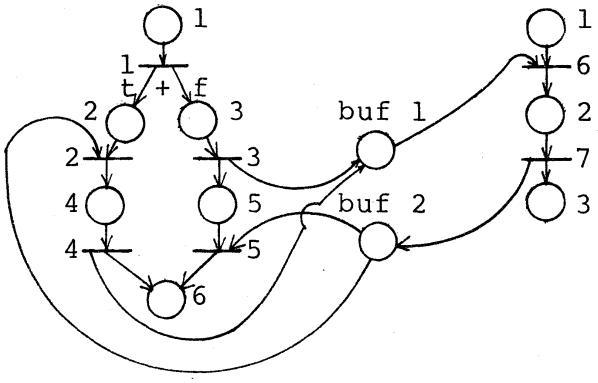
4 Analysis

4.1 Analysis of deadlock

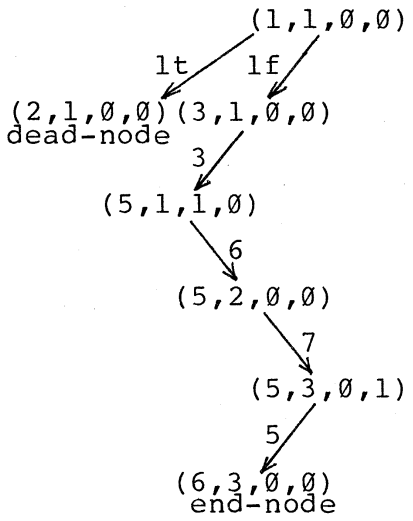
Definition

A process P_i is said to be in **deadlock** at node n (except when the control point of P_i is at the end of P_i) of a marking graph, if P_i's control point is always invariable at all nodes reachable from n. The case is described as deadlock(P_i,n).

An example of deadlock is described in Fig 7.



Petri net



marking graph

Figure 7. An example of deadlock.

Using this definition, deadlockable and deadlock-free can be defined as follows.

Definition

deadlockable(P_i,n) <==> $\exists$ n'$\in$ n-reachable, deadlock(P_i,n')
(There is a node n' reachable from n, such that P_i is in deadlock at n')

deadlock-free <==> $\forall i, \neg$ deadlockable(P_i,$n_\emptyset$),
where $n_\emptyset$ is the root-node of the marking graph.

In general, a deadlock can be classified into two types, according to how it appears in a marking graph. (Fig 8).

type 1. which occurs as a dead-node.
type 2. which has no dead-node, but occurs as a loop where the control point of a process is invariable while other processes are running.

An important property of a network of processes connected by.

streams, is that there is no deadlock of type 2.

Figure 8. Two types of deadlock in a marking graph.

Theorem 1

There is no deadlock of type 2 for a network of processes connected by streams, assuming that there is no infinite loop without put/get operations of streams.

A proof is given in the Appendix. The property that there are no isolated processes in the network (Assumption (1) of section 2) plays an essential role to this theorem.

By the theorem, only a deadlock of type 1 , i.e. dead-node, can occur, and it can be easily detected using a marking graph.

4.2 Livelock

Definition

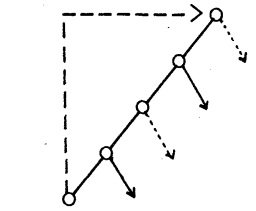
A process P_i is said to be in **livelock** (starvation) at node n (except when the control point of P_i is at the end of P_i) of a marking graph, if the following two conditions hold.

(1) There is a loop reachable from n and containing a pre-node in a marking graph, such that the control point of P_i is invariant on the loop, and the loop contains at least one node where P_i is not enabled.

(2) Not deadlock(P_i,n)

The reader is referred to [Kwo] for strict definition and classification of livelock. Livelockable and livelock-free properties are defined similarly to deadlock.

Possible situations of livelock are shown in Fig 9.



where
bold lines are enabled transitions,
dotted lines are unable transitions.

Figure 9. Livelock in a marking graph.

An important property of networks of processes connected by streams is that there is no livelock of any type.

Theorem 2

> There is no livelock of any type, for a network of processes connected by streams, assuming that there are no infinite loops without put/get operations of streams.

A proof is given in the Appendix. The property that a stream has a single producer and single consumer process (Assumption (2) of section 2), which means that there is no competition on resources, plays an essential role in the theorem.

