Petri Net Modelling of Occam Programs for Detecting Indeterminacy, Non-termination and Deadlock Anomalies

Zhiwei Xu

Department of Computer Science,
New York Polytechnic University,
New York 11735,
United States of America.
zxu@polyof.poly.edu

Olivier de Vel

Department of Computer Science,
James Cook University,
Townsville Q4811,
Australia.
olivier@curacoa.cs.jcu.edu.au

Abstract

In this paper we present a graph model based on Petri nets that can be used as a software development tool for parallel computational programs written in the occam language. In parallel computing the cooperation aspect of concurrency is emphasized, rather than the competitive aspect of multiple processes found in operating system applications. In developing and debugging a parallel computational program, users often want to ensure that their program will terminate (that is, it stops in finite time) and be determinate (that is, the same input data always produce the same result). Using the Petri net graph model, termination and determinacy can be mathematically defined, and algorithms for detecting these properties developed. In fact, the graph model defines an operational semantics for a non-trivial subset of occam, and can be used as a pedagogical aid as well as a tool for developing and synthesizing occam programs in parallel computing applications.

1 Introduction

The processing activities undertaken in parallel computing are different from those used in operating system applications. Although both have to deal with concurrency, an operating system is more interested in the *competition* aspect [1]. Here, multiple processes compete for shared resources. The operating system often makes these resources a critical region, and uses synchronization techniques such as locks, semaphores, and monitors to ensure a safe and fair use of them. This is usually achieved through mutual exclusion, i.e., only one process is allowed to enter a critical region at a time.

Parallel computing, on the other hand, emphasizes the *cooperation* aspect of concurrency. Multiple processes cooperate in computing a function, each process passes its (partial) result by communicating with the others. Users usually want to be ensured that their program is *terminating* (the program enventually stops) and *determinate* (the program always produces the same result for the same input), for then it is easier to test and debug the program for correctness and efficiency.

Indeed, one of the major applications of computing is the modeling, simulation and visualization of the physical world, as exemplified by computational physics, computational chemistry, computational biology, etc.... If parallel computing is to help scientists, it must have two basic properties of scientific experiments : termination and determinacy. That is, a scientist should be able to complete an experiment in a finite amount of time, and the experiment should be repeatable, even by other scientists.

This paper reports our research in developing a Petri net graph model for detecting termination, determinacy and deadlock of a parallel computing program using the occam programming language. The Petri net model also defines an operational semantics for a sufficiently rich subset of occam, thus it can be used as a tool for both the teaching and the synthesis of parallel occam programs. We note that the results can be generalised to other parallel languages.

