Detection of Ada Static Deadlocks Using Petri Net Invariants

TADAO MURATA, FELLOW, IEEE, BORIS SHENKER, MEMBER, IEEE, AND SOL M. SHATZ, MEMBER, IEEE

Abstract—This paper presents a method for detecting deadlocks in Ada tasking programs using structural and dynamic analysis of Petri nets. Algorithmic translation of the Ada programs into Petri nets that preserve control flow and message flow properties is described. Properties of these Petri nets are discussed, and algorithms are given to analyze the nets to obtain information about static deadlocks that can occur in the original programs. Petri net invariants are used by the algorithms to reduce the time and space complexities associated with dynamic Petri net analysis (i.e., reachability graph generation).

Index Terms—Ada, deadlock, invariants, Petri net, static analysis, tasking.

I. Introduction

ONE important contribution of the Ada programming language is its ability to support concurrent programming. Ada's tasking and rendezvous-based communication provide a convenient mechanism for programmers to implement both concurrent and distributed software. In this paper, we assume that the reader is familiar with Ada tasking and the related concepts of Entry call statements and Accept statements [1], [2].

Over the past few years, a few techniques have been proposed for use in the validation of Ada tasking programs. These range from static analysis techniques, which do not require actual execution of the source program, to dynamic analysis techniques, which introduce a special monitoring task into the program. Taylor [3] has presented a general static analysis technique for Ada tasking and has shown that such analysis has an exponential time complexity (in terms of the number of tasks). German [4] has described an approach for detecting deadlocks in Ada tasking programs using dynamic analysis. Since the approach uses dynamic analysis, results are dependent on the supporting environment, especially on the scheduler characteristics. In this paper, we propose the use of a Petri net invariant method for detecting deadlocks in Ada tasking programs. Our technique reduces to some extent the time and space complexities associated with the general state enumeration method used by Taylor.

Manuscript received November 26, 1986. This work was supported by the National Science Foundation under Grants ECS83-10719 and ECS85-10208.

T. Murata and S. M. Shatz are with the Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60680.

B. Shenker was with the Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60680. He is now with AT&T Bell Laboratories, Naperville, IL.

IEEE Log Number 8825773.

Our deadlock analysis technique starts with the translation of Ada tasking programs into Petri net models [5] and uses structural and dynamic analysis of these specific Petri nets, which we call Ada nets. Ada nets are abstract models of the source programs since an Ada net only models the Ada source program's control flow and message flow. As such, only those Ada statements which can alter the control flow (such as If, Loop, and Select statements) or which constitute a rendezvous (Entry calls and Accepts) contribute to the Petri net translation. Further details on the translation of Ada programs into Ada nets will be given in Section II.

Unlike the previously reported techniques of Shatz and Cheng [7], which are concerned with both deadlock and general tasking analysis, our method is not entirely based on generation of the complete Ada net reachability graph. Instead, we have been able to combine the specific properties of Ada nets with the Petri net concepts of place and transition invariants in order to reduce the complexity associated with generating the complete reachability graph.

Since the intuitive concepts and definitions of Petri nets are well known and space is limited, we will follow the terminology and notations of Petri nets as defined in [6], unless otherwise stated. Since the concepts of Petri net invariants play a key role in our subsequent discussion, we briefly review the definitions related to the invariants. For a Petri net having n transitions and m places, the incidence matrix is an $n \times m$ matrix $A = [a_{ij}]$ where $a_{ij} = a_{ij}^{+} - a_{ij}^{-}$, a_{ij}^{+} = the number of directed arcs from transition i to place j, and a_{ij}^{-} = the number of directed arcs from place j to transition i. An *S-invariant* (*T-invariant*) is defined as an integer solution[1] $y(x)$ of the homogeneous equation $Ay = 0 (A^T x = 0)$ where A is the incidence matrix of a Petri net. The subset of places (transitions) corresponding to nonzero entries of an m-vector y (an n-vector x) is called the *support*[2] and is denoted by $\|y\|$ ($\|x\|$) where m is the number of places and n is the number of transitions in a Petri net. A support is said to be *minimal* if no proper nonempty subset of the support is another support. An S-invariant (T-invariant) whose support is minimal is called a *minimal S-invariant* (*T-invariant*).

The invariants have the following intuitive meanings.

[1] All S-invariants (T-invariants) as used in this paper are nonnegative.

[2] The term "support" in this paper will also be applied to the vectors corresponding to markings and submarkings of an Ada net.

0098-5589/89/0300-0314$01.00 ©1989 IEEE

The firing count vector $\bar{\sigma}$ of a firing sequence σ leading from a marking M_0 back to M_0 is a T-invariant $x = \bar{\sigma} \geqq 0$. Conversely, given a T-invariant $x \geqq 0$, there exist a marking M_0 and a firing sequence σ leading from M_0 back to M_0 such that $x = \bar{\sigma}$, i.e., the ith entry of x is the number of times that the ith transition fires in the firing sequence σ, $i = 1, 2, \cdots, n$. On the other hand, the ith entry of an S-invariant y is a weight $y(i)$ associated with the ith place, $i = 1, 2, \cdots, m$, such that the weighted sum of tokens remains the same for all the markings reachable from an initial marking. There exists a simple algorithm to obtain all net invariants [8].