4.3 Analysis of buffer size

Given a marking graph, necessary sizes of buffers for each stream can be analyzed.

We begin by the treatment of buffer size Ø. The situation where a get operation on a stream is executed immediately after the corresponding put operation, in other words, a value of the stream is immediately transferred from a producer process to a consumer process without buffering, can be conceived of as buffer size Ø for that stream. The situation could be modelled in a Petri net without buffer places. However, another method is used to represent the situation in our model, based on a useful property for checking whether a local transition sequence using a buffer of size >= 1 can be realized using a buffer of size Ø.

Property

If the number of token in a buffer place is limited to Ø, the get operation must occur immediately after the put operation.

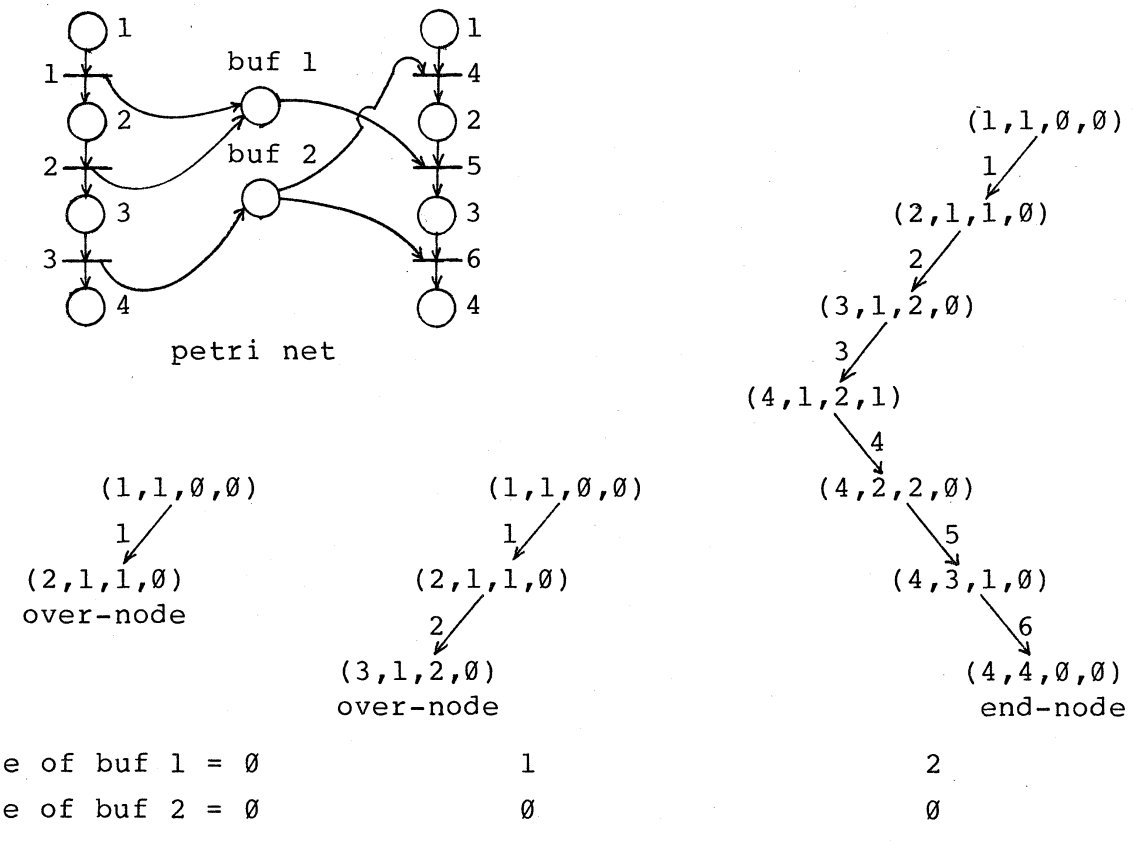


Figure 1Ø. Analysis of buffer size.

The analysis of buffer sizes proceeds as follows

Let b_i(i=1,...,n) be buffers.

Ø. For all buffers, set SIZE of b_i <- Ø

1. If b_k overflows, i.e., b_k corresponds to an over-node,
   SIZE of b_k <- SIZE of b_k + 1
2. Extend the marking graph and go to step 1.

For example in Fig 1Ø, the size of the buffer 1 which cause an over-node is incremented from Ø to 1 and eventually to 2.