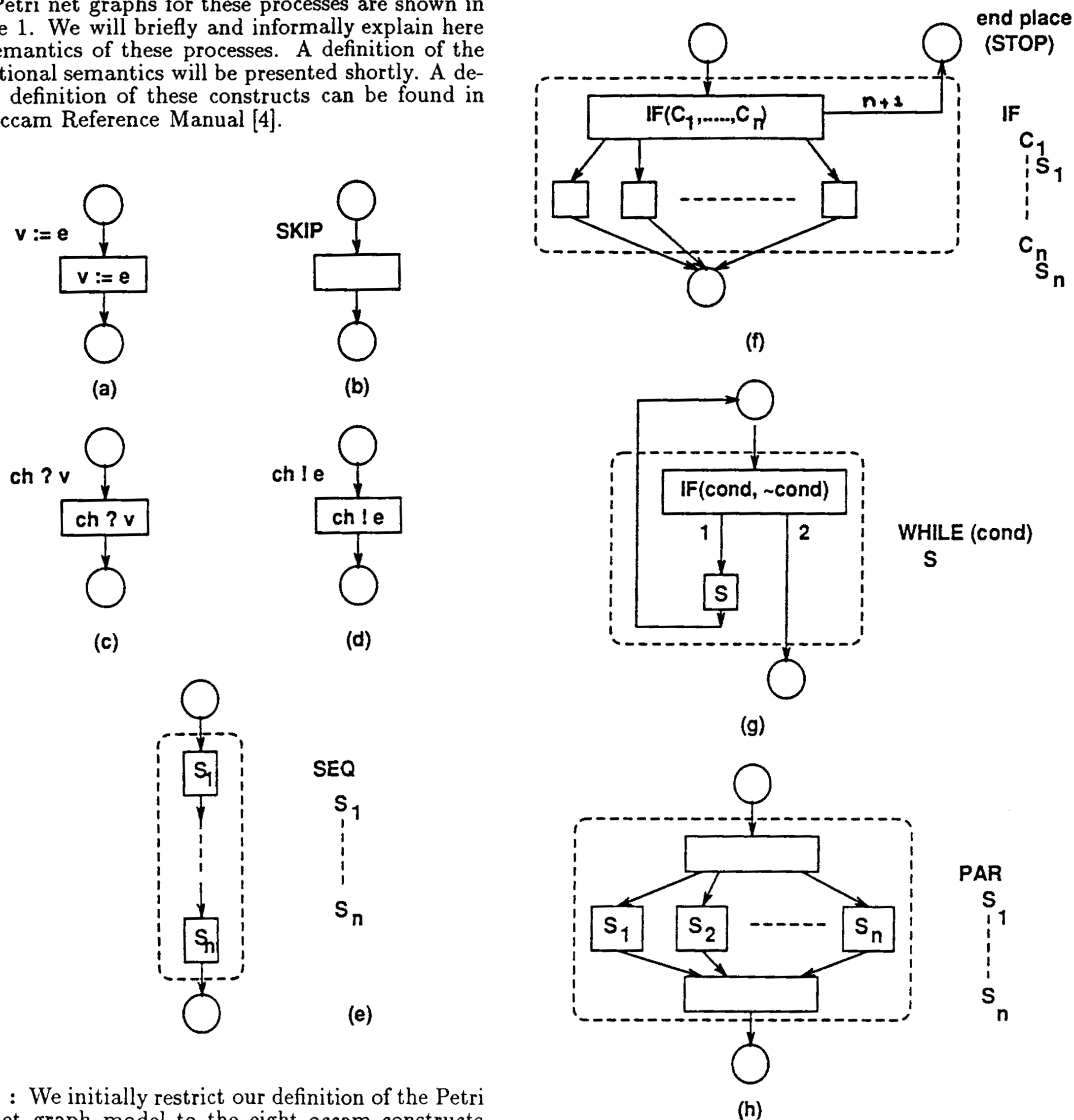
2 The Petri Net Model of Occam Programs

We shall represent a Petri net by a graph [5], where places are denoted by circles and transitions by rectangles. We shall also use *virtual* nodes, denoted by squares, to represent subnets (or subgraphs). If an arc in a graph points from node v_i to node v_j, we call v_i a *precedent* of v_j, and v_j a *successor* of v_i. A *marking* assigns to each place a nonnegative integer. If a marking assigns to place p_i a nonnegative integer k, we say that p_i is marked with k *tokens*. Pictorially, we place k black dots (i.e. tokens) in place p_i.

Occam is a simple, elegant and powerful concurrent programming language, derived from the CSP language developed by Hoare [3]. The basic model for an occam program is a network of communicating processes, each process being the main vehicle for the specification of an action which can take place in parallel with other actions. Parallel computing programs written in the occam language are built from four types of *primitive processes* (assignment, skip, input, and output) and four types of *combining processes* (se-

TH0386-3/91/0000/0116/$01.00 © 1991 IEEE

quential, parallel, conditional, and loop). The needs of data communication and synchronisation in occam are combined in the input and output primitive processes. The Petri net graphs for these processes are shown in Figure 1. We will briefly and informally explain here the semantics of these processes. A definition of the operational semantics will be presented shortly. A detailed definition of these constructs can be found in the Occam Reference Manual [4].



Note : We initially restrict our definition of the Petri net graph model to the eight occam constructs shown in Figure 1. The reasons are that we want our definition to be simple, and that our target application is parallel computing. We choose not to explicitly define graphs for the occam constructs **FUNCTION**, **PROC**edure, and replicators because they can always be expanded into and defined in terms of the eight constructs. The other two constructs, **STOP** and **ALT**ernation are, at this stage, not defined at all. Of course, it is still possible to design parallel programs by using the **PAR** construct and channel I/O.

Fig. 1 : *Primitive and combining processes and their Petri net graphs. (a) Assignment process, (b)* **SKIP** *process, (c) Input process, (d) Output process, (e) Sequential process, (f) Conditional process, (g) Loop (***WHILE***) process, and (h) Parallel process.*

The assignment and the loop processes are quite similar to the corresponding statements in Pascal.

The **SKIP** process and the **SEQ**uential process correspond to the empty statement and the **begin**...**end** compound statement in Pascal, respectively. The conditional process is similar to the conditional statement in Pascal, except that in occam all the conditions (referred to as *guarded choices*) are evaluated first, then the subprocess whose condition is true is executed. If no condition is true, the conditional process cannot proceed and it cannot terminate. The parallel process is just Dijkstra's **parbegin**...**parend** construct [2]. A pair of input and output processes with the same channel (as, for example, channel *ch* in Figure 1(c) and (d)) must be executed simultaneously, which pass the value of the expression e to the variable v. An exception to this rule happens when the channel is a *hard* channel, which is used to realise communication to external I/O devices in occam. An input or output process with a hard channel (such as *Keyboard* ? x or *Screen* ! e) can be executed alone, without having to wait for a partner process to get ready.

Any combining process has a number of *subprocesses*, which can be any primitive or combining processes themselves. In Figure 1, a square (or virtual node) denotes a subgraph representing a subprocess. Note that, with our Petri net model, there are only five types of transition nodes in a graph for any parallel computing program : *assignment*, *skip*, *if*, *input*, and *output* transitions.