The remainder of this paper is organized as follows. Section II presents an overview of a technique for translating Ada tasking programs into Petri nets. Section III presents notations and properties associated with Ada nets. Section IV discusses the notion of static deadlocks and presents algorithms for detecting the two types of static deadlock: inconsistency deadlocks and circular deadlocks. Finally, Section V summarizes the main results of this paper.

II. Overview of Ada Transition by Example

Automated translation from Ada programs into Petri net models produces what we call Ada nets. This translation can be considered a part of the syntactic analysis and intermediate code generation phases of an Ada compiler. Thus, Ada nets would be generated along with the machine language code. The process of Ada net generation can be simply described as follows. Whenever the right-hand side of any Ada syntactic construct is recognized during parsing, the Ada actions associated with the construct are invoked (along with other code generation actions of the compiler). These actions generate a part of the Ada net, the part that models the given construct. Control then passes to the syntactic parent of the translated construct. The translation process continues until the root of the parse tree is reached. At this point, the input Ada program is either recognized or rejected. If it is recognized, the Ada net corresponding to the program will have been successfully constructed. The parts of the net that are generated are linked together into an Ada net that preserves the behavior of the input program. Consider the following fragment from an Ada program[3]:

```
   task body TASK1 is          task TASK2 is
   begin
1) while cond1 loop        0) entry ENTRY2;
2) TASK2.ENTRY2;              end TASK2;
3) end loop;                  task body TASK2 is
   end TASK1;                 begin
                           4) accept ENTRY2;
                              end TASK2;
```

The Ada translation maps the program fragment into the Ada subnet illustrated in Fig. 1(a). The dashed places,

[3]The statement numbers are not a part of the Ada syntax; they are used only for later reference.

Fig. 1. Stepwise translation of an Ada program into an Ada net.

transitions, and arcs represent connections to the parts of the program not shown.

First, we show that the behavior of the program fragment is properly modeled by the subnet. Then, we briefly discuss the translation process for the example. Place p_6 represents a Loop statement. The Entry call statement 2) is represented by t_3, t_4, p_5, p_2, p_3, and p_4. The Accept statement 4) is represented by t_5, t_6, p_7, p_2, p_3, and p_4. Places p_2, p_3, and p_4 represent the communication channel of ENTRY2 in TASK2. In general, every entry in any Ada program is represented by three unique places (p_{mcp}, p_{send}, p_{ack}) in the corresponding Ada net. Thus, for this example, the three places for entry "ENTRY2" are p_2, p_3, and p_4, respectively. Place p_2 is initially marked and belongs to the set of mutual exclusion places. Such a place is referred to as p_{mcp}. Execution of statement 2) is represented by firing transition t_3. This firing is possible only if p_{mcp} of ENTRY2 (place p_2) is marked. When p_2 is not marked, some Entry call of ENTRY2 must be currently involved in a rendezvous with an Accept statement of ENTRY2. When t_3 successfully fires, places p_3 and p_5 become marked. This submarking of the Ada subnet represents the situation in which the Entry call [statement 2)] is trying to initiate a rendezvous with an Accept statement of ENTRY2. When the Accept statement [statement 4)] is ready to rendezvous, t_5 fires and removes the token from p_3. Note that if there exists more than one Accept statement for ENTRY2, then p_3 serves as an input place to multiple transitions. The place p_3 is referred to as a sending place (p_{send}). Firing t_6 represents the control of TASK2 exiting the Accept statement. After the Accept statement ends, place p_4 enables transition t_4. This means that the control of TASK1 may exit from the suspended state (due to the Entry call). Place p_4 is referred to as an acknowledging place (p_{ack}). When t_4 fires, p_2 becomes marked. This means that no Entry call is now rendezvousing with any Accept statement of ENTRY2.

The details of translation from an Ada program into an

Ada net are fairly complicated and require syntactic analysis as well as some semantic analysis (for example, to distinguish package function calls from task Entry calls). The major technique is to generate pieces of the net during each production rule reduction during parsing. We briefly describe the idea in the previous example. Since it is assumed that Ada task specifications precede task bodies, statement 0) is reduced first. Three communication places (p_2, p_3, and p_4) for ENTRY2 in TASK2 are generated, as illustrated in Fig. 1(b). Statement 2) is reduced next. It causes t_3, t_4, and p_5 to be generated and linked with the communication places of ENTRY2, as illustrated in Fig. 1(c). The dashed places and arcs are not yet generated. The reduction of the Loop statement in TASK1 creates the Ada subnet illustrated in Fig. 1(d). Place p_6 is substituted for both dashed places in Fig. 1(c). Finally, statement 4) is reduced. Two new transitions, t_5 and t_6, are linked to the places p_3 and p_4 of ENTRY2. The Ada subnet for the program fragment is then complete and is illustrated in Fig. 1(a). A detailed discussion of the general Ada translation technique used here is found in [5]. Another approach to automated translation of Ada tasks into Petri net equivalent models is discussed in [7].

III. Properties of Ada Nets

A. Definitions and Characteristics

A sequential program is characterized by only one process running at any time. The control flow of a sequential program can be represented by a state machine, a subclass of Petri nets where each transition has exactly one input place and one output place. In a distributed environment, we usually have more than one process running at a time. If we disregard all communication between processes, such a distributed program can be modeled by a set of state machines, one state machine per process.