4.4 Proper execution and Scheduling

Given a network of processes connected by streams and (preferably short) buffer sizes (including buffer size Ø) for each buffer, we can construct a marking graph. Owing to the livelock-freeness of the network, if the marking graph is deadlock-free and has no over-node, the network can be properly executed. In the case of single processor, it can be executed by any fair scheduler, where fairness of a scheduler can be defined strictly, for example as a 'valid' computation in [Kwo].

5 In-line expansion

A network of concurrent processes connected by streams is sometimes derived as a result of representing nontrivial problems by combining simple modules[Nak]. In this case, it is desirable that the network can be transformed into a sequential program, which can be executed efficiently.

If the marking graph of a network assures deadlock-freeness (i.e., there are no dead-nodes) and if all buffer sizes are limited, the network can be transformed into a sequential program. This is called in-line expansion. The expanded in-line code may need additional program variables for storing stream values, but no other variables nor conditional branches are

introduced to prevent simulation of a scheduler in the expanded code. If all buffer sizes are ∅, no program variables are necessary.

Several methods for in-line expansion can be conceived. One of them is the direct transformation method[Nak][Hag]. This has the lowest cost, but can not realize executable codes with maximum space efficiency. In this paper, we present a method using a Petri net which realizes optimal in-line expanded codes. Although the cost of expansion is higher than other methods, all movement of processes can be considered in advance.

We use two sets for each node i.e., a transition set(TS) which is the set of transitions fired from the root-node to the given node, and a node set(NS) which is the set of nodes that may be crossed when going from the root-node to the given node. These two sets can be computed during the construction of the marking graph.

Using TS and NS, pre-nodes and join-nodes are defined precisely as follows.

The marking graph is constructed level-wise. If a new node n_1 has the same marking as another node n_2 in the already constructed marking graph, n_1 is processed as follows.

In the case $level_{n1} > level_{n2}$,
  if $n_2 \in NS_{n1}$, make n_1 a pre-node
    and mark n_2 as a pre*-node,
  if $n_2 \notin NS_{n1}$, make n_1 a join-node
    and mark n_2 as a join*-node,
In the case $level_{n1} = level_{n2}$,
  if $TS_{n2} = TS_{n1}$, no new node is made and n_1 is identified
    with n_2,
  if $TS_{n2} \neq TS_{n1}$, make n_1 a join-node and mark n_2 as a
    join*-node.

where the check for a pre-node must precede that of a join-node

Actually, in-line expansion is derived from the marking

graph by combining paths from the root-node to pre-nodes, join-nodes and an end-node. We call this set of paths a path set. If a path terminates at a pre-node, it must include the corresponding pre*-node. If a path terminates at a join-node, some path in the path set must include that join*-node. For any conditional branch, the in-line expanded code (path set) must include both the true and the false case of the branch.

We begin by the definition of a 'branch pair' for a conditional branch. The two output arcs a_1 and a_2 in the EOR transition of Fig 4(a) are called a branch pair. For example, in the Petri net of SQR of Fig 5, the branch pair is two arcs from the 7th transition to the 3rd place and to the 6th place.