A *program* is defined as any primitive or combining process, which is called the *main process* of the program. A complex program can be built by combining one or more subprocesses, which themselves can be combinations of other subprocesses. A program can be syntactically translated to a Petri net graph in the following four steps :

1) Label all processes of the program distinctly.

2) Each process is translated according to Figure 1.

3) Recursively expand all virtual nodes in the main process graph. That is, replace a virtual node by the corresponding subgraph.

4) Eliminate redundant places as shown in Figure 2.

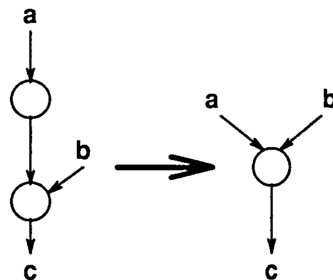


Fig. 2 : *Elimination of redundant places.*

We note that most process graphs in Figure 1 have a unique *begin place* (that is, the place without any precedent), and a unique *end place* (that is, the place without any successor). The only exception being the **WHILE** process graph (see Figure 1(g)), where we define the begin place as the precedent place of the if transition. With this amendment, the graph of any program constructed in the above way will have unique begin and end places, which are the ones of the main process graph.

As an example, Figure 3 shows the graph for the following occam program segment (which has a communication deadlock) for computing $(x, y) = (f(x, y), g(x, y))$ using an iterative method :

```
P1:   WHILE notconverged
P2:     PAR
P3:       SEQ
P4:         ch1 ? a
P5:         ch2 ! y
P6:         b := y
P7:         y := g(a, b)
P8:       SEQ
P9:         ch2 ? d
P10:        ch1 ! x
P11:        c := x
P12:        x := f(c, d)
```

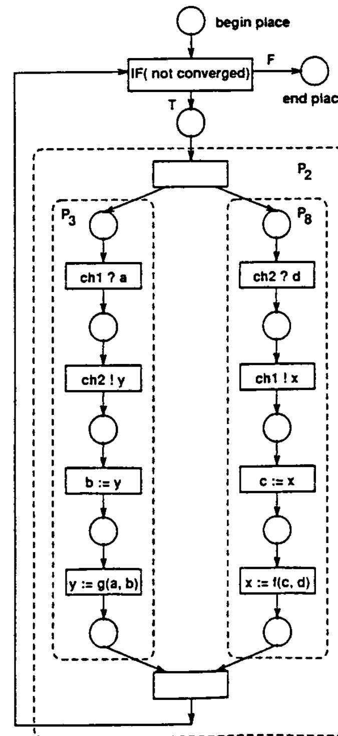


Fig. 3 : *Example occam program Petri net graph.*

The main process is the **WHILE** process P_1, which

contains a parallel subprocess P_2, which in turn contains two sequential subprocesses P_3 and P_8. The begin and end places of the program are just the ones for the main **WHILE** process. Two communication channels are used, $ch1$ and $ch2$. In subprocess P_3, variable a first receives a value from channel $ch1$ and, secondly, the value of variable y is output on channel $ch2$. Concurrently, in subprocess P_8, variable d first receives a value from channel $ch2$, followed by the output of the value of variable x on channel $ch1$.

Now let us define the semantics of a program P. Denote by $S(t)$ the *state* of the program P at any time t, where the state is the content of all variables, files, etc. involved in P. At $t = 0$, we put a token in the begin place of the program's Petri net graph. At subsequent times, transition nodes of the graph will fire, which updates the state of the program. This firing process continues up to the time when a token is put into the end place of the program graph, then we say the program *terminates*.

There are three rules for transition firing, defining (i) the conditions upon which a transition can fire (i.e., is enabled) ; (ii) the timing of firing ; and (iii) the effects of firing :

(i) An *assignment*, *skip*, or *if* transition are enabled when there is a token in each of its precedent place(s). A pair of input and output transitions with the same channel name both are enabled when there is a token in every precedent place of each of them.[1]

(ii) When a transition is enabled, it will eventually fire in a finite amount of time, although the exact moment of firing is not known. This is called the finite-progress and nondeterminism assumption [7].

(iii) The effects of firing various transitions are shown in Figure 4. An *assignment* transition fires by removing a token from each precedent place, performing the corresponding assignment operation to update the program state, and placing a token in all its successor place(s). A *skip* transition fires in a similar way, except that the state is not changed. The firing of an *if* transition $\mathbf{IF}(C_1, C_2, \ldots, C_n)$ differs from the skip transition in that a token is placed in only the successor of the i-th branch, if C_i is TRUE. The $C_1, C_2, \ldots, C_n$ are evaluated in sequence until one is found which yields the value TRUE. If none of the $C_1, C_2, \ldots, C_n$ are TRUE, the conditional behaves like a **STOP** construct ; that is, the process starts but never proceeds and never terminates. In this case a token is placed in the $(n+1)$th branch. It should be preferable (although, syntactically not necessary) to include a $C_n :=$ TRUE guard condition together with a **SKIP** process to trap the unwanted stopping. We say the conditions are *exhaustive*, if $C_1 \vee C_2 \vee \ldots \vee C_n =$ TRUE.