Definition: A *process-subnet* is a state machine that models one of the processes (Ada tasks) in a distributed environment when communication between processes is ignored.

Fig. 2(a) illustrates two separate process-subnets. Note that each begins with a place and ends with a place.

At translation time, we introduce two transitions into the Ada net: t_1 (par-begin transition) and t_2 (par-end transition) in order to give a cycle in the Petri net graph. These two transitions are connected through a place p_1 (cycle place). In Fig. 2(b), we have the two process-subnets of Fig. 2(a) structured as a distributed program with no communication. However, most useful distributed programs are characterized by some communication between the processes. In Ada, processes communicate through communication statements (i.e., Accept statements and Entry call statements). As discussed in Section II, every communication statement is represented by two transitions and at least four places. Fig. 2(c) illustrates the distributed program of Fig. 2(b) with some communication.

A transition that represents a communication statement

Fig. 2. Process-subnets in a distributed environment.

is referred to as an *AE transition* (Accept or Entry transition). Since every communication statement is represented by exactly two AE transitions in an Ada net, we will refer to one of them as "first" [e.g., t_5 in Fig. 2(c)] and the other one as "last" [e.g., t_6 in Fig. 2(c)], based on the order of their occurrence in a transition firing sequence starting with t_1. A place in a process-subnet that is either an output place of some first AE transition and/or an input place of some last AE transition is referred to as an *AE place* [e.g., place p_7 in Fig. 2(c)]. The places generated by the translation of an entry specification are referred to as *communication places* [e.g., p_2, p_3, and p_4 in Fig. 2(c)]. They are not considered part of any process-subnet. All noncommunication places are referred to as *sequential places*. Any arc connecting an AE transition and a communication place is referred to as a *communication arc*. A communication arc is not considered part of any process-subnet. All noncommunication arcs are referred to as *sequential arcs*. Three communication places, p_{mcp}, p_{send}, and p_{ack} (p_2, p_3, and p_4 in Fig. 2(c), respectively), are required for token (information) exchange between two or more process-subnets. The places may be thought of as a communication channel between potential "senders" and "receivers." The mutual exclusion place p_{mcp} guarantees that only one process-subnet may send a token at any time to the receiving process-subnet. No other process-subnet will send a token to that receiver until the current sender is acknowledged by p_{ack}.

Definition: A net is *strongly connected* iff there is a directed path from any place and transition to any other place and transition.

Definition: An *Ada net* is a strongly connected Petri net APN composed of process-subnets possibly interconnected with communication arcs and places. The strong connectiveness of APN is provided by p_1 (cycle place), t_1

(par-begin transition), and t_2 (par-end transition), as in Fig. 2(c).

We can easily observe the following important characteristics of any Ada net.

1) $|\cdot t_1| = |t_2 \cdot| = 1$ (p_1 serves as both the only input and the only output place of t_1 and t_2, respectively).[4] $|t_1 \cdot| = |\cdot t_2| = k$ where k is the number of process-subnets in APN (since t_1 concurrently enables all the process-subnets and t_2 synchronizes their terminations).

2) For the three communication places representing an entry E (channel) we have $|\cdot p_{mep}| = |\cdot p_{send}| = |p_{ack}^{\cdot}| = |p_{mep}^{\cdot}| = C_E$, ($C_E$ is the number of Entry calls of E). $|p_{send}^{\cdot}| = |\cdot p_{ack}| = A_E$ where A_E is the number of Accept statements of E. Let t_{af} and t_{al} be the first and last transitions, respectively, representing an Accept statement of E. Let t_e be any of the transitions representing the Entry calls of E. Then, $|\cdot t_e| = |t_e^{\cdot}| = |\cdot t_{af}| = |t_{al}^{\cdot}| = 2$ and $|t_{af}^{\cdot}| = |\cdot t_{al}| = 1$.

3) For any other transition t not discussed above, $|\cdot t| = |t \cdot| = 1$.

B. Dynamic and Structural Properties of Ada Nets

The initial marking M_0 for an Ada net represents the situation in which a program modeled by the net is ready to start executing. The marking M_0 consists of places p_1 and every p_{mep} initially marked with one token. No other places have tokens initially. M_0 for the Petri net in Fig. 2(c) is $(1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)^T$.

Theorem 1: An Ada net APN $< P, T >$[5] is safe for M_0, i.e., $M(p) \leqq 1$ for each place p in P and for any marking M reachable from the initial marking M_0.

Proof: Consider first an Ada net APN $< P, T >$ consisting of n process-subnets with no communication involved. Since each process-subnet is a state machine, the sequential places of each process-subnet together with the cycle place p_1 constitute the support of an S-invariant having exactly one token. There are n such S-invariants, and each place p in P belongs to one of these n S-invariant supports. Since the token content of each S-invariant support remains the same and is one for any marking M reachable from M_0, we have $M(p) \leqq 1$ for each place p in P and for any marking M reachable from M_0. Next, when there are communications between process-subnets of APN, by the construction of an Ada net, there are S-invariants whose supports consist of communication places (p_{mep}, p_{send}, p_{ack}) and some sequential places. Each of the supports of these S-invariants contains exactly one token (initially in p_{mep}). Thus, every place p in P is in the support of at least one S-invariant having one token. Therefore, $M(p) \leqq 1$ for each p in APN and for any marking M reachable from M_0. Thus, APN is safe for M_0. Q.E.D.