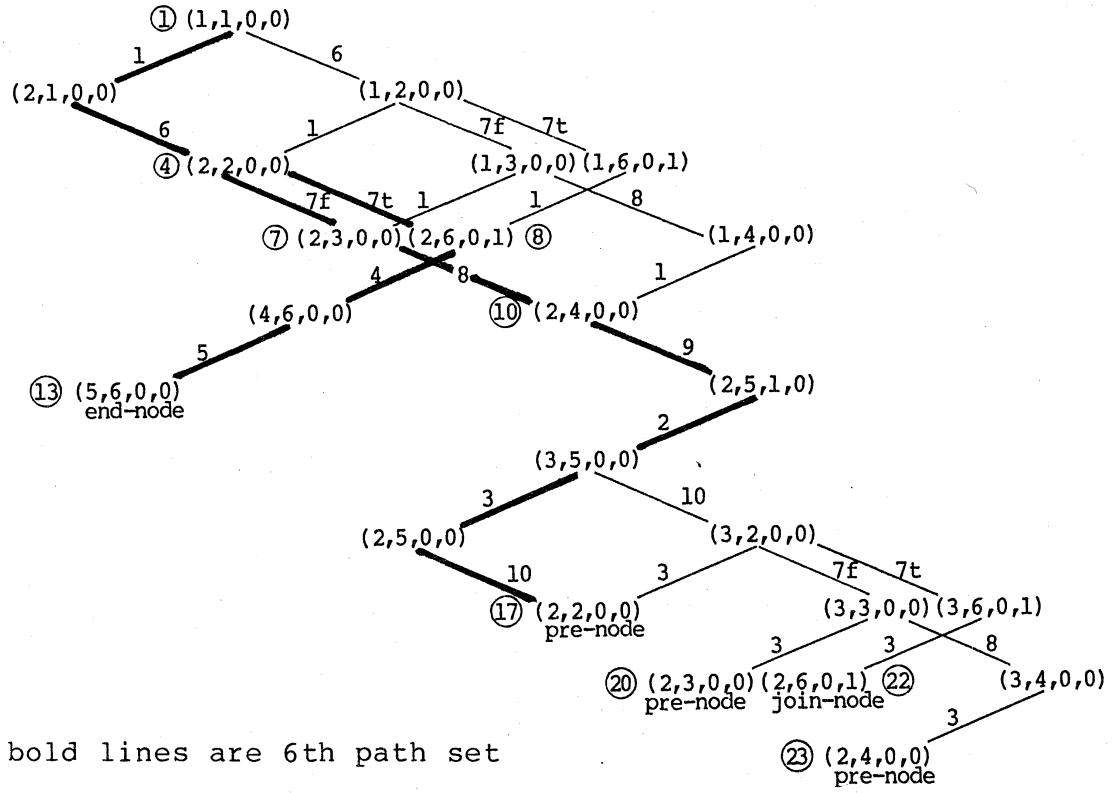


Figure 11. Marking graph for the Petri net of Figure 5. (buffer size = Ø)

The algorithm of in-line expansion using a simple example (SUM-SQR, Fig 5) can be explained as follows :

(1) For each path from the root-node to pre-, join-, or end-node n of the marking graph, pick up only branch transitions in each TS_n, which we call a BS (branch sequence) of the path.

For example, the marking graph of Fig 11 (the part of Fig 5 realizable with buffer size=Ø) has one end-node(13), three pre-nodes(17,2Ø,23), and one join-node(22). Their pre*- and join*-nodes are 4, 7, 1Ø, and 8, respectively.

path no.	terminal node no.	BS (branch sequence)	pre*	join*
1	13 (end)	(7t)	{}	{}
2	17 (pre)	(7f)	{4}	{}
3	2Ø (pre)	(7f,7f)	{7}	{}
4	22 (join)	(7f,7t)	{}	{8}
5	23 (pre)	(7f,7f)	{1Ø}	{}

(2) Combine two paths if all of their BS (branch sequences) except the last one are exactly the same and if the last branch transitions in their BS constitute a branch pair. The combined path is called a path set. The BS of the path set becomes the sequence of the branch transitions excluding the last one.

In this example, three new path sets can be made, the 6th path set from 1st and 2nd, the 7th path set from 3rd and 4th, and the 8th path set from 4th and 5th.

path set no.	terminal node set	BS	pre*	join*
6	{13,17}	()	{4}	{}
7	{2Ø,22}	(7f)	{7}	{8}
8	{22,23}	(7f)	{1Ø}	{8}

We can get further paths by combining 1st and 7th, and by combining 1st and 8th, giving a path set whose BS is empty.

path set no.	terminal node set	BS	pre*	join*
9	{13,2Ø,22}	()	{7}	{8}
1Ø	{13,22,23}	()	{1Ø}	{8}

(3) For each path set with empty BS, select concrete paths. Each concrete path must include its pre*-node if its terminal node is a pre-node. The set of concrete paths must include join*- nodes.

path set no.	terminal node set	check of including join*- nodes
9	{13,20,22}	$\{8\} \in NS_{13} \cup NS_{20} \cup NS_{22}$ O.K.
10	{13,22,23}	$\{8\} \in NS_{13} \cup NS_{22} \cup NS_{23}$ O.K.

