9-1990

An Approach for Minimizing Spurious Errors in Testing ADA Tasking Programs

N. Mansouri
Syracuse University, Department of Engineering and Computer Science, namansou@ecs.syr.edu

Amrit L. Goel
Syracuse University, algoel@syr.edu

SU-CIS-90-33

An Approach for Minimizing Spurious Errors in Testing ADA Tasking Programs

Nashat Mansour and Amrit L. Goel

September 1990

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

SU-CIS-90-33

An Approach for Minimizing Spurious Errors in Testing ADA Tasking Programs

Nashat Mansour and Amrit L. Goel

September 1990

School of Computer and Information Science
Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

(315) 443-2368

AN APPROACH FOR MINIMIZING SPURIOUS ERRORS IN TESTING ADA TASKING PROGRAMS

Nashat Mansour
School of Computer and Information Science

Amrit L. Goel
Department of Electrical and Computer Engineering
School of Computer and Information Science

Syracuse University

September 1990

ABSTRACT

We propose an approach for detecting deadlocks and race conditions in Ada tasking software. It is based on an extension to Petri net-based techniques, where a concurrent program is modeled as a Petri net and a reachability graph is then derived and analyzed for desired information. In this approach, Predicate-Action subnets representing Ada programming constructs are described, where predicates and actions are attached to transitions. Predicates are those found in decision statements. Actions involve updating the status of the variables that affect the tasking behavior of the program and updating the Read and Write sets of shared variables. The shared variables are those occurring in sections of the program, called concurrency zones, related to the transitions. Modeling of a tasking program is accomplished by using the basic subnets as building blocks in translating only tasking-related statements and connecting them to produce the total Predicate-Action net model augmented with sets of shared variables. An augmented reachability graph is then derived by executing the net model. Deadlocks and race conditions are detected by searching the nodes of this graph. The main advantage offered by this approach is that the Predicate-Action extension of the net leads to pruning infeasible paths in the reachability graph and, thus, reducing the spurious error reports encountered in previous approaches. Also, this approach enables a partial handling of loops in a practical way. Implementation issues are aslo discussed in the paper.

Index terms: Ada tasking, concurrent programs, deadlock detection, Petri net applications, race conditions, software testing, static analysis.

1. INTRODUCTION

Software testing is an important phase in the development lifecycle since it has an important effect on the reliability of the software in operation. Testing is a systematic, though nonformal, validation method that aims at gaining confidence in the correctness of a program. It is costly and difficult for sequential as well as concurrent software [Hausen 84, Tai 89b].

The growing use of concurrent computers, centralised, parallel or distributed, for solving a variety of problems, accentuates the need for more research in the area of testing concurrent programs. In particular, there is a need for developing automated tools to reduce the complexity and the effort involved. Research in this area is still in its early stages. Testing concurrent software is more difficult than sequential software testing because in a concurrent program a number of processes are considered. These processes may run, on the target machine, on several processors. They communicate and synchronize with each other in order to produce a total solution. In such a concurrent processing environment, a number of factors contribute to the complexity of testing the software. The main factors are different processor speed, unpredictable scheduling of processes and nondeterministic constructs in languages used for asynchronous processing. These factors lead to nondeterministic sequence of execution and cause the reproducibility or replay problem [Tai 85, 89a, 89b], where different executions of the program may yield different results. Moreover, if shared variables are allowed in the programming language, concurrent processes may enter a race condition.

In addition to the sequential computational and domain errors [Howden 76], concurrent programs may contain synchronization and concurrency errors and anomalies. The most important of these are deadlocks and data-usage anomalies, namely potential race conditions on shared global variables. The term deadlock is used in this paper and in most of the testing literature to represent all kinds of infinite wait or blockage of processes which prevent a program from normal termination. A race condition occurs when two or more processes nondeterministically access shared data and at least one process is updating the data. Other anomalies which can be detected by static analysis of parallel programs have been discussed in [Taylor 80] and [Bristow 79].