Theorem 2: An Ada net APN $< P, T >$ is conservative, i.e., there exists an m-vector y such that the weighted sum of tokens $y * M$ (the inner product of y and M), where $y(p) > 0$ for each p in P, remains the same for each marking M reachable from M_0.

Proof: Since each place in APN is in the support of at least one S-invariant, it is obvious that $y * M = y * M_0$ = a constant for any marking M reachable from M_0 where y is the sum of all minimal S-invariant vectors in APN and $y(p) > 0$ for each p in P. Therefore, APN is a conservative Petri net. Q.E.D.

Example: The incidence matrix of the Petri net shown in Fig. 2(c) is given by

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\ \begin{bmatrix} -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \end{array}$$

Each place in the net shown in Fig. 2(c) is covered by one of the following three S-invariants:

$$y_1 = [1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0],$$

$$y_2 = [0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0],$$

$$y_3 = [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1].$$

Since the support of each of them contains exactly one token, we have, for each p, $M(p) \leqq 1$ for any marking M reachable from M_0 and $y * M = y * M_0 = 3$ where $y = y_1 + y_2 + y_3$.

[4]Here, $\cdot t$ ($t \cdot$) denotes the set of input (output) places of transition t, and $\cdot p$ ($p \cdot$) denotes the set of input (output) transitions of place p.

[5]Here $P(T)$ is the set of APN places (transitions).

IV. Detection of Static Deadlocks in Ada Programs

We now define static deadlocks and discuss algorithms for their automatic detection. The method for Ada deadlock analysis consists of the following basic steps (see Fig. 3):

1) translation of an Ada program into an Ada net, as discussed earlier,

2) detection and removal of a class of deadlocks called inconsistency deadlocks, and

3) finally, detection of a class of deadlocks called circular deadlocks.

A. Static Deadlocks

There is no commonly accepted definition of a static deadlock in a distributed programming environment. Traditional deadlock detection techniques (reachability tree or state graph analysis), using exhaustive search, label every potentially reachable deadlock as a "static deadlock." However, such techniques can generate some extra information. For example, a variable index loop containing communication statements always creates a potentially reachable deadlock. Since the number of times the loop executes is determined at execution time, any number of loop iterations is assumed possible in this paper. Therefore, here we do not consider such a deadlock to be a static deadlock. Since Ada nets do not distinguish between loops with variable and constant indexes, potentially reachable deadlocks caused by Loop statements would not be reported by our static deadlock detection system. We would report any other deadlocks that can be predicted by static analysis of a program's source code. In the following sections, we define two classes of static deadlocks: inconsistency deadlocks and circular deadlocks. We consider SD = ID ∪ CD where SD is the set of all static deadlocks for a program and ID and CD are the sets of all inconsistency and circular deadlocks, respectively, for the program.

B. Inconsistency Deadlocks

Some potentially reachable Ada program static deadlocks can be easily detected using only structural analysis of Ada nets. We call these deadlocks inconsistency deadlocks. In this section, we present an algorithm to detect such deadlocks.

An Ada program is called *static executable* when its corresponding Ada net has t_2 (par-end transition) potentially enabled for M_0. We will refer to a set of linearly independent T-invariants x of an Ada net, such that $x(t) \geq 0$ for each t in the net and $x(t_1) = x(t_2) = 1$ where t_1 is the par-begin transition and t_2 is the par-end transition, as a set of *Ada T-invariants* of the net.

Since there is always an Ada T-invariant in an Ada net corresponding to a static execution path of the Ada program, it can be seen that the following holds.

Theorem 3: A necessary condition for an Ada program Ap to be static executable is that its corresponding Ada net APN $< P, T >$ has at least one Ada T-invariant.

By Theorem 3, if an Ada net APN $< P, T >$ modeling

Fig. 3. Overview of the static deadlock detection method.

an Ada program Ap does not have an Ada T-invariant, then Ap is not static executable. Moreover, if some transition does not belong to the support of the linear combination of all Ada T-invariants of APN, then the statement of Ap represented by the transition does not belong to any successful execution path. Such a situation indicates that Ap has at least one static deadlock, caused by a communication statement (or a group of communication statements). For example, consider the following program fragment:

```
TASK1:                    TASK2:
TASK2.ENTRY2;             if cond1 then accept ENTRY2;
TASK2.ENTRY1;                      else accept ENTRY1;
```

The net corresponding to this Ada program would not have Ada T-invariants. Thus, the program is not static executable (it has a static deadlock), as can easily be checked. Note that removal of some communication statements (e.g., "TASK2.ENTRY1" and "accept ENTRY1") could make the fragment static executable since then every statement in the modified program would belong to at least one static execution path. Then, the support of a linear combination x' of all Ada T-invariants of the Ada net (for the modified program) would cover every transition in the net. Since the detection of these deadlocks depends on a test for T-invariants, we refer to them as *inconsistency deadlocks*. Now, we present an algorithm that detects inconsistency deadlocks in an Ada program Ap (with its corresponding Ada net APN) and reports the minimal number of communication statements that need to be removed in order to remove the deadlock potential. This information is useful to the programmer in determining the actual causes of the deadlocks.

Algorithm INCONSISTENCY_DETECT

Input:

A—the incidence matrix of an Ada net APN;

k—the number of communication statements in the Ada program Ap;

S—the set of all possible subsets of communication statements of Ap.