(4) For each path set, calculate cost(code length). If costs of all transitions are approximated by the same value, it can be calculated by using the level information about each node. In combining two path sets, the cost is computed by adding levels of the terminal node of each path set and subtracting the level of the node with the longest common part of the two path sets.

path set no. set no.	calculation of cost
6	$level_{13} + level_{17} - common_{13,17}$ $= 5 + 8 - 2 = 11$
9	$level_{13} + level_{20} - common_{13,20}$ $+ level_{22} - common_{(13,20),22}$ $= 5 + 9 - 2 + 9 - 7 = 14$
10	$level_{13} + level_{22} - common_{13,22}$ $+ level_{23} - common_{(13,22),23}$ $= 5 + 9 - 2 + 10 - 7 = 15$

The above is the space cost and corresponds to the textual length of the expanded code.

(5) Select the path set with smallest cost. It is the optimal in-line expansion.

In this example, since 11 < 14 < 15 , the 6th path set is optimal (bold lines in Fig 11). The program made from the 6th path set is shown in Fig 12.

```
      X := Ø;
      I := 1;
L1  :if I > N then goto L2;
      S := I * I;
      T := S;        {put/get operation}
      X := X + T;
      I := I + 1;
      goto L1;
L2  :R := X;
```

Figure 12. In-line expansion for the program of Figure 2.

Note that the time cost or the running time of the expanded code is approximately the same for any path set satisfying step (3) as given above, except for the possibly useless code immediately before the exception handling. This can be easily shown from the nature of the marking graph.

Note also that in some cases, the code length of expanded code may become much larger than the sum of original code lengthes. An example is the network of processes in Fig 13.

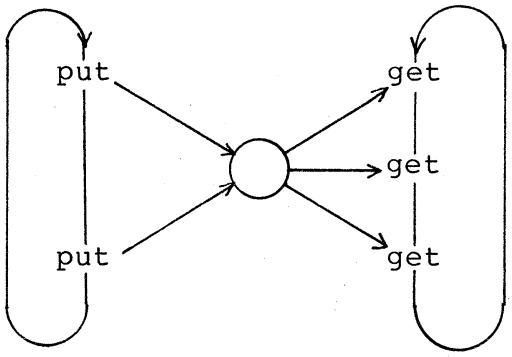


Figure 13. A network whose expanded code is long.

6 Concluding remarks

We have analyzed a network of concurrent processes connected by streams using a Petri net. In the absence of infinite loops without put/get operations of streams, there is no livelock and no deadlock of type 2. The result demonstrates an important characteristic of networks connected by streams.

Other features of our analysis as compared with general concurrent processes, are the handling of process termination such as eos and blocked exception, and analysis of buffer size.

Our analysis does not consider program semantics, i.e., both branches in each conditional branch are assumed to have the possibility of being executed. Thus, some analysis, e.g., decision of deadlock and buffer size may give worse results than what occurs in reality. This often indicates a need for an infinite buffer size, even if this need can never arise in practice. This is a limitation of analysis without program semantics. But, in many cases the user will be able to get more useful properties by adding other information concerning behavior of the program.

In-line expansion using Petri nets presented in this note is not of low cost, but by this strategy we can get an optimal in-line expansion. The trade-off is the same as in optimizing compilers. We believe that our method is an interesting one that provides executable code with maximum efficiency.

These analysis and transformation can be realized automatically.

References

[Age] Agerwala,T.: Comments on capabilities, limitations and "correctness" of Petri nets, Proc. 1st. Ann. Symp. Computer Architecture, ACM, pp.81-86 (1973).

[Arv] Arvind,Streams and Managers,in Lecture Notes in Computer Science,Vol.143,PP.452-465.

[Bae] Baer,J.L.: Techniques to exploit parallelism, in Evans,D.J. (ed.),Parallel Processing Systems, pp.75-98.

[Bur] Burge,W.H: Recursive programming techniques, Addison-Wesley, 1975.

[Cla] Clark, K.L. and Gregory, S: A Relational Language for Parallel Programming, Proc. of the 1981 Conf. on Functional Programming Languages and Computer

Architecture,ppl71-178,(Oct.1981).

[Den] Dennis, J.B. and Weng, K.K.-S.: An Abstract Implementation for Concurrent Computation with Streams, Proc. 1979 Int.Conf.on Parallel Processing, pp.35-45,1979

[Hag] Hagino,T., Proofs of Communicating Sequential Processes, Preprint of WG on Fundamental Theories of Software of IPSJ, (in Japanese) (Oct.1982)

[Hen] Henderson,P.: Purely Functional Operating Systems ,in Darlington et al.(eds.), Functional Programming and its Applications ,Cambridge Univ. Press, (1982).