A pair of input and output transitions with the same channel name fire by each removing a token from every precedent place, performing the communication operation to update the state, and placing a token in every successor place.

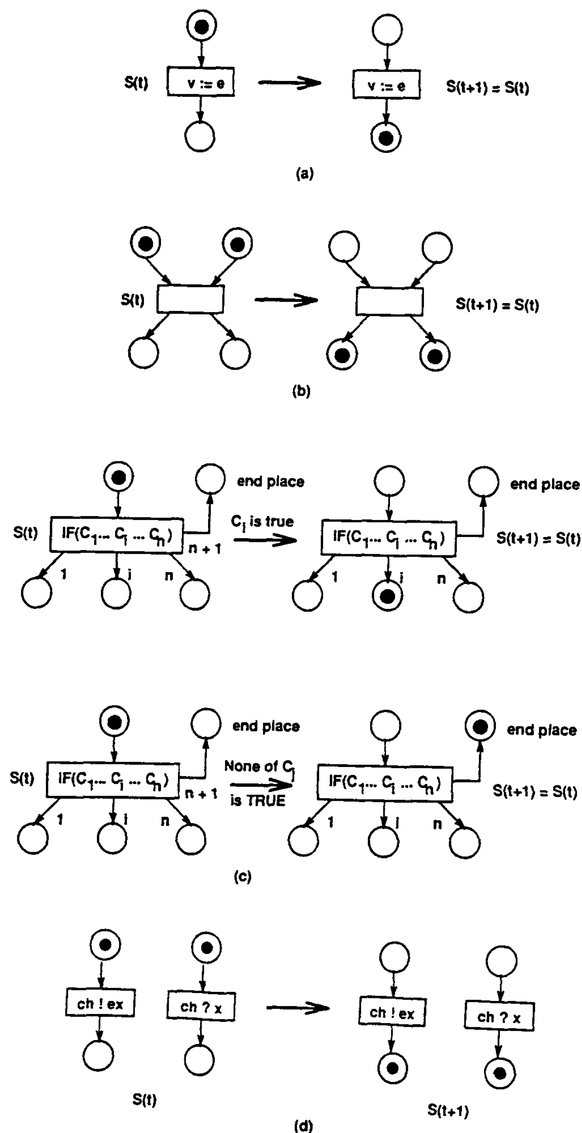


Fig. 4 : *The effects of firing various transitions at time t. (a) assignment transition, (b)* **SKIP** *transition, (c)* **IF** *transition, and (d) I/O transition.*

[1] This rule does not apply to an input (or output) transition with a hard channel, which is enabled if there is a token in its precedent place.

Definition 1. *A program (or its corresponding Petri net graph representation) is* terminating, *if a token will be placed into the end place of the graph in finite time after the program begins execution with any possible initial state* $S(0)$.

Definition 2. *If a program P is terminating, we can consider P as a mapping from* $S(0)$ *to* $S(n)$, *where n is the time P terminates. P is said to be* determinate *if the mapping is a function (i.e., one-to-one). That is, given an initial state* $S(0)$, *we always get the same final state* $S(n)$.

The important thing here is that, for a given program P, we have to consider all possible initial states $S(0)$ and all possible runs of the program. If there is a single run of P such that P does not terminate, P is not terminating. If there is a single initial state $S(0)$ with which two runs of the same program P may result in different final states, P is not determinate, in which case we say that P is *indeterminate*.

3 Indeterminacy and Non-Termination Anomalies

Even in sequential programs, indeterminacy is not an unknown phenomenon. A sequential program with the same input data may produce different results in different runs, when a variable is assigned a value through random number generation. However, this type of indeterminacy is not an anomaly, since it is a desired effect in applications such as Monte Carlo simulation. In other words, the *application* is of an indeterminate nature. Consequently, we will ignore this type of indeterminacy (or randomness) by assuming that random number generation is not used in programs.

It may happen that the application function to be computed is mathematically determinate. However, the parallel program realizing the function becomes indeterminate as a result of parallel execution. It is this type of indeterminate anomaly that we would wish to detect. As an example, consider the parallel process graph in Figure 5. We have in the parallel process two input transitions $ch\ ?\ x$ and $ch\ ?\ v$, both trying to receive a value from $ch\ !\ z$ through the same channel ch. Because of the nondeterminism assumption (firing condition (ii)), in one run of the program x may receive the value of z, while in another run v may receive the value of z. Thus it is possible that two runs of the same program with the same input data may produce different results.

Note : This parallel process graph also violates the semantic rule for the use of channels which requires that a channel can be used in at most one (different) parallel thread for output. This check is not performed at this level, but can be enforced at compile-time. In the absence of the **ALT** construct, and if this channel rule is not enforced, an occam program is determinate in the sequence of values communicated over each channel and in the program's termination behaviour when started in a known state. However, at this stage, we are not assuming that the channel rule is being enforced.