Output:

The set of communication statements of Ap "causing" inconsistency deadlocks in Ap.

Method:

1) loop1: For $i = 0$ to k do
2) loop2: For each element s of S such that $|s| = i$ do
3) $A' := A$;
4) Update A' to reflect the removal of all communication arcs associated with communication statements in s;
   For any entry, if the three columns of A' corresponding to p_{mcp}, p_{send}, and p_{ack} each have all zero entries then remove the three columns from A';
   (* some communication channels become disconnected *)
   Else if not every column of A' has both positive and negative entries then return to step 2);
   (* some communication channels have lost all their Accepts or Entry calls (but not both) *)
5) Find the support $\|x'\|$ of any linear combination of Ada T-invariants of the Petri net APN′ corresponding to A';
6) If every transition of APN′ belongs to $\|x'\|$, then report s to the user and exit the algorithm;
   (* Here, s is the subnet of communication statements causing the inconsistency deadlock. *)

end loop2;
end loop1;

As an example, consider a modified version of the classical consumer–producer problem:

```
task body Producer is
  begin
    while cond1 loop
      Buffer.produce;
      Buffer.msg_send;
      accept msg_ack;
      accept msg_ack;
    end loop;
  end;
```

```
task body Buffer is
  begin
    while cond2 loop
      select
        when
          buffer not full => 

          accept produce;
          Producer.msg_ack;
          accept msg_send;
      or
        when
          buffer not empty =>
            accept consume;
      end select;
    end loop;
  end;
```

```
task body Consumer is
  begin
    while cond3 loop
      Buffer.consume;
    end loop;
  end;
```

Fig. 4. Ada net for a producer-consumer program with inconsistency deadlock.

Note that after Producer supplies an item to Buffer, both tasks intend to exchange messages (msg_send and msg_ack). Unfortunately, the programmer made a mistake in coding this exchange. The result is such that whenever Producer sends an item to Buffer, the program deadlocks.

The Ada net for the program is shown in Fig. 4. The first iteration of loop1 in the algorithm INCONSISTEN-

CY_DETECT reveals two Ada T-invariants:

$$x_1 = (1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$$

and

$$x_2 = (1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0).$$

Since some transitions do not belong to $\|x'\|$ where $x' = x_1 + x_2$, the program has an inconsistency deadlock. During the second iteration of loop1 (with $|s| = 1$), we are trying to remove the deadlock. After we remove the communication arcs associated with the Producer's first Accept statement, the algorithm reports three Ada T-invariants:

$$x_1 = (1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),$$

$$x_2 = (1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),$$

and

$$x_3 = (1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1).$$

Let $x' = x_1 + x_2 + x_3$. Now, $\|x'\|$ includes every transition of the net. Therefore, the Accept statement is reported as a "cause" of the inconsistency deadlock. Presumably, the programmer would then update the program by removing the extra Accept statement. The updated version is as follows:

```
task body Producer is
  begin
    while cond1 loop
      Buffer.produce;
      Buffer.msg_send;
      accept msg_ack;
    end loop;
  end;

task body Buffer is
  begin
    while cond2 loop
      select
        when
          buffer not full => 
          accept produce;
          Producer.msg_ack;
          accept msg_send;
      or
        when
          buffer not empty => 
            accept consume;
      end select;
    end loop;
  end;

task body Consumer is
  begin
    while cond3 loop
      Buffer.consume;
    end loop;
  end;
```

The reader may note that this new version still suffers from a deadlock. This deadlock, however, is not an inconsistency deadlock; it is a circular deadlock and will be treated in the next section. The Ada net for the above program is illustrated in Fig. 5.

C. Circular Deadlocks

Even if an Ada program is free of inconsistency deadlocks, some statements may not belong to any static execution path of the program. In this section, we discuss potentially reachable static deadlocks in Ada programs free of inconsistency deadlocks.

Definition: A *circular deadlock* is a deadlock (in a program free of inconsistency deadlocks) due to a set of communication statements, each in a separate task, mutually suspending each other and, thus, also the control flows of their respective tasks.

A typical example of a circular deadlock is the deadlock that can occur in the classical dining philosophers problem. The circular deadlock occurs after each philosopher has picked up his left fork, but no philosopher has picked up his right fork.

An Ada net dicircuit (directed circuit or closed directed path), which represents a circular deadlock, is referred to as a *C-type dicircuit* (dicircuit of circular type). For example, the only circular deadlock for the program of Fig. 5 is represented by the C-type dicircuit (t_{18}, t_{19}, t_{20}, t_9, t_{10}, t_{11}). A C-type dicircuit possesses the following properties.

1) The dicircuit is composed of directed paths in process-subnets (segments), which start and end with communication transitions (e.g., [t_{18}, t_{19}, t_{20}] and [t_9, t_{10}, t_{11}] in Fig. 6). The starting transition of such a segment is referred to as a *ready transition*, and the ending transition is a *last-executable transition*. The last-executable transition of one segment is connected to the ready transition of another segment through a communication place (e.g., in Fig. 5, last-executable transitions t_{20} and t_{11} are attached to ready transitions t_9 and t_{18}, respectively). Thus,

Fig. 5. Ada net for the producer-consumer program with no inconsistency deadlocks.