[Hoa] Hoare,C.A.R.: Communicating sequential processes, Comm.ACM, Vol.21, No.8, pp.666-677 (Aug. 1978).

[Kah] Kahn,G. and MacQueen,D.B.: Coroutines and networks of parallel processes, Information Processing 77, pp.993-998, North-Holland, 1977.

[Kwo] Kwong,Y.S.: On the absence of livelocks in parallel programs, in Lecture Notes in Computer Science Vol.70,pp.172-190.

[Nak] Nakata,I. and Sassa,M.: Programming with streams, IBM Research Reports RJ3751(43317) (Jan. 1983).

[Occ] OCCAM Programming manual, INMOS Limited (1982)

[Pet] Peterson,J.L.: Petri net theory and the modeling of systems, Prentice-Hall, 1981, or, Peterson,J.L.: Petri nets, Computing Surveys, Vol.9, No.3, pp.223-252 (Sep. 1977).

Appendix

Proof of theorem 1 and 2

(There is no type 2 deadlock or livelock.)

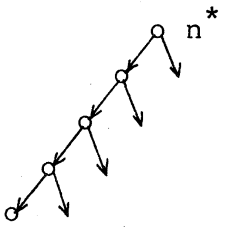
Assumption A

There is no infinite loop without a put/get operation. That is, a loop must contain a put/get operation or it must be exited at some time.

Proof

We first note that type 2 deadlock or livelock arises as a loop in a marking graph. A loop in a marking graph can be classified into three classes as follows.

(1)

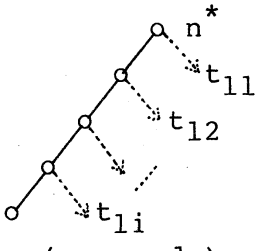


n (pre-node)

↙:transition of process P_2 ↘: that of process P_1

Solid lines represent enabled transitions, and dotted lines represent disabled (not enabled) transitions.

(2)

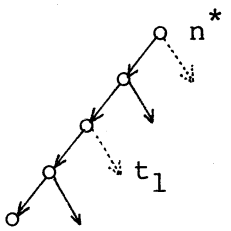


n (pre-node)

Transitions of P_1 are disabled at all nodes.

P_1 is in deadlock of type 2.

(3)



n (pre-node)

There are some nodes where transitions of P_1 are disabled.

P_1 is in livelock.

The proof is made according to the above classification.

(1) This situation often appears in a marking graph. It has no deadlock and no livelock.

(2) P_1 is in type 2 deadlock. In networks connected with streams, a disabled transition occurs only at put/get operations of streams as in the following two cases.

case (i) Transition t_1 of P_1 is a get operation. The t_1 is disabled because the associated buffer is empty.

case (ii) Transition t_1 of P_1 is a put operation. This case arises only if the size of the associated buffer is finite. The t_1 is disabled because this buffer is full.

In general, a type 2 deadlock may involve three or more processes. But since the situation arose due to a put/get operation on a stream, we can reduce it to the situation involving only two processes without loss of generality. This is made by letting P_2 be the process connected to P_1 by the relevant stream. The existence of the process P_2 is assured by assumption(1) of section 2.

From assumption A , either of the following holds.

(A1) The infinite loop is exited at some time.

(A2) A get or put operation will be performed within the loop.

In case (A1), no deadlock occurs. In case (A2), process P_2 will make t_1 enabled as follows.

In case (i), P_2's loop must include some put operation. After the execution of this operation, the buffer has a token, and the get transition of P_1 becomes enabled.

In case (ii) , P_2's loop must include some get operation. After the execution of this operation , the buffer becomes not full, and the put transition of P_1 becomes enabled.

In both cases, a transition of P_1 is enabled, which is not a deadlock.

(3) P_1 is in livelock. A proof similar to type 2 deadlock can be followed.

Especially, case(A2) is as follows.

In case(i), after a transition of P_1 becomes enabled, it is always enabled until it is executed. From assumption(2) of section 2, no process except P_1 can get from the buffer.

Case(ii) is similar.

Thus, all arcs t_{11} ,...,t_{1i} of P_1 become enabled, which is the situation of class(1) without livelock.

Note

In [Nak], a class of streams called passive streams is introduced as an extension to the usual stream connection. Livelocks may occur in the presence of passive streams, since an element in passive streams are subject to resource sharing.