Modelling and Analysis of Concurrent Processes Connected by Streams

KAZUSHI KUSE*, MASATAKA SASSA** and IKUO NAKATA**

A network of concurrent processes connected by streams is worth notice as a simple method for representing nontrivial problems by combining simple modules. We define such a network as a subclass of general communicating concurrent processes. A network in this class has some restrictions, for example, a stream must have only one producer and one consumer. However this is a practical class which can be used to represent many problems, and some representative classes of concurrent processes such as Hoare's CSP belong to this class. In this paper, we formulate the analysis of networks in this class based on the theory of Petri nets. We have clarified some general characteristic features of this class, a major one being that the class is livelock-free. The analysis includes the detection of deadlocks, dead code and the possibility of termination, along with the determination of a necessary buffer size for each stream. Here, we applied the analysis to networks described by the programming language *Stella*, which we developed, but the analysis is independent of specific languages. In order to achieve the above analysis, we implemented an automatic analysis system called *SPRAT* on a VAX-11/750. We successfully analyzed many networks which belong to this class using SPRAT.

1. Introduction

Networks of concurrent processes communicating with each other are widely used in recent programming systems [Hoa] [Kah] [Occ]. Among these, a network of concurrent processes connected by *streams* is worth notice as a simple method for representing nontrivial problems by combining simple modules. 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 paper, we use the theory of Petri nets [Pet] to analyze a stream-connected network. This analysis can be applied to CSP [Hoa] and to more general networks connected with streams. Here, we applied the analysis to networks described by *Stella* which is a language we designed and implemented for programming with streams [Nak]. In Stella, each stream must have only one producer process and only one consumer process. Such networks can be regarded as a class of communicating concurrent processes. We found that this class is practical in that many problems can be represented by it and it has some good features which will be described later. Yet it is easy to analyze as a result of the formalization which will be given in this paper.

Our analysis determines properties of the network system, such as detection of deadlocks, livelock (starvation), dead code and the possibility of terminating. An interesting subject is the determination of the minimum buffer size for each stream, assuming a limited resource for buffers.

This analysis also relates general properties of the class. By investigating characteristics of the Petri net which models the above class's network, general characteristic features of such a network are clarified. One major discovery was that no livelock occurs in this class's network. The attractive properties of this class seem to be a consequence of the restriction that each stream has only one producer process and only one consumer process.

In order to facilitate the analysis, we implemented an automatic analysis system called *SPRAT* (Stream-connected PRocesses Analyzer and Transformer) on a VAX-11/750. We successfully analyzed many networks of stream-connected processes using this system and obtained useful information on them. Moreover, at the same time, SPRAT can perform an in-line expansion. An in-line expansion is transformation of concurrent processes into a sequential process and is an interesting method of implementation realizing maximum run-time efficiency in a sequential machine. The details of in-line expansion will be given in a separate paper.

Section 2 describes streams in the framework of Stella in more detail and gives a definition of the class of stream-connected processes which we deal with. Section 3 describes modelling with Petri nets and introduces the marking graph which is obtained from a Petri net. Section 4 proceeds to describe the analysis and characteristic features of stream-connected processes such as deadlocks and livelocks. The algorithm for

*Doctoral Program in Engineering, University of Tsukuba
**Institute of Information Science and Electronics, University of Tsukuba

minimum buffer size analysis is also given. Section 5 describes implementation of the automatic analysis system, SPRAT. Section 6 presents concluding remarks. An earlier version of this paper appeared in [Kus].

2. Streams and Concurrent Processes Connected by Streams

Here we describe streams in the framework of Stella. A stream is a sequence of values of a certain fixed type. For example

$$\langle 1, 4, 9, 16, 25, \ldots \rangle$$

is of the type stream of integer.

A network of concurrent processes is represented by a group of processes and the streams which connect these processes. The processes can concurrently run without strict synchronization since each stream can store some data. When the stream flows from a producer process it is called an *output stream*, and when the stream flows to a consumer process it is called an *input stream*. Multiple output and input streams are allowed for each process, and in general a network of concurrent processes connected by streams can be realized. Each stream must have only one producer process and only one consumer process. A member of such a class of concurrent processes is called an *SCCP* (Stream-Connected Concurrent Processes) and is defined as follows.

Def 2.1

An SCCP is a network of communicating concurrent processes which satisfies the following conditions.

(1) Processes communicate with each other only with streams.

(2) A stream has only one producer process and only one consumer process.

To illustrate this, case (a) of Fig. 1 is not allowed, while case (b) is allowed. Output of the same stream to many processes can be realized by case (b).

(3) A connection of streams cannot be dynamically changed.

(4) A process cannot be dynamically created.

The SCCP is practical in that many problems can be represented by it and most communicating concurrent processes belong to this class. In this paper, the formalization is described with Petri nets.

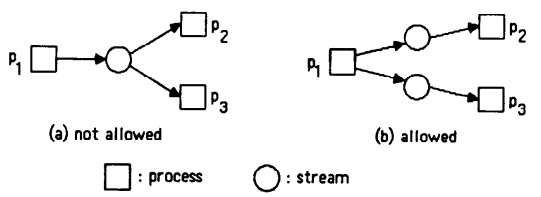


Fig. 1 A restriction on SCCPs.

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

$$1^2+2^2+3^2+\ldots+n^2.$$

The structure of a program to calculate this can be best expressed by means of a stream, as shown in Fig. 2. The process "sqr" produces the stream $\langle 1, 4, 9, 16, \ldots \rangle$, and the process "sum" consumes it and calculates the sum of its elements.

Stella was designed and implemented by us as a language for representing SCCPs [Nak]. The language is designed as an extension of Pascal. Stella is introduced here only to explain SCCPs. The modelling and analysis in this paper are independent of the language used. For example, a representation with LISP or FP can be analyzed with the same method.