The approaches for testing concurrent programs can be divided into static analysis and dynamic analysis. No actual program execution takes place in static analysis. Instead, the program code is transformed into a model and the model is then analyzed for detecting specific error states, perhaps, in addition to other useful information. For example, Taylor [Taylor 83a] models a program with flowgraphs, whereas Shatz [Shatz 88a] translates a program into a Petri net. Static analysis has the advantage that it is independent of the characteristics of the target machine and can be performed in relatively inexpensive and convenient environments. However, it suffers from a lack of program semantics that may lead to spurious error reports. In dynamic analysis, the program is executed on the target computer with selected input test data, and its behavior and output are examined. The insertion of debugging statements may alter the program behavior in dynamic analysis. This is referred to as the probe effect [Gait 86]. Static and dynamic analyses may be integrated to exploit the complementarities in both approaches [Osterweil 84]. A small number of tools have been reported for dynamic testing [Tai 89a] and static analysis [Shatz 89, 88a, McDowell 88].

The major testing techniques are illustrated in the next section. They point out the considerable difficulty in developing practical testing methodologies for concurrent software. These approaches suffer from several shortcomings. In particular, static analysis approaches, that have been based on the program's syntax, may give rise to spurious error reports because they fail to inhibit infeasible paths. Also, it does not seem that a practical method has been found to handle conditional loops when they include synchronization statements. Conditional loops may result in a very large program state space, which is impractical to analyze.

The automatable testing approach presented in this paper is based upon static analysis of concurrent software using a Petri net model. It is concerned with the tasking behavior of Ada concurrent programs, namely with the detection of deadlock errors and data-usage anomalies.

Like other static analysis approaches, this work assumes that the sequential behavior of individual processes is tested by means of sequential techniques independently of testing the concurrency features. The model of communication and synchronization in Ada [DoD 81] is the rendezvous type, which is also adopted in a wide class of message-passing languages such as CSP [Hoare 78]. In this paper, Ada is chosen as a representative of this class of language notations for concurrent systems.

Our approach is based upon Petri net modeling and reachability analysis, which has been previously used for deadlock detection [Shatz 88a, Murata 89a, Goel 90]. However, it extends the Petri net framework in order to reduce spurious error reports encountered in the previous static analysis approaches and to add other analysis capabilities. The model employed in our approach is an augmented high level Petri net called Augmented Predicate-Action Net (APrAN). The analysis is performed on a reachability graph augmented with sets of shared variables. APrAN allows the inclusion of program semantics in the analysis. This alleviates the problem of infeasible paths encountered in traditional Place-Transition Petri net-based static analysis, and helps in the detection of synchronization errors caused by incorrect predicates in decision statements. The extended model also allows a simple and useful way for handling finite conditional loops containing tasking statements, which have not been dealt with in the previous approaches. APrAN is augmented with data usage and hence anomalies of race conditions on shared variables can be detected. All these enhancements and additions are offered in a unified and coherent framework. Implementation notes are also included.

The paper is organized as follows. The next section presents a brief survey of most of the known testing techniques. Section 3 introduces Petri nets and Ada tasking constructs. In Section 4, The APrAN-based approach is presented and illustrated by an example. In Section 5, implementation issues are presented. Section 6 contains conclusions.

2. PREVIOUS WORK

A number of approaches have been proposed for testing concurrent programs. Most of them have used Ada's rendezvous as a model for synchronization and communication. These testing approaches are either static, which are based upon code analysis, or dynamic, which require actual program execution. Dynamic analysis usually refers to debugging techniques also, but such techniques are not considered here.