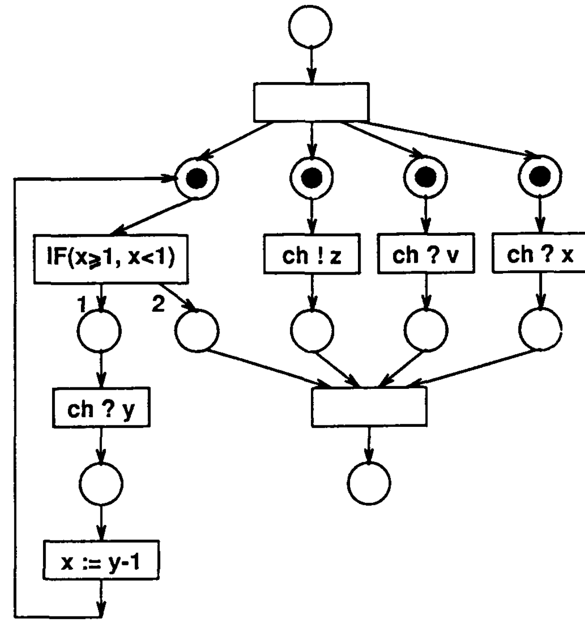


Fig. 5 : *A parallel process with WHILE-I/O and Odd-I/O indeterminacy anomalies.*

There are two reasons why a parallel program is not terminating : The first type of non-termination occurs when a program is trapped in an *infinite loop*, such as the trivial Pascal loop **while** $x > 0$ **do** $x := x + 1$ (assuming initially that $x > 0$). Once the program enters this loop, it will never exit out.

The second type of non-termination occurs when a program enters a state such that no processes can proceed because they are all *waiting infinitely* for some events (conditions) that will never occur. Examples of this include communication deadlocks and other anomalies to be discussed shortly.

These two types of non-termination have some notable differences :

- The first type (called *infinite looping*) may appear in both sequential or parallel programs, while the second type (called *infinite waiting*) only appears in a parallel program. Infinite waiting is the result of mutual waiting among multiple processes in a parallel program. Thus infinite waiting may also be called *parallel non-termination*.
- In infinite looping, something is repeatedly executed (some transitions fire repeatedly). In infinite waiting, nothing can be executed (no transition can fire).
- Detecting infinite looping (of the **WHILE** type) involves checking of semantics. This semantical checking problem is equivalent to the Turing Machine halting problem, which is undecidable. Therefore, in the remainder of this paper, we will focus on detecting infinite waiting by syntactical checking.

There are four types of anomalies that could lead to infinite waiting. These are listed as follows. We note in passing that these types all have to do with communication processes (i.e. input/output).

1) *WHILE-I/O* : Consider the left branch of the program in Figure 5. Since the number of iterations of the **WHILE** loop is run-time dependent, it surely can be more than one. In the second iteration, $ch\ ?\ y$ can never fire since the pairing of $ch\ !\ z$ with $ch\ ?\ v$ has already been passed. The program hangs at $ch\ ?\ y$.

2) *Odd-I/O* : Consider the parallel process in Figure 5 again. If we delete the **WHILE** loop branch, we will be left with two input and one output transitions in the remaining three branches. After one of the input transitions fires, the other can never fire since the pairing $ch\ !\ z$ has already been executed. Thus we end up with an odd input transition, and the program hangs.

3) *IF-I/O* : Consider Figure 6. When $b = 0$, $ch\ !\ w$ can never fire, since the pairing input transition $ch\ ?\ x$ is by-passed. The program hangs at $ch\ !\ w$.

4) *Deadlock* : Consider the program in Figure 3. Input transition $ch1\ ?\ a$ can fire only when the pairing $ch1\ !\ x$ is fireable, which can happen only after $ch2\ ?\ d$ fires. But $ch2\ ?\ d$ cannot fire until $ch2\ !\ y$ becomes fireable, which can happen only after $ch1\ ?\ a$ fires. This circular waiting results in a communication deadlock situation, and the program hangs at $ch1\ ?\ a$ and $ch2\ ?\ d$.

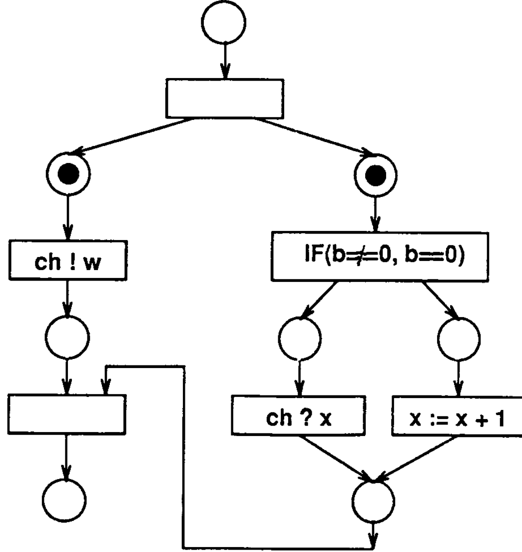


Fig. 6 : *A parallel process with an IF-I/O anomaly.*

4 The Algorithm for Detecting Infinite-Waiting and Indeterminacy

In this section, we present an algorithm for detecting indeterminacy and infinite waiting anomalies. Before presenting the algorithm, we need to make some assumptions. We say that the conditions $C_1, C_2, \ldots, C_n$ in a conditional process are *mutually exclusive*, if no pair of conditions can be true simultaneously (i.e., $C_i \wedge C_j$ = FALSE for any $i \neq j$). Mutual exclusiveness guarantees a correct conditional process in the sense that at any time, only one subprocess can be branched to. We assume the programmer or user is responsible for semantic checking : that is, ensures that the conditions in every conditional process are both exhaustive and mutually exclusive, and that the condition of any **WHILE** loop will eventually evaluate to be false. Thus, the program to be checked is assumed to be free of infinite looping.

4.1 The Main Algorithm

This algorithm checks the Petri net graph G of a given program P and reports infinite waiting and indeterminacy anomalies.

Step 1 : *Identify all (soft) channels in G and label them as $C_1, C_2, \ldots, C_n$.*

Step 2 : *Perform Procedure 2 (the Contraction Procedure) to check for possible IF-I/O, WHILE-I/O, or Odd-I/O anomalies.*

Step 3 : *Perform Procedure 3 to check for deadlock anomalies.*

4.2 Procedure 2 (The Contraction Procedure)

This procedure checks the Petri net graph G of a given program P and reports IF-I/O, WHILE-I/O, and Odd-I/O anomalies. This is done by recursively contracting (reducing) G into a primitive process graph and recording the pairing of input/output processes by a *channel vector* $\mathrm{M}(T)$.

Step 1 : *Delete all assignment, hard channel input/output, and skip transitions which have a single precedent and a single successor. Then combine the dangling precedent/successor pair into a single place.*

Step 2 : *For each I/O transition T_j with channel C_i, mark T_j with an n-dimensional channel vector $\mathrm{M}(T_j)$, where :*

n is the number of distinct channels in graph G;

$\mathrm{M}(T_j) = \langle 0, \ldots, 0, w_i, 0, \ldots, 0 \rangle$;

w_i is the i-th component of $\mathrm{M}(T_j)$, i.e. $w_i = M_i(T_j)$; and

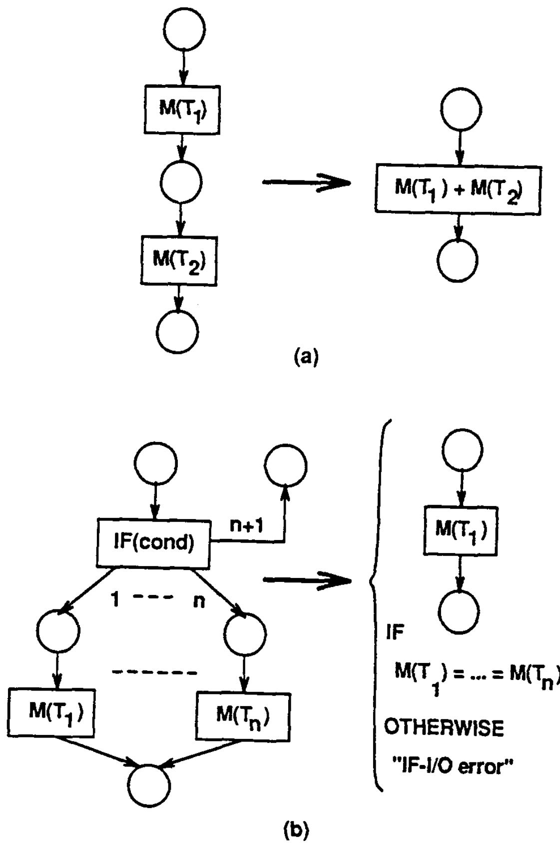
$w_i = 1\ (-1)$ if T_j is an input (output) transition node.

Step 3 : *Contract the obtained graph recursively by applying one of the four methods shown in Figure 7 to a smallest combining graph (i.e., a combining process graph that does not contain any combining process graph). The methods are sequential contraction, conditional contraction,*

loop contraction, and parallel contraction. The addition operator in Figure 7 is the ordinary component-wise vector addition operator. It may happen that during the contraction, some subprocess of a combining process becomes missing and without a channel vector. In this case, this subprocess is treated as if it had a zero channel vector, that is, channel vector $\mathrm{M}(T_j) = \langle 0, \ldots, 0 \rangle$.

If an infinite waiting anomaly is detected, report it and stop.

Step 4 : *If no infinite waiting anomaly is detected, the graph will finally be reduced to the graph as shown in Figure 8. Report "No anomaly in contraction" if all components of* $\mathrm{M}(T_j)$ *in Figure 8 are zero. Otherwise report "Odd-I/O with respect to channel* C_i*", if the i-th component of* $\mathrm{M}(T_j)$ *is non-zero, that is,* $M_i(T_j) \neq 0$.



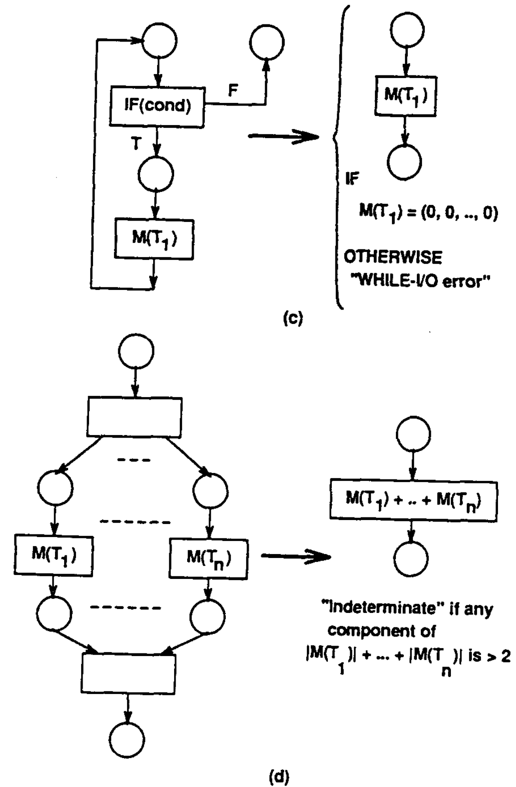


Fig. 7 : *Four contraction methods.* (a) *Sequential contraction,* (b) *Conditional contraction,* (c) *Loop contraction, and* (d) *Parallel contraction.*

As an example, consider Figure 5. Let us first consider the **WHILE** loop. After applying Steps 1 and 2 of Algorithm 1, we obtain the graph shown in Figure 9. Notice that there is only one channel *ch*, and the channel vector for the only input transition is $\langle 1 \rangle$. In Step 3, we apply the Loop Contraction method of Figure 7(c) to Figure 9 and detect a WHILE-I/O nontermination anomaly.

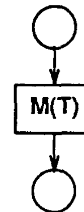


Fig. 8 : *Result of the Contraction Procedure, with no infinite waiting anomalies defined.*

Now suppose the user, upon receiving the error report, checks the program and decides that (s)he made a mistake in using channel *ch*, while (s)he really meant to input the value of y from the keyboard. The user corrects this mistake by replacing *ch* ? y with *Keyboard* ? y, and then applies Algorithm 1 again.

After Steps 1 and 2, the graph in Figure 10(a) results. Note that both transition nodes for the assignment $x := y - 1$ and hard channel input *Keyboard* ? y are deleted in Step 1, since they have a single precedent/successor. In Step 3, the smallest combining process, that is, the **WHILE** loop is first chosen and Figure 7(c) is applied. The resultant graph is shown in Figure 10(b).

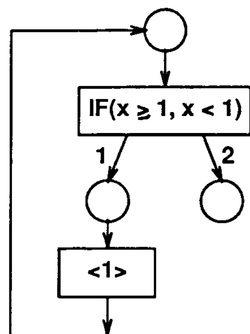


Fig. 9 : *Graph obtained by applying Steps 1 and 2 of the Contraction Procedure to the* **WHILE** *process of Figure 5.*

Since now we have a parallel process graph, parallel contraction in Figure 7(d) should be applied, which produces the graph in Figure 10(c) and, since ;

$$\sum_{j=1}^{4} |M_1(T_j)| = |0| + |-1| + |+1| + |+1| = 3 > 2,$$

reports an indeterminacy error. Finally, Step 4 is applied to Figure 10(c). Since the channel vector has a non-zero component, Algorithm 1 reports an Odd-I/O error associated with channel *ch*.

4.3 Procedure 3 (Static Communication Deadlock Detection)

This procedure detects static communication deadlocks :

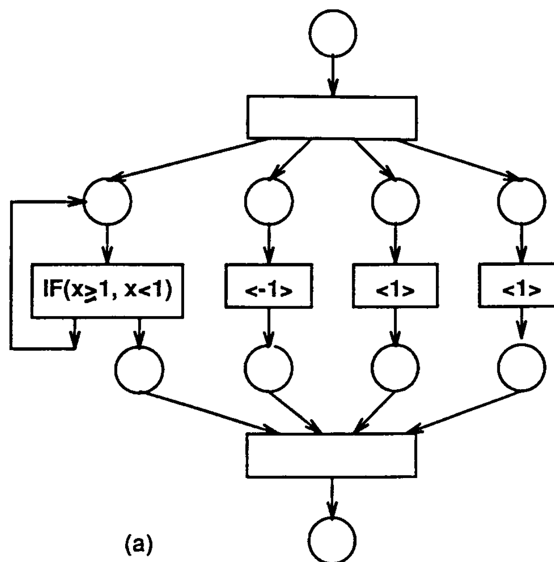
Step 1 : *Create a new graph G' from program graph G as follows :*