Fig. 6. Loop transformation.

a C-type dicircuit has a set of ready transitions and a set of last-executable transitions.

2) A set of communication places representing at least two Ada entries belongs to the C-type dicircuit (e.g., p_{10} represents entry msg_ack and p_7 represents entry msg_send in Fig. 5). Otherwise, the dicircuit is called a *single-entry* dicircuit and is not a C-type dicircuit.

3) If more than one segment of a process-subnet belongs to the same dicircuit, such a dicircuit is called a *multi-sequential-part* dicircuit and is not considered a C-type dicircuit. It is easy to show that such a dicircuit does not represent any circular deadlock. For if it did, then in order to "enable such a dicircuit," we would need to have a marking which would enable all the ready transitions of the dicircuit. Since one of the process-subnets has more than one ready transition, reachability of such a marking would contradict the safeness of Ada nets (Theorem 1).

Theorem 4: A sufficient condition for an Ada net to have a C-type dicircuit is that there be a minimal S-invariant y such that the places of its support $\| y \|$ have no tokens at M_0 and do not form either a single-entry or multi-sequential-part dicircuit.

Proof: If the support $\| y \|$ does not have any tokens at M_0, then the number of tokens in $\| y \|$ is 0 for any marking reachable from M_0 [6]. If the places of $\| y \|$ do not form either a single-entry or a multi-sequential-part dicircuit, then the places in $\| y \|$ form a C-type dicircuit. Q.E.D.

The condition of Theorem 4 can be used to detect every circular deadlock in a program that has only one Entry call and one Accept statement per entry. The example of Fig. 5 shows an Ada net for which the reporting of S-invariants provides complete information about its C-type dicircuits.

Finding circular deadlocks in general is a more complicated task. We detect C-type dicircuits by isolating them from any other possible dicircuits. The algorithm CIRCULAR_DETECT iteratively removes vertices (transitions or places) which are not part of any C-type dicircuit. To do so, the algorithm first removes the dicircuits representing iterative statements, such as Loop statements. The procedure for that is illustrated in Fig. 6. We remove a dicircuit representing a Loop statement by replacing place p_l (representing the Loop statement) with two places p_k and p_m, as shown in Fig. 6. Since no initially marked places can be part of any C-type dicircuit, these places are removed. As a result, some transitions can then be removed, and this in turn leads to the removal of more places and transitions. We keep removing such vertices until every vertex belongs to at least one dicircuit. Finally, all single-entry and multi-sequential-part dicircuits are eliminated, leaving only the C-type dicircuits.

Now, we are ready to present the algorithm CIRCULAR_DETECT, which finds and reports all circular deadlocks in an Ada program.

Algorithm CIRCULAR_DETECT

Input:

A—the incidence matrix of an Ada net APN.

Output:

The set of minimal supports of submarkings enabling all C-type dicircuits in APN. Communication places are not included in the supports.

Method:

1) Disconnect the dicircuits representing Loop statements.
2) Remove all initially marked places from the incidence matrix A. The result is a new incidence matrix A' for some new Ada net APN'. Perform procedure PRUNE_TREE.
3) Perform procedure ALL_IN_DICIRCUITS.
4) Trace all the dicircuits in A' and place them into a set C. Remove all single-entry and multi-sequential-part dicircuits from C. For each dicircuit in C, find the support of the minimal submarking enabling the dicircuit and place it into a set R (if it is not already there).

Procedure PRUNE_TREE

(* The following procedure makes a Petri net connected *)

Repeat the following step until every place and transition of APN' has at least one input and at least one output arc.

If any place or transition of APN' has no input or output arcs, then remove it from APN'. The new incidence matrix of APN' is still denoted by A'.

Procedure ALL_IN_DICIRCUITS

(* The following procedure makes a Petri net strongly connected *)

1) If APN' has n transitions and m places, then create an $(m + n) \times (m + n)$ matrix L for all vertices of APN'. Initially, L has all entries set to 0. If there exists an arc from a transition t to a place p in APN', then set $L(t, p)$ to 1. By symmetry, an arc from a place p to a transition t in APN' implies that we set $L(p, t)$ to 1.
2) For each entry of L, perform Dijkstra's shortest-path algorithm to derive a new matrix L [9].
3) For all vertices v_i (places and transitions), if there is no vertex v_j such that $L(v_i, v_j) > 0$ and $L(v_j, v_i) > 0$ and APN' has more than four transitions and four places, then remove v_i from A' (from APN'), perform procedure PRUNE_TREE, and go to step 1) of this procedure.
4) Return the resulting Ada net APN'.

A detailed stepwise discussion and analysis of the algorithm can be found in [5]. The following examples illustrate the algorithm CIRCULAR_DETECT.

Consider the Ada net illustrated in Fig. 5. Step 1) disconnects Loop statements represented by places p_{25}, p_{16}, and p_{14}. Step 2) removes all original places and transitions of the net, except for the ones with thickly drawn arcs (in Fig. 5) as their inputs and outputs. Step 3) has no effect in this example since every place (transition) left has at least one input and at least one output thickly drawn arc. Step 4) traces the only C-type dicircuit of the net: $(t_{18}, t_{19}, t_{20}, t_9, t_{10}, t_{11})$. When the algorithm terminates, set R has only one element, the set $\{ p_{28}$ and $p_{20} \}$ (representing the communication statements that would block if the deadlock were reached).