Some of the issues and difficulties encountered in testing concurrent programs are the same as those for sequential programs, such as the combinatorial explosion problem in path selection, whereas others are specifically related to concurrent programs, such as the reproducibility problem. The main issues in dynamic testing of concurrent software are forcing the execution of a synchronization sequence to address the reproducibility problem, the selection of the synchronization sequence, the selection of input data, the management of the combinatorial explosion problem in selecting sequences and test data and the measurement of test coverage. The main issues and difficulties in static analysis are the reduction in the size of the model used to represent the synchronization behavior of the program, the reduction in the time complexity required by the analysis which has been shown to be NP-complete [Taylor 83b], the handling of conditional loops which aggravate the combinatorial problem in statically testing parallel programs, the elimination of infeasible paths from the program's state space and hence the prevention of spurious error reports, and the handling of dynamic operations such as recursion and dynamically-created objects related to synchronization.

Most of the dynamic testing work has been based on deterministic execution testing (DET) [Tai 89a, 87, 86, 85, Carver 86]. The DET approach is geared towards solving the reproducibility problem. An input test case in DET consists of data, x, and a synchronization sequence, S. In the language-based implementation, the program is transformed by inserting statements, which pass synchronization requests to a control task, to force the execution of the program according to S. The output is correct if it is valid with respect to specifications and if

S proves feasible. In [Taylor 86], structural testing is proposed based on a concurrency state graph derived by static analysis of the program. Several coverage metrics are described and it is suggested that only the selection of some interesting paths in the concurrency graph may be practical. The use of a controllable scheduler to force the execution of a path is proposed and the difficulties in coverage measurement and test data generation are also discussed. Weiss [Weiss 88] has suggested a formal framework for the study of testing. To reduce the number of tests to a practical level, the assignment of levels of importance to shared variables and intertask communication is proposed. Serializations for sufficiently important shared variables and communication statements can then be generated for testing.

The first static analysis approach appeared in [Taylor 83a]. This approach is based on flowgraph models of concurrent tasks. A directed graph of concurrency states is then derived from the flowgraphs where a state represents the control state of the parallel tasks, including synchronization information. A path in the graph, called a concurrency history, represents a sequence of synchronization events. Deadlock errors are detected by searching the concurrency state graph for terminal states occurring while some tasks are still active. With some post-processing, the anomaly of concurrent updating of shared variables may be revealed. In [Young 88], this static concurrency analysis is combined with symbolic execution so that the concurrency analysis acts as a path selection mechanism for symbolic execution and the symbolic execution prunes infeasible paths in the concurrency graph

A similar analysis approach to that of Taylor's appears in [Shatz 88a, 89] but within a Petri net framework. In [Shatz 88a], a procedure and its implementation are described for translating a concurrent Ada program to a Petri net model. A separate 'general-purpose' tool [Morgan 87] is then employed to derive the reachability graph, which represents all possible synchronization sequences for the Petri net. This tool is also used to analyze the reachability graph. The analysis results include information about deadlock states and the tasking behavior of the program, such as the maximum number of rendezvous requests queued for a task and the rendezvous that can occur while a task is waiting to rendezvous with another task. Within the Petri net framework, [Murata 89a] presents algorithms based on structural and reachability analysis to detect inconsistency and circular deadlocks. A concurrent Ada program is translated to a Petri net model. Then place and transition invariants of the Petri net and their supports are computed. This structural information is used to guide a selective generation of the reachability graph leading to reduction in the time and space required for deadlock detection.

Other approaches for static analysis of concurrent programs have recently appeared in the literature. A task interaction graph (TIG) is proposed in [Long 89] as a model for tasks. A TIG represents a task as a set of regions and a set of interactions between regions, and thus its division of a task is based on interactions not on control flow. A task interaction concurrency graph (TICG) is then derived from the TIGs of tasks, where a vertex represents a state and an edge represents the start and end of a rendezvous. The number of states in a TICG has been found to be smaller than that for control flow-based models for a number of programs. In this approach, deadlock is detected if a task is waiting for a rendezvous and no other task is able to rendezvous at a certain point. [McDowell 89, 88] derives a reduced state concurrency history graph (CHG) from the control flowgraphs of the program, where some states represent merged sets of states. Merging is possible when parallelism in the program is a result of parallel execution of multiple copies of the same task. A state in CHG represents a set of task states, values of shared variables and local variables that derive their values directly from the synchronization operations. In this approach, deadlock and the anomaly of parallel update of shared variables can be detected. In [Wileden 88] and [Avrunin 86] a different static analysis approach is taken, which is based on constrained expressions. A constrained expression corresponds to strings of a language where these strings represent possible program behavior, such as a rendezvous request. In this approach, program design is translated into constrained expressions.