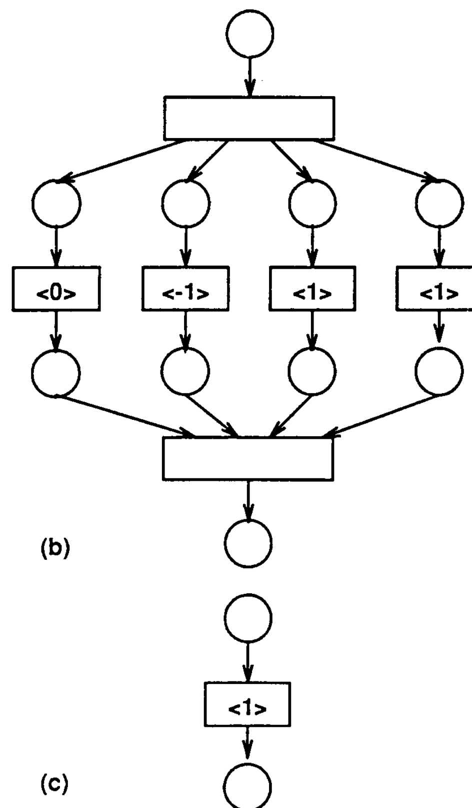
For each **WHILE** *loop subgraph H (as shown in Figure 1(g)) in G, delete the arcs to and from the loop body S. That is, isolate the loop body S from the rest of the graph.*

Step 2 : *Create a channel graph $C(G')$ from the program graph as follows :*

Each (soft) channel C_i is a node in $C(G')$; there is an arc from C_i to C_j in $C(G')$ if there is a path in G from I/O transition T_r with channel C_i to I/O transition T_s with channel C_j.

Step 3 : *Report "Deadlock" if $C(G')$ is cyclic, otherwise report "No communication deadlock anomaly".*





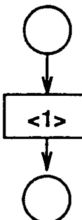
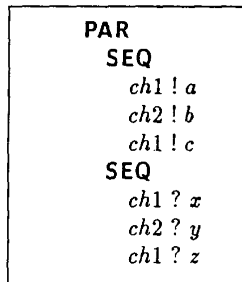


Fig. 10 : *Graphs produced by the Contraction Procedure in detecting indeterminacy and Odd-I/O*

anomalies.

As an example of static communication deadlock detection, consider the deadly embrace in its simplest form in Figure 3. In Step 1, the T arc from the **IF** transition and the feedback arc are deleted. In Step 2, a channel graph is cyclic, and a deadlock is reported.

A channel graph $C(G')$ contains channels of exactly one instance. If there are several instances of a given channel in a parallel component process, then the existence of a cycle does not necessarily imply that a communication deadlock has occurred. In such a case, a cycle in $C(G')$ is a necessary but not a sufficient condition for the existence of a deadlock. In the following example, a cycle exists but no dealock has occurred :



```
PAR
  SEQ
    ch1 ! a
    ch2 ! b
    ch1 ! c
  SEQ
    ch1 ? x
    ch2 ? y
    ch1 ? z
```

More than one channel graph is required if multiple instances of a given channel occur in a parallel component process.

5 Conclusions and Further Work

This paper has reported a Petri net model for detecting termination, determinacy and deadlock of parallel computing programs written in the occam language. Distinction has been made of non-termination due to infinite looping and infinite waiting. Four types of anomalies that could lead to infinite waiting have been identified. An algorithm for detecting these anomalies and indeterminacy is presented.

To clarify the presentation, we have given a simplified version of the algorithm for detecting non-termination and indeterminacy. A more sophisticated version of the algorithm should not only report the existence or absence of anomalies, but should also give their approximate locations in the program.

The algorithms that have been developed could also be reformulated by modelling each communication channel as a place (or a set of places), derive an "occam net", and perform structural and dynamic analysis. A similar representation is used in the context of Ada programs in [6]. Further work in this area is required.

Finally, the Petri graph model presented provides an aid for teaching concurrency as well as a tool for developing and synthesizing occam programs in parallel computing applications.

References

[1] F. Andre, D. Herman, and J.-P. Verjus. *Synchronization of Parallel Programs*, MIT Press, Cambridge, Massachussetts, 1985.

[2] E.W. Dijkstra. "Cooperating sequential processes", in *Programming Languages*, Genuys F. (Ed.), Academic Press, New York, NY, 1968.

[3] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[4] Inmos Ltd. *Occam 2 Reference Manual*, Prentice-Hall, Hertfordshire, England, 1988.

[5] T. Murata. "Petri nets : Properties, analysis, and applications", *Proc. of the IEEE*, **77**, 4, pp. 541-580, 1989.

[6] T. Murata, B. Shenker. and S. Shatz. "Detection of Ada static deadlocks using Petri net invariants", *IEEE Trans. on Software Engng*, **15**, pp. 314-325, 1989.

[7] M. Raynal. *Algorithms for Mutual Exclusion*, MIT Press, Cambridge, Massachussetts, 1986.