As another example, consider the following static-deadlock-free Ada fragment Ap:

TASK1:	TASK2:	TASK3:
TASK2.ENTRY21	TASK3.ENTRY31	accept ENTRY31
accept ENTRY11	accept ENTRY21	TASK1.ENTRY11
	TASK3.ENTRY31	accept ENTRY31

Fig. 7(a) illustrates the Ada net APN corresponding to Ap. After performing steps 1), 2), and 3) of CIRCULAR_DETECT, we obtain the Ada net APN' shown in Fig. 7(b). Note that the dashed places are not a part of the output of step 3); they are used in step 4) to find the ready submarkings of all C-type dicircuits of the Ada subnet in Fig. 7(b). The elements of set C are as follows.

Single-Entry Dicircuits:

$(t_8, t_9, t_{10}, t_{11}, t_{17}, t_{18})$,
$(t_8, t_9, t_{10}, t_{11}, t_{13}, t_{14}, t_{15}, t_{16}, t_{17}, t_{18})$,
$(t_8, t_9, t_{10}, t_{11}, t_{13}, t_{14})$.

Multi-Sequential-Part Dicircuit:

$(t_{13}, t_{14}, t_{15}, t_5, t_6, t_{16}, t_{17}, t_{18}, t_8, t_9, t_{10}, t_{11})$.

The Only C-Type Dicircuit c':

$(t_{16}, t_{17}, t_{18}, t_8, t_9, t_{10}, t_4, t_5, t_6)$.

The C-type dicircuit c' is illustrated in Fig. 7(c) along with the minimal support of the submarking (dashed circles $\{ p_{12}, p_{17}$, and $p_{26} \}$) which would enable c'. From this example, we can see that not every circular deadlock is of interest to a programmer. Only the ones that are actually reachable by static execution paths should be detected and reported. Such circular deadlocks are referred to as *potentially reachable circular deadlocks*. Structural analysis (using structural restrictions and Petri net invariants) is necessary, but not sufficient for detecting potentially reachable circular deadlocks. Dynamic (reachability tree) analysis must be used too. The algorithm CIRCULAR_REACHABLE, presented next, reports only the potentially reachable C-type dicircuits of an Ada net.

Algorithm CIRCULAR_REACHABLE

Input:

An Ada program Ap and its corresponding Ada net APN;

X—the set of Ada T-invariants of APN;

R—the set of the minimal supports of submarkings enabling the C-type dicircuits of APN (the output of CIRCULAR_DETECT).

Output:

R'—the set of the minimal supports of reachable

Fig. 7. Example for the algorithm CIRCULAR_DETECT.

submarkings enabling the C-type dicircuits of APN.

Method:

(* T-invariant-directed Reachability Graph Generation *)

0) If R is empty, then report that Ap is circular deadlock free and exit.

1) LOOP1: For each x in X perform
(* Select T-invariant as a firing count vector *)
$M := M_0$; $x' := x$; $R' := \varnothing$; STACK $:= \varnothing$;

2) LOOP2: Repeat
(* Perform selective Reachability Graph Generation *)
If no t_j with $x'(t_j) > 0$ is enabled at M then do
(* if $x' = 0$ then static execution path in Ap is found; *)
if STACK is empty then exit LOOP2;

```
            retrieve the last saved marking of APN,
              firing count vector, and transition in
              conflict, and store them into M, x', and
              t_k, respectively;
          end do;
          Else if more than one transition is enabled
          at M then do
            assign one of those transitions to t_k; for
            every other transition in conflict
              push current values of M and x' onto
              STACK;
          end do;
          Else assign the enabled transition of APN
          to t_k;
          Fire transition t_k to get a new marking M;
          Let x'(t_k) = x'(t_k) - 1;
          If the current M was considered before then
          do
            if STACK is empty then exit LOOP2;
            retrieve the last saved marking of APN,
              firing count vector, and transition in
              conflict, and store them into M, x', and
              t_k, respectively;
          end do;
          Else if any of the elements of R is a subset
          of the support of M
            and the element does not belong to R'
            then add the element to R';
    END LOOP2;
    END LOOP1;
```

An Ada program that is free of inconsistency deadlocks and free of potentially reachable circular deadlocks is referred to as a *static-deadlock-free program*. Even though the program corresponding to the net in Fig. 7(a) has a circular deadlock, it is a static-deadlock-free program since the circular deadlock reported by the algorithm CIRCULAR_DETECT is not reachable (as would be reported by the algorithm CIRCULAR_REACHABLE).