To reduce the number of infeasible paths, Carver and Tai [Carver 88] suggest the derivation of feasibility constraints from the syntactic as well as the semantic information of a concurrent program. These constraints restrict the ordering of synchronization events and hence yield a better approximation of the set of feasible synchronization sequences. The constraints are derived from semantics-graphs of tasks. A semantics graph represents control flow in addition to 'relations' between variables, where these relations extract semantics information from predicates in decision statements and loops. Based on this approach, deadlock detection is expected to contain less spurious error reports.

Symbolic execution is used in the formal verification of Ada tasking programs in [Dillon 88a, 88b] and [Harrison 88]. Most of the issues and difficulties which have been discussed for other approaches above are also relevant for symbolic execution. Dillon [Dillon 88a] highlights issues such as exponential growth in the size of the execution tree, possible infeasible paths when loop invariants do not capture the relation between variables in different tasks and infinite tree size if loops contain communication statements. Another method for symbolic execution of concurrent programs is proposed in [Ghezzi 89] and [Morasca 89]. It is based on a Petri net formalism, called Environment/Function (EF) nets. Symbolic execution algorithms are presented. The modeling power of EF nets and the utility of the algorithms are discussed and illustrated by means of a case study.

3. PRELIMINARIES

In this section, a description is given of some basic concepts utilized throughout this paper. The description includes Petri nets, the rendezvous model of synchronization and relevant Ada programming constructs.

3.1 PETRI NETS

A system can be modeled by a Petri net (PN), which becomes a mathematical representation of the system [Murata 89a, Peterson 81]. Analysis of the Petri net, then, yields information about the structure and the behavior of the system. The type of Petri nets employed throughout this paper is the Place-Transition (PT) type. PT nets are defined below. Description of their analysis is integrated into subsection 4.3, where the analysis of the augmented model used is presented.

Definition: A PT net is a 5-tuple, PN = (P, T, I, O, M_0), where

$P = \{p_1, ..., p_m\}$ is a finite set of places,

$T = \{t_1, ..., t_n\}$ is a finite set of transitions,

$I \subseteq PxT$ is a set of transition input arcs,

$O \subseteq TxP$ is a set of transition output arcs,

M_0: P --> {0, 1} is the initial marking,

$P \cap T = \emptyset$ and $P \cup T = \emptyset$.

For the purpose of this paper, it is assumed that the weight on every arc is 1 and that the maximum capacity of a place is 1. A graphical representation is depicted in Figure 3.1(a), where bars represent transitions and circles represent places.

Enabling Conditions

• A transition t_i is enabled if each of its input places contain a token, i.e.

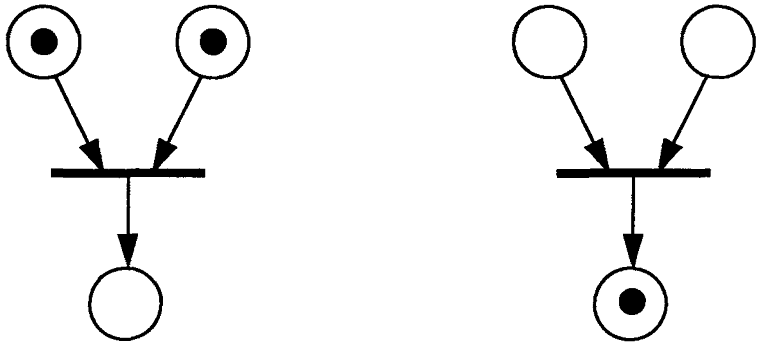

(a) Before (b) After

Figure 3.1 A PT net before and after firing a transition.