The representation of the module structure of Fig. 2 by Stella is shown in Fig. 3. Put and get operations are represented by assignment statements with **next**. Statement (2) puts one element to a stream, and statement (4) gets one element from the stream.

Termination is one of the main problems of concurrent processes communicating with each other. In SCCP, termination is dealt with by exception handling. When the producer "sqr" terminates first and the generated stream is exhausted, the status of the stream becomes *eos* (end of stream). On the other hand, when the consumer "sum" terminates first, the status of the stream becomes *blocked* (blocked stream). When a get or put operation is attempted on a stream with eos or blocked status, control is passed to the corresponding exception handling statement. In Stella, the exception handling statement is enclosed with "《" and "》", and attached after the put/get operation.

Statement (5) of Fig. 3 is an example of exception

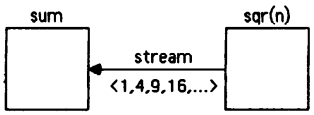


Fig. 2 Module structure for the computation of $1^2+2^2+3^2+, \ldots, +n^2$.

```
module sqr(n: integer) outs: stream of integer; .....(1)
    var i: integer;
begin
    for i := 1 to n do
        next outs := i * i      {put i*i to a stream} .....(2)
end;

module sum(ins: stream of integer); .....(3)
    var x: integer;
begin
    x := 0;
    loop
        x := x + next ins       {get one element from a stream} .....(4)
          << writeln(x) >>      {write x as the result value} .....(5)
    end
end;

...
sum( sqr( 10 ) );       {connect two processes by a stream}
...
```

Fig. 3 A program for Fig. 2.

handling statements. In order to change the flow of control, one of the following statements can be used at the end of the exception handling statements: an **exit** statement which passes control to the outside of the innermost loop including this statement, a **terminate** statement which terminates the relevant process, or a **goto** statement which is used to pass control to another point. The **terminate** statement is assumed as a default when none of these specific statements appear at the end of an exception handling statement. For example, statement (5) is equivalent to "《writeln (X); **terminate**》". When the entire exception handling statement is omitted, "《**terminate**》" is assumed.

3. Modelling

3.1 Modelling with Petri Nets

A network of processes (in the SCCP class) is modelled as a Petri net as follows (Fig. 4).

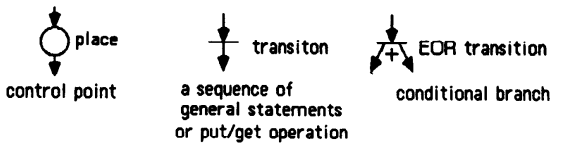
(1) For each process, a Petri net is constructed by using *transitions* to represent statements and *places* to represent control points (Fig. 4(a)). More precisely, a sequence of general statements not including put/get operations of streams is represented by a transition. Each put/get operation is also represented by a transition. A conditional branch where one side or both sides include at least one put/get operation is represented by a special *EOR* (exclusive or) transition [Bae]. A token output from this EOR transition can go through only one arc. Transitions for general statements are necessary for in-line expansion, but they can be omitted for other analyses such as detection of deadlocks.

(2) Transitions for put and get operations on a stream are connected by a place modelling a buffer between them (Fig. 4(b)). The place is of capacity k for a finite buffer of size k.

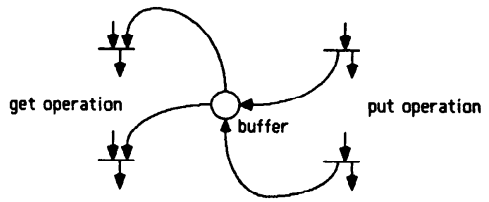
(3) Eos and blocked exception handling are modelled as in Fig. 4(c). For each buffer, two places with a capacity of 1 for an eos flag and a blocked flag are supplied. In order that eos or blocked exception handling arises only when a process connected to the relevant stream terminates, transitions for put or get operations and places for buffers and flags are connected using *inhibitor arcs*. The inhibitor arc enables a transition to fire only if the input place of this arc is empty. Therefore, eos transition occurs only when the buffer place is empty and the eos flag is set. The blocked transition occurs only when the blocked flag is set. Note also that, only one of the transitions for get and eos or put and blocked occurs.

(4) The transitions preceding end places are connected to flag places 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))

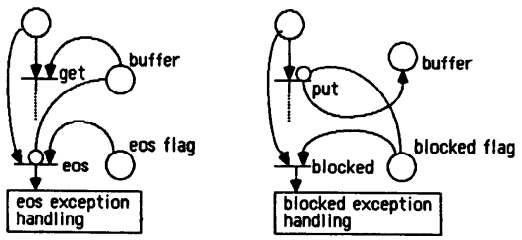
We introduce place capacity [Mat] and inhibitor arcs



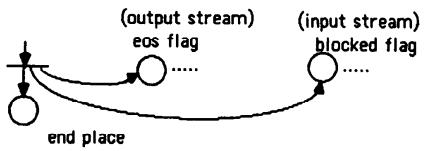
(a) representation of each process



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



(c) representation of an eos/blocked exception



(d) connection of an end place and flag places

Fig. 4 Correspondence between programs and Petri nets.

[Age] to aid in the representation, but their introduction does not change the modelling power of the Petri net [Pet].

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 a Petri net in Fig. 5.

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

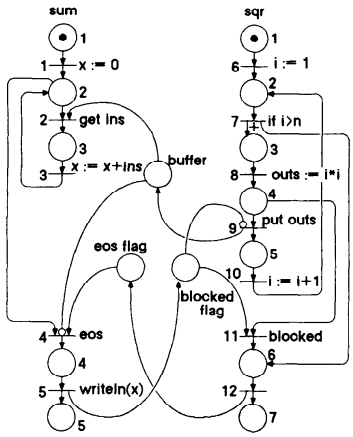


Fig. 5 Petri net for the program of Fig. 3.

firing rule:

Firing rule—Only one transition and fire at a time.

This rule reduces the complexity of the Petri net, while maintaining the generality and the correctness of later analyses.

3.2 Marking Graphs

Usually, a *reachability tree* [Pet] is used for analysis of a Petri net. The reachability tree represents the reachability set of a Petri net, where each node represents a marking of the Petri net. But we use a variant of the reachability tree called a *marking graph*. It has two distinguishing features as explained in the following.

First, we introduce a simple encoding to represent a marking, i.e. node of a marking graph, as follows. 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 giving the number of tokens in all places. The marking of places for buffers and flags is shown as usual by the number of tokens in these places. Thus a marking for a network of n processes connected by m streams is represented as

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

where p_i is the control point of process i
(the location of the single token for the process),
b_j is the number of elements in buffer j,
ef_j is 0 or 1 as the eos flag of buffer j,
bf_j is 0 or 1 as the blocked flag of buffer j.

Secondly and more important, a reachability set is represented not by a tree, but by a graph, for the same nodes often appear many times in a reachability tree. Especially in a Petri net which models concurrent processes, many nodes are duplicated because of their concurrency. In order to improve on this, we unite two nodes if certain conditions hold. To do this, we assign a *transition set* to each node. A transition set for a given node is a set of transitions from the root node of the reachability tree to that node. 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 on the same level (distance from the root node) in the reachability tree,

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

This means that the nodes which are made by nondeterministic execution join at the same point.

A marking graph may contain the following special terminal nodes. These are introduced for an analysis which will be presented later.

(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 (end place).

(3) *pre-node*: a node n_1 such that there is another node n_2 of the same marking in the same or lower level of the marking graph.

(4) *full-node*: a node which is not enabled due to the limitation of a buffer size.

(5) *dead-node*: a node not enabled. (except for full-nodes)

A marking graph is made with a fixed capacity for each buffers. As a special case, we introduce the concept of buffer size (or capacity) 0. The situation where a get operation on a stream is executed immediately after the corresponding put operation, in other words, the value of an element of the stream is immediately transferred from a producer process to a consumer process without buffering, can be conceived of as buffer size 0 for that stream. The situation could be modelled in a Petri net using a single transition without using buffer places. But, for generality, it is modelled in the same way as for buffers with buffer size >0, adding the following convention.

Convention

If the number of tokens in a buffer place becomes 0 immediately after it becomes 1, the buffer size is assumed to be 0.

A marking graph corresponding to the Petri net of Fig. 5 is shown in Fig. 6 (The capacity of the buffer is set to 0). In Fig. 6, the two arcs from the root-node mean that transition 1 and 6 are enabled. If transition 6 fires, the marking changes to (1, 2, 0, 0, 0). Since transition 7 is a conditional branch, the corresponding arc is split into two arcs. In time, when transition 9 corresponding to the put-operation fires, the number of tokens in the buffer changes from 0 to 1. Immediately after that, when transition 2 corresponding to the get-operation fires, the number changes from 1 to 0.

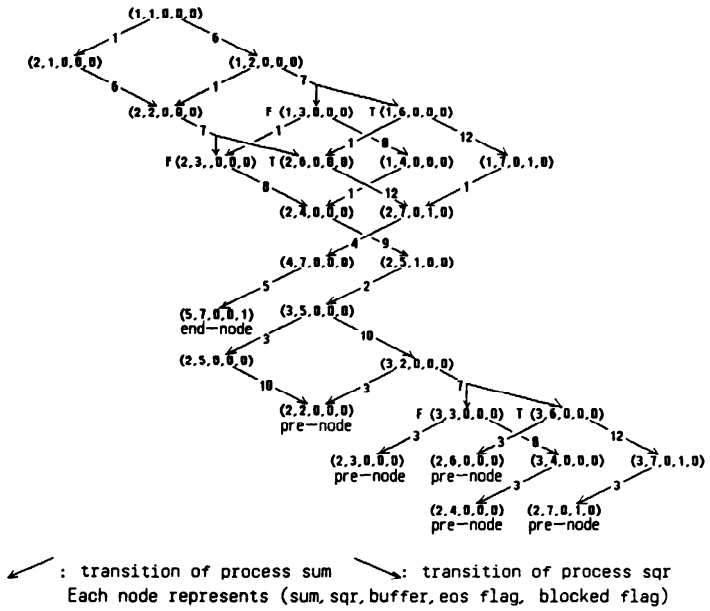


Fig. 6 Marking graph for the Petri net of Fig. 5.
(buffer size=0, cf. section 3.2)

Two paths from the root-node to the node 2-levels lower (2, 2, 0, 0, 0) represent nondeterministic execution of concurrent processes. The two occurrences of node (2, 2, 0, 0, 0) are united since the conditions for uniting nodes are satisfied (both transition sets are {1, 6}).

In general, the root-node and the end-node represent a start point and a stop point of the execution of the network, and a pre-node represents either the loop structure or a join of control points for a conditional branch. In Fig. 6, pre-node (2, 2, 0, 0, 0) corresponds to the loop (transitions 2-3 and 7-8-9-10) of Fig. 5.

Often the existence of loops in a program results in an infinite number of tokens in a buffer place and this in effect produces an infinite buffer. We represent such a case by using a special symbol, ω(infinity), which represents a number of tokens which can be made arbitrarily large [Pet]. The detail is described in section 4.3.

Since the cardinalities of $p_i's$ and the number of tokens in finite buffers $b_j's$, $ef_j's$ and $bf_j's$ are finite, and infinite buffers are represented by ω, the marking graph becomes finite.

4. Analysis

In this section, we present as general properties of SCCP, deadlock, livelock, buffer size, termination and dead code. We describe also analysis techniques for each of them.

4.1 Deadlock

In this section, we give the definition of deadlock in general concurrent processes, describe characteristics of SCCP deadlock, and explain the detection of SCCP deadlocks. First, we recall the general definition of deadlock on marking graphs according to the definitions for a Petri net in [Kwo].

Def 4.1

A reachable set for n is the set of all nodes reachable [Pet] from n and is represented by $\boldsymbol{R}(n)$. ($\boldsymbol{R}(n)$ includes n and also includes nodes in both the true part and the false part of EOR transitions.)

Def 4.2

A process p_i is said to be dead at node n of a marking graph and is represented by $dead(p_i, n)$ iff the following holds.

$dead(p_i, n) \Leftrightarrow p_i's$ control point at n is not an end place, and for all $n' \in \boldsymbol{R}(n)$, $p_i's$ control point at n' is the same as $p_i's$ control point at n.

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

Def 4.3

$deadlockable(p_i, n) \Leftrightarrow {}^{\exists}n' \in \boldsymbol{R}(n)$, $dead(p_i, n')$.

Def 4.4

A network is said to be deadlock-free iff the following

holds.
deadlock-free $\Leftrightarrow {}^{\forall}i$, not deadlockable($p_i$, root-node)

The deadlock can be classified into two types.

Def 4.5
Type 1 deadlock $\Leftrightarrow {}^{\forall}i$, dead($p_i$, n).
Type 2 deadlock $\Leftrightarrow {}^{\exists}i$, dead($p_i$, n)
and ${}^{\exists}j$, not dead(p_j, n).

In the marking graph, type 1 deadlock appears as a terminal node(dead-node), but a type 2 deadlock does not appear as a terminal node. Therefore, type 1 deadlock is easy to detect, but type 2 deadlock is difficult to detect in general concurrent processes since the reachable set must be calculated according to the definition of deadlocks. The time complexity of this detection is $O(N^2)$ where N is the number of nodes in the marking graph. This is because the computation of the reachable set, which is of order $O(N)$, must be done at each node. In SCCP, however, a type 2 deadlock is also easy to detect because of a characteristic property of SCCP, as follows.

A deadlock of SCCP occurs when some processes want to get streams (resources) which are to be put by other processes but all other processes are unable to put to the stream. The deadlock can be determined by the relation between the consumer and the producer since the connection of streams is fixed (Def 2.1 (2) and (3)). Strictly speaking, consider a relation between a consumer process whose control point is before a get operation from an empty stream and the corresponding producer process. If the set of these relations forms a cycle, these processes are in deadlock. For example, in Fig. 7, the connections of process f and process h, g and f, and h and g form a cycle.

Thus, type 2 deadlocks can be detected by examining the relation between the consumer and the producer at the time the marking graph is generated. The algorithm follows.

Algorithm 4.1
Introduce a consumer/producer relation called a *wait relation*, which is represented by (p_i, p_j), and which indicates that the control point of process p_i is before a get operation from an empty stream which is to be put to by process p_j.

In constructing the marking graph, each time a new node n is made by firing a transition t on the marking graph, apply the following rules.
1. If node n is a dead-node, type 1 deadlock occurs.
2. At node n, if process p_i's control point is before a get operation from an empty stream which is to be put to by process p_j, add relation (p_i, p_j) to the set of wait relations. If a cycle exists in the wait relations, type 2 deadlock occurs.
3. If transition t was a put-operation of process p_j and process p_i's waiting stream became not empty by the put operation, delete (p_i, p_j) from the set of wait relations.

Examples of deadlock are shown in Fig. 8 and Fig. 9. In the marking graph of Fig. 8, a type 1 deadlock occurs since a dead node appears. In the marking graph of Fig. 9, a type 2 deadlock occurs since (p_1, p_2) and (p_2, p_1) in the set of wait relations form a cycle in the second node.

By the above algorithm, it was shown that the detection of deadlocks of SCCP is much easier than that for general concurrent processes.

4.2 Livelock

If some process cannot run forever due to the scheduler strategy, they are said to be in *livelock* or *in-*

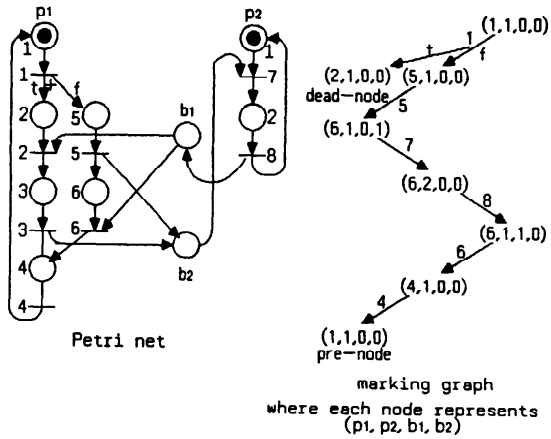


Fig. 8 An example of a type 1 deadlock.

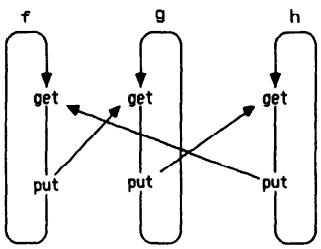


Fig. 7 An SCCP deadlock.

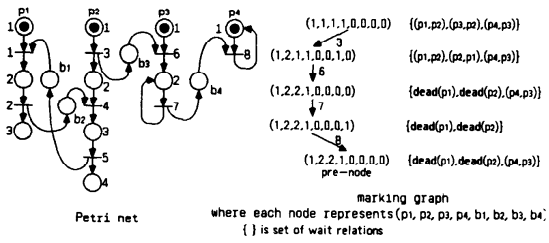


Fig. 9 An example of a type 2 deadlock.

dividual starvation [Ash]. A famous example is the possibility of livelock in the Dining Philosopher Problem [Dij].

In general concurrent processes, the existence or non-existence of livelocks is a complicated problem. We will show an important property of SCCP, which is that there is no livelock for SCCP. This can be explained by using Petri nets and marking graphs as follows.

First, the definition of livelocks is presented as a refinement of the definition in [Kwo].

Def 4.6

A process p_i is said to be *starvable* at node n and represented by *starvable*(p_i, n) iff the following holds.

starvable(p_i, n) $\Leftrightarrow$ p_i's control point at n is not an end place, and there is a loop in a marking graph, such that the control point of p_i is invariable on the loop, and the loop contains at least one node where p_i is disabled and at least one node where p_i is enabled.

Using this definition, "livelockable" and "livelock-free" can be defined as follows similarly to deadlockable and deadlock-free.

Def 4.7

A process p_i is said to be livelockable at node n and represented by *livelockable*(p_i, n) iff the following holds.

livelockable(p_i, n) $\Leftrightarrow$ $^{\exists}n' \in \boldsymbol{R}(n)$, *starvable*($p_i$, n')

Def 4.8

A network is said to be *livelock-free* iff the following holds.

livelock-free $\Leftrightarrow$ $^{\forall}i$, not *livelockable*(p_i, root-node)

The reader is referred to [Kwo] for a strict definition and classification of livelocks.

A possible livelock situation is illustrated in Fig. 10. In Fig. 10, process p_2 is in livelock since the marking graph includes a loop, p_2's control point is invariable, and there exist some nodes where transition of p_2 are disabled.

An important property of SCCP is that it is livelock-free.

Assumption

Each buffer is infinite.

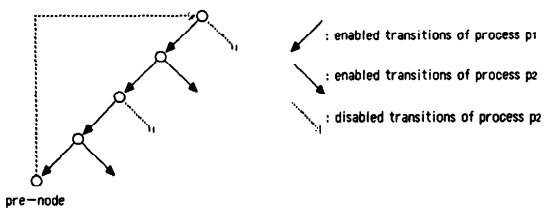


Fig. 10 Livelock in a marking graph.

Theorem

SCCP is *livelock-free*.

Proof

We first note that in SCCP, a disabled transition occurs only at a get operation of a stream. The transition corresponding to a get operation is disabled when the associated buffer is empty. Since it is assumed that each buffer is infinite, the transitions corresponding to put operations are always enabled.

Livelock arises as a loop in a marking graph as shown in Fig. 10. Let us suppose that process p_2 is in livelock. Loops where p_2's control point is invariable (Since our interest is the livelock, we consider only the status where a process is not running) in the marking graphs are classified into three types as shown in Fig. 11.

Type (b) is a livelock status. In type (a), since there are no disabled transitions both processes can always run and there is neither livelock nor deadlock. In type (b), the bold lines and the dotted lines of process p_2 should both be associated with one transition corresponding to a get operation by the above note. The dotted lines represent the fact that a get operation from an empty buffer is impossible. Transition t_1 is a put operation which puts data into the buffer since after transition t_1 fires, transition t_2 becomes enabled.

In type (c), process p_2 never runs since the associated buffer is empty. This type represents a deadlock status.

Now, return to the livelock status in type (b). We want to point out that this type does not occur in SCCP. If in type (b), transition t_1 fires and t_2 does not fire, the buffer becomes not empty the next time, therefore the disabled transitions t_3 and t_4 become enabled at the next loop, realizing a loop of type (a). Thus type (b) never occurs meaning there is no livelock.

The characteristic that SCCP is livelock-free is an attractive feature of SCCPs since this means we need not worry about the scheduling.

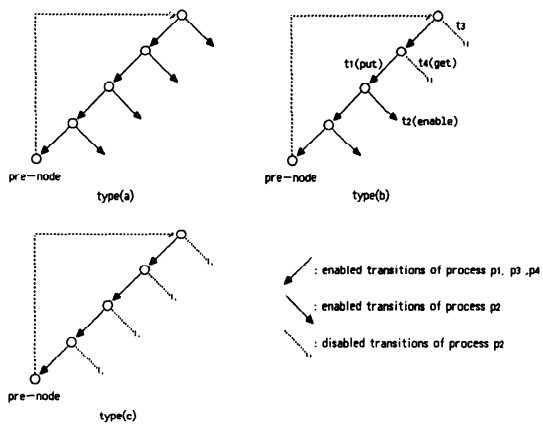


Fig. 11 Three types of loops in SCCP marking graphs.

4.3 Analysis to Determine the Necessary Buffer Size

In SCCP, processes communicate with each other through streams which are realized by buffers. Determination of necessary buffer sizes before the actual execution of SCCP networks is important for allocating a buffer of proper size to each stream. This analysis is also needed to determinate whether or not in-line expansion [Kus] is possible. The necessary size (including infinity) can be determined by the following algorithm using the capacity of each buffer place in the Petri net.

Algorithm 4.2

Let $b_i(i=1, \ldots, m)$ be a buffer. We use b_i for both the buffer in the running program and the buffer place in the Petri net corresponding to the program. Let $C(b_i)$ be the capacity of buffer b_i. Let $M(pl_j, n_k)$ be the marking of place pl_j at node n_k.

1. For $b_i(i=1, \ldots, m)$, set $C(b_i)$ to 0. Generate a marking graph with capacity $C(b_i)$ for b_i.
2. If there are no full-nodes, set the size of each $b_i(i=1, \ldots, n)$ to $C(b_i)$ and stop. Otherwise, go to step 3.
3. Choose one b_x which is full at a full-node, and let

$$C(b_x) \leftarrow C(b_x)+1.$$

4. Extend the marking graph using the new capacity $C(b_i)$. If a node n_1 and a buffer b_x satisfy the following conditions, let $M(b_x, n_1)$ and $C(b_x)$ be ω (infinite) and make n_1 a pre-node.

(1) At the level of the marking graph including n_1, there are no other nodes except for n_1.

(2) There is another node n_0 in the marking graph where

$$M(pl_j, n_1)=M(pl_j, n_0) \ (pl_j \neq b_x)$$

and

$$M(b_x, n_1)>M(b_x, n_0).$$

Go to step 2.

For example in Fig. 12, we start with 0 capacity for b_1 and b_2. The marking graph is only part (a) and (1, 1, 0, 0) is a full-node. Since there is a full-node in the marking graph, increase the capacity of b_2 ($C(b_2)$) from 0 to 1, and extend the marking graph from the full-node. Now part (b) is added to the marking graph. There is again a full-node (1, 2, 0, 1). Increase the capacity of b_2 from 1 to 2. This time part (c) is added to the marking graph and there are no full-nodes. Thus the necessary sizes of buffer b_1 and b_2 are 0 and 2, respectively.

An example of the determination of infinite buffer size is shown in Fig. 13. After extending the marking graph from the full-node (1, 4, 0, 0), we get node n_1 which is (1, 1, 0, 1). Since n_1, n_0 and b_2 satisfy the condition described in step 4 of the algorithm, we let $C(b_2)$ be ω and set n_1 to be (1, 1, 0, ω) and a pre-node. The result is that the necessary size of b_1 is 0 and the necessary size of b_2 is infinite.

A proof of the correctness of the algorithm is shown in Appendix A.

4.4 Examination of the Possibility of Termination

A network, except for filter-type networks, is usually assumed to terminate sooner or later. Thus, analysis of a network for its possibility of termination is generally useful.

In SCCP networks, each process has a terminating machanism at least implicitly with exception handling statements. But, the possibility of termination of a network depends on how the processes are connected.

If there are no end-nodes on the marking graph, the network is nonterminating. On the other hand, if all paths on the marking graph lead to end-node(s), the network always terminates. If some paths (with branches) lead to end-node(s), the network has a possibility of termination. Whether or not termination occurs depends on the semantics of the program. For example, "sum-sqr" may possibly terminate since there is an end-node in Fig. 6.

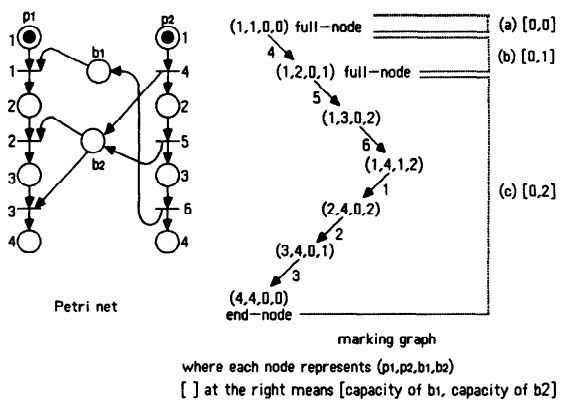


Fig. 12 Analysis of necessary buffer size.

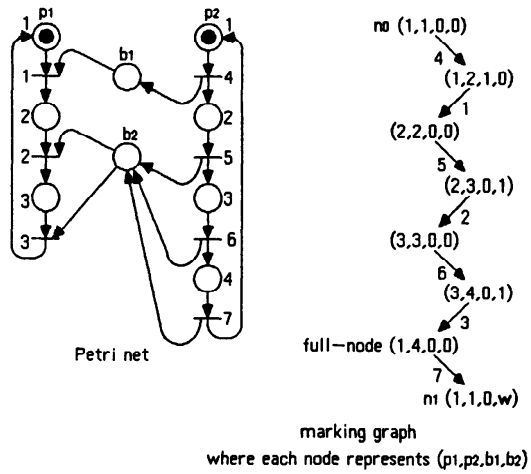


Fig. 13 Analysis of necessary buffer size.

4.5 Detection of Dead Code

Dead code may be caused by linking individual processes to make a network. This is because each process usually has exception statements, but some of these may never be executed in the connected network. The detection of such dead code is useful since programmers may mistakenly expect that they are executed in exception statements and it is not always easy to detect them by simple observation.

Dead code of a network can be detected easily as transitions which do not appear in the marking graph of the network.

For example, in the case of "sum-sqr", transition 11 is dead code (Fig. 5) since it does not appear in the marking graph of Fig. 6.

5. Automatic Analyzing System

We constructed a system called SPRAT as an automatic analyzing system for SCCPs. It consists of two parts, a translator and an analyzer. The translator translates an SCCP into an equivalent Petri net. The analyzer analyzes the subjects described in section 4 and also performs an in-line expansion which is a transformation of concurrent processes into a sequential process. While the translator depends on the programming language Stella, the analyzer is independent of the language used. This system has been implemented on a VAX-11/750 using Pascal.

The analyzer analyzes and transforms an SCCP network modelled by a Petri net in the following order.

(1) Detection of deadlocks

A marking graph is constructed for analysis of deadlocks assuming that all buffers are infinite.

(2) Analysis of buffer sizes

(3) Examination of possibility of termination

(4) Detection of dead code

(5) In-line expansion

If, as a result of (2), all buffers are shown to be realizable with a finite size, the network can be in-line expanded.

We analyzed many networks with SPRAT, and obtained useful information of them. For example, for a program of an inventory control system [Ku2] which includes four processes and six streams, SPRAT detected a deadlock that would be difficult for us to detect by hand. In the program for a Hamming problem [Di2], SPRAT correctly found that three streams need infinite buffers. When we regard processes as black boxes and connect them, it often happens that the network never stops. SPRAT has also detected such a situation in the program for the Hamming problem. SPRAT also did in-line expansions which would be extremely difficult to perform by hand. In Appendix B, an example of the analysis for the "sum-sqr" program by SPRAT is shown.

6. Concluding Remarks

We have defined a class of networks of concurrent processes connected by streams and labeled this class SCCP. One special property of SCCPs is that each stream has one producer and one consumer. We have modelled and analyzed an SCCP using a Petri net.

We showed how to detect deadlocks and dead code, and how to determine the possibility of termination of an SCCP. We also presented an analysis for finding the necessary buffer sizes. Buffer size analysis has not been studied much in other works concerning general concurrent processes.

We have also shown important characteristics of SCCP. The occurrence of deadlocks is restricted and can be checked more easily than in general concurrent processes. There is no livelock in SCCP. These results demonstrate a good basis for programming and validation using SCCPs.

In this paper, we have not fully examined program semantics, i.e., both branches in each conditional branch are assumed to have a possibility of being executed. Though this assumption seems to be a good approach without getting into complex reasoning on program semantics, some results of this analysis, e.g., decision of deadlock and buffer size may be worse than what occurs in reality. For example, the analysis may sometimes indicate a need for an infinite buffer size, even if this need can never arise in practice. This is one limitation of an analysis which does not include program semantics. But, in many cases the user will be able to find more useful properties by adding other information concerning the actual behavior of the program.

Using SPRAT for this analysis can be a great help to programmers of SCCPs.

Acknowledgements

I would like to thank Harushi Ishizuka who studied the analysis with me, and Kenji Toriya who implemented the automatic analysis system SPRAT with me. I would also like to thank David Duncan who helped with the English in this paper.

References

[Age] AGERWALA, T., Comments on Capabilities, Limitations and "Correctness" of Petri Nets. *Proc. 1st. Ann. Symp. Computer Architecture, ACM*, (1973), 81–86.

[Arv] ARVIND and BROCK, J. D. Streams and Managers, *Lecture Notes in Computer Science*, **143**, 452–465, Springer.

[Bae] BAER, J. L., Modeling for Parallel Computation: A Case Study, *Proc. of the* 1973 *Conf. on Parallel Processing, IEEE*, (1973), 13–22.

[Bur] BURGE, W. H. Stream Processing Functions, *IBM J. Res. Dev*, **19**, (1975), 12–25.

[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*, (Oct. 1981), 171–178.

[Den] DENNIS, J. B. and WENG, K. K.-S. An Abstract Implementation for Concurrent Computation with Streams, *Proc. of the* 1979 *Int. Conf. on Parallel Processing*, (1979) 35–45.

[Dij] DIJKSTRA, E. W. Hierachical Ordering of Sequential Processes, *Acta. Inf.*, **1**, 2, (1971), 115–138.
[Di2] DIJKSTRA, E. W. A Discipline of Programming, Prentice-Hall (1976).
[Hag] HAGINO, T. Proofs of Communicating Sequential Processes, Preprint of WG on *Software Foundation 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*, **21**, 8, (Aug. 1978), 666–677.
[Kah] KAHN, G. and MACQUEEN, D. B. Coroutines and Networks of Parallel Processes, *Information Processing* 77, North-Holland, (1977), 993-998.
[Kus] KUSE, K. Analysis and Program Transformation of Concurrent Processes Connected by Streams, Master Thesis, Univ. of Tsukuba (1984).
[Ku2] KUSE, K., Programming an Inventory Control System Using Stella- A Programming Language with Streams, *J. IPS Japan*, **26**, 5, (in Japanese) (1985), 497–505.
[Ku3] KUSE, K. and SASSA, M. Analysis of Necessary Buffer Size for Concurrent Processes Connected by Streams, Tech. Momo PL-12, PL Research group, Univ. of Tsukuba (1986).
[Kwo] KWONG, Y. S. On the Absence of Livelocks in Parallel Programs, *Lecture Notes in Computer Science*, **70** 172–190, Springer.
[Mat] MATSUBARA, Y. Capacity Designated Petri Nets, *Trans. IEICE Japan*, **62**, 5, (in Japanese) (1979), 309–316.
[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, *Comput. Surv.*, **9**, 3, (Sep. 1977), 223–252.

Appendix A

Proof of the algorithm

We only prove the nontrivial point, i.e., the correctness of the determination of infinite buffer size. (The determination of buffer size for the case of finite size is obvious.) For this purpose, it is enough to prove that the following proposition holds when there are nodes n_1 and n_0 and buffer b_x satisfying the condition in step 4 of algorithm 4.2.

Proposition

Even if we extend the marking graph from node n_1 by increasing $C(b_x)$, another node n_2 which is the same as n_1 except for a larger b_x marking and which satisfies the condition will appear again. Inductively, when this fact is applied repeatedly the marking of buffer b_x eventually becomes infinite (see Fig. A1).

Proof

Assume that the marking graph is extended of node n_1. In this case, there will be a node n_2 with the same marking as n_1 except for b_x, for which the marking becomes larger. Let m_1 and m_2 be the part of marking graph from n_0 to n_1 and n_1 to n_2, respectively. Since the marking of buffer b_x at each node of m_2 is larger than the marking at the corresponding node of m_1, part m_2 includes (is a superset of) part m_1. (When $C(b_x)$ is 0, we made a convention allowing a path where the marking of b_x changes as $0 \rightarrow 1 \rightarrow 0$. Accordingly, when $C(b_x)$ is greater than 1, we may also add such a path $C(b_x) \rightarrow C(b_x)+1 \rightarrow C(b_x)$ to the marking graph without a loss of generality.)

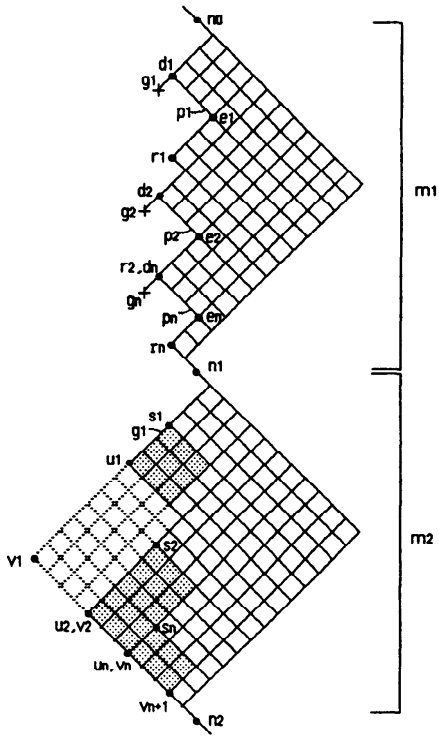


Fig. A1 Proof of determination of infinite buffer size.

There may be a new path in m_2 which does not have the corresponding path in m_1 (meshed part in Fig. A1). We want to show that even if there is an expanded part in m_2, node n_2 is a single node in that level.

Let start nodes of such expanded paths in m_2 be $s_i(i=1, \ldots, n)$ and the corresponding node in m_1 be $d_i(i=1, \ldots, n)$. Since the paths do not exist in m_1 but exist in m_2, the marking of b_x of d_i is 0 and the marking of b_x of s_i is more than 1, and the control point of one process (let us call it a consumer process) at d_i and s_i is before a get operation. Since the marking of b_x is 0 at node d_i and it is greater than or equal to 1 at node n_1, there must be at least one transition which is a put operation to buffer b_x on any paths from d_i to n_1. Let $p_i(i=1, \ldots, n)$ be such transitions which first appear starting from d_i. Let the node immediately after p_i be $e_i(i=1, \ldots, n)$.

Since the marking of b_x is 1 at node e_i is 1 at node e_i and the control point of the consumer process does not change from d_i up to e_i (it is before a get operation), transition g_i for the get operation from b_x is enabled at e_i. Let $r_i(i=1, \ldots, n)$ be a node reached from e_i by firing only transitions of the consumer process.

In m_2, let $u_i(i=1, \ldots, n)$ be the node reached from s_i by firing only transitions of the consumer process. Let $v_i(i=1, \ldots, n)$ be a node reached from s_i by

(hypothetically) firing only the transitions of the consumer process up to the point where the consumer process is at the same status as in r_n. The transition sequence from s_i to v_i is the same as the concatenation of the transition sequence from e_i to r_i, consumer process transitions from r_i to d_{i+1}, from e_{i+1} to r_{i+1}, . . . , and from e_n to r_n. Let v_{n+1} be the node in m_2 corresponding to r_n in m_1.

In m_1, the cause for transitions of the consumer process to be disabled at d_i or r_i is either that (i) the transition is a get operation from an empty b_x or (ii) the transition is a get operation from another empty buffer or (iii) the transition is a put operation to another full buffer. In case (i), there may be expanded paths in m_2, and edges from s_i to u_i may exist. In case (ii) or (iii), there are no expanded paths in m_2 since the status of those empty or full buffers does not change in m_2.

Now, let us examine r_n. We show that the consumer process can not fire at r_n not because of cause (i). If we suppose that the consumer process could not fire at r_n because of cause (i), the marking of b_x at r_n should be 0. Since the marking of b_x is more than 1 at n_1, there is at least one put transition to b_x between r_n to n_1 which will then enable the get transition from b_x. Thus, there exists another expanded path in m_2 starting from such node. This contradicts the assumption that d_n or s_n corresponds to the last starting node for expanded paths.

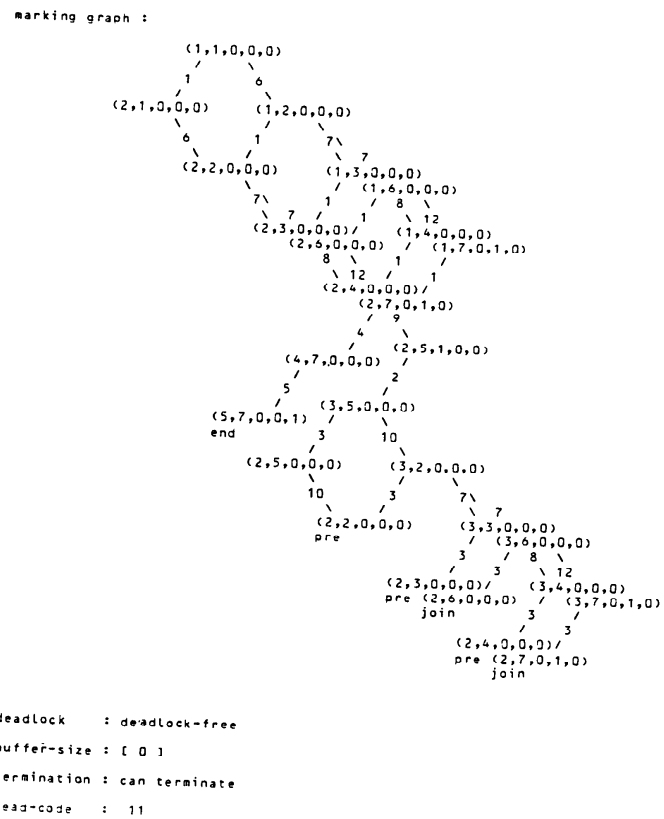
Since the consumer process could not fire at r_n because of cause (ii) or (iii) and the status of buffers other than b_x does not change in m_2, the consumer process can not fire at node v_{n+1} and the paths from v_{n+1} to n_2 is the same as the paths from r_n to n_1. Considering also that the condition by which the consumer process can fire at $v_i(i=1, \ldots, n)$ is the same or stricter than the condition at v_{n+1}, the transition sequence from s_i to $u_i(i=1, \ldots, n)$ which actually fires must be included in the transition sequence s_i to v_i. In other words, transition sequence from s_i to $u_i(i=1, \ldots, n)$ is limited by the straight line $v_1, v_2, \ldots, v_{n+1}$ in the marking graph.

From the above facts, all new paths go through n_2 and n_2 is a single node in that level. Then n_2 satisfies the condition in step 4 of algorithm 4.2 and the proposition holds.

There are two specific cases which require a more detailed proof. The first case is that the marking graph from n_0 to n_1 includes a conditional branch and the second case is that more than two buffers are full at a full-node. The proof for these cases can be made similarly to the above one. The details are given in [Ku3].

Appendix B

An analyses generated by SPRAT for sum-sqr




```
deadlock    : deadlock-free
buffer-size : [ 0 ]
termination : can terminate
dead-code   : 11
```

(Received March 10, 1986; revised July 11, 1986)