The algorithm CIRCULAR_REACHABLE is more efficient than "blind" tracing of the Petri net's reachability graph. CIRCULAR_REACHABLE uses the knowledge of the Petri net's Ada T-invariants in order to reduce the time complexity of Petri net dynamic (reachability) analysis. As an example, consider the Ada net illustrated in Fig. 5. As noted earlier, the Ada net is free of inconsistency deadlocks and has three Ada T-invariants:

$$x_1 = (1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),$$

$$x_2 = (1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0),$$

and

$$x_3 = (1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1).$$

T-invariant x_1 indicates that the program may terminate (i.e., the Petri net may cycle) without any communication between the tasks Producer, Buffer, and Consumer. Ada T-invariants x_2 and x_3 correspond to the nets shown in Fig. 8(a) and (b), respectively. Generating the reachability graph for the net in Fig. 5 using x_1 or x_2 as a guiding firing count vector does not lead to a marking whose support would include both p_{28} and p_{20}, the places previously identified by the algorithm CIRCULAR_DETECT. On the other hand, when the algorithm generates the reachability graph using x_3 as the guiding firing count vector, we eventually reach marking

$$M = (0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0).$$

Since M's support includes both p_{28} and p_{20}, CIRCULAR_REACHABLE reports the potentially reachable C-type dicircuit. Presumably, the user would then eliminate this deadlock by interchanging the order of the statements "Producer.msg_ack" and "accept msg_send" in the task Buffer. If we were to use "blind" reachability graph analysis for the Ada net in Fig. 5, then we would generate a reachability graph of 82 vertices. In contrast, our algorithm generates a 2-node reachability graph when using x_1 as the guiding T-invariant, a 19-node graph when using x_2 as the guiding T-invariant, and a 16-node graph when using x_3 as the guiding T-invariant. Thus, the static deadlock detection system is more time and space efficient than unrestricted reachability graph analysis. In fact, for an Ada program Ap modeled by an Ada net APN, the more linearly independent Ada T-invariants APN has, the more efficient the analysis of APN by CIRCULAR_REACHABLE. The number of T-invariants of APN depends on the number of conflict statements (If, Select, and Case) that have nested communication statements.

The analysis can also be considered to be modular since a user interested in some particular statements may not have to deal with the whole reachability graph of the Ada net modeling his system. Only the Ada T-invariants whose supports include the transitions which represent the statements of interest need to be used in the algorithm CIRCULAR_REACHABLE.

V. Summary

In this paper, we have described a static deadlock detection method for Ada tasking programs. First, an Ada program is translated into a Petri net, called an Ada net, which properly models the communication patterns and control flow of the source program. Then, the Ada net is

(a)

(b)

Fig. 8. Ada net determined by the Ada *T*-invariants.

analyzed for the existence of static deadlocks based on structural analysis (using *T*-invariants, *S*-invariants, and structural restrictions) and dynamic analysis (using the reachability graph). The overall structure of the proposed method (system) is illustrated by the flowchart in Fig. 3. Here, static deadlocks are considered either inconsistency or circular deadlocks. Algorithms for the detection of both classes were presented and illustrated by examples. Due to the use of both structural and dynamic Petri net techniques, our static deadlock detection method is relatively time (space) efficient and can support modular analysis of Ada tasks.

References

[1] I. C. Pyle, *The Ada Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[2] N. Gehani, *An Advanced Introduction Including Reference Manual for the Ada Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[3] R. Taylor, "A general purpose algorithm for analyzing concurrent programs," *Commun. ACM*, vol. 26, no. 5, pp. 362-376, May 1983.

[4] S. German, "Monitoring for deadlocks and blocking in Ada tasking," *IEEE Trans. Software Eng.*, vol. SE-10, Nov. 1984.

[5] B. Shenker, "Using Petri nets for automated detection of static deadlocks in Ada programs," Master's thesis, Dep. Elec. Eng. Comput. Sci., Univ. Illinois, Chicago, IL, Sept. 1985.

[6] T. Murata, "Modeling and analysis of concurrent systems," in *Handbook of Software Engineering*, C. R. Vick and C. V. Ramamoorthy, Eds. New York: Van Nostrand Reinhold, 1983, ch. 3.

[7] S. M. Shatz and W. K. Cheng, "A Petri net framework for automated static analysis of Ada tasking behavior," *J. Syst. Software*, vol. 8, pp. 343-359, Dec. 1988.

[8] J. Martinez and M. Silva, "A simple and fast algorithm to obtain all invariants of a generalized Petri net," in *Application and Theory of Petri Nets (Informatik-Fachberichte 52)*. New York: Springer-Verlag, 1982, pp. 301-310.

[9] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*. Potomac, MD: Computer Science Press, 1977.

[10] R. Taylor and T. Standish, "Steps to an advanced Ada programming environment," *IEEE Trans. Software Eng.*, vol. SE-11, Mar. 1985.

[11] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato, "Modeling the Ada task system by Petri nets," *Comput. Lang.*, vol. 10, no. 1, pp. 43-61, 1985.

Tadao Murata (S'62-M'66-SM'77-F'85) received the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana.

He is presently a Professor of Electrical Engineering and Computer Science at the University of Illinois, Chicago. During occasional leaves of absence from the University of Illinois, he has taught at the University of California, Berkeley, and Tokai University, Tokyo, Japan, and was invited to visit Petri's Institute at GMD mbH in Germany and several other research institutes and universities in Europe. His current research interests include applications and theory of Petri nets, concurrent computer systems, and data flow and parallel computations. In these areas he has published extensively and been awarded several National Science Foundation research grants since 1975. Prior to that he worked in the area of circuits, systems, and applied graph theory.

Dr. Murata has served on the U.S. National Academy of Sciences Computer Science and Technology Board panels. He is an Editor of the IEEE Transactions on Software Engineering and served as the General Chairman of the International Workshop on Petri Nets and Performance Models, held August 24-26, 1987. He is a member of the Association for Computing Machinery, EATCS, IECE, and the Information Processing Society of Japan. He is listed in *Who's Who in Engineering* and *Who's Who in America*.