for all $p_j \varepsilon I(t_i)$, $M(p_j) = 1$,

for all $p_j \varepsilon O(t_i)$, $M(p_j) = 0$

<u>Transition Firing Rules</u>

- When a transition t_i fires, tokens are removed from input places and placed in output places, i.e.

  for all $p_j \varepsilon I(t_i)$, $M(p_j) = M(p_j) - 1$,

  for all $p_j \varepsilon O(t_i)$, $M(p_j) = M(p_j) + 1$,

Figure 3.1, shows an example of a PN before and after firing a transition. The state of a PN is given by the marking of the places, M, which changes by firing enabled transitions. One way of analyzing PNs consists of determining different reachable states and, then, extracting information out of the state space, called the reachability graph. Reachability analysis is explained in subsection 4.3.

3.2 THE RENDEZVOUS MODEL OF SYNCHRONIZATION AND ADA

The rendezvous is a message-passing mechanism for process synchronization and communication. Two processes are engaged in a rendezvous when one process makes a rendezvous request and the other accepts the rendezvous. If one of the two processes arrives at its rendezvous activity first, it is suspended until the other process performs the matching activity. After rendezvous-ing, the two processes may proceed concurrently. The rendezvous model is the basis of interprocess communication in CSP [Hoare 78] and its variants.

Ada [DoD 81] also adopts the rendezvous model and it is used in this work as a representative concurrent programming language, as is the case in most of the literature on concurrent program testing. In Ada, tasks are equivalent to processes. Tasks enter a rendezvous when one task makes an entry call to another task and the called task accepts the entry. An entry call specifies that the calling task is ready for a rendezvous with another task that has this entry. The called task is ready to accept an entry call when its execution reaches a corresponding accept statement, which also specifies the action to be done. A task reaching an entry call or an accept statement may not proceed until a rendezvous has been made. After the completion of the rendezvous, both tasks may continue their execution concurrently. The Ada constructs for rendezvous request and accept are illustrated in a simple example in Figure 3.2.

Moreover, the Ada language includes a nondeterministic select statement. this statement provides a mechanism for a called task to select among alternative entry calls. An example is given in Figure 3.3. It should also be noted that in Ada, concurrent tasks are allowed to access shared global variables in addition to communication by rendezvous.

```
1     Task body SENDER is
2             story   :  integer;
      . . .
3     begin
4             create (story);
5             RECEIVER. takemessage (story);
6             z := story + w,
7     end SENDER

8     Task body RECEIVER is
9             y   :  integer;
      . . .
10    begin
11            accept takemessage (message : in integer) do
10            z   :  = message + y;
13            end,
14            z := message - w;
15    end RECEIVER
```

Figure 3.2 An example illustrating Ada constructs for rendezvous.

(Variables z and w are assumed to be global)

```
select
    accept storemessage (message : in messageformat) do
        consume (message);
    end;
    . . .
or
    accept retrievemessage (message : in messageformat) do
        consume (message);
    end;
    . . .
end select;
```

Figure 3.3 An example illustrating the select statement.

4. EXTENDED PETRI NET-BASED TESTING APPROACH

As discussed above, the Petri net model which has been used to represent concurrent programs is of the Place-Transition (PT) type [Shatz 88a, Murata 89a, Goel 90]. It is based entirely on program syntax. Hence, its analysis may produce spurious error reports due to the inability to prune infeasible paths. Furthermore, the previous Petri net-based approaches have not incorporated analysis capabilities for detecting race conditions on global variables and have not dealt with conditional loops that contain synchronization statements.

In this section, an extension is presented to the previous Petri net framework for testing the tasking behavior of Ada concurrent programs. The extension is based upon a high-level Petri net model called Predicate-Action net and is introduced to overcome shortcomings of previous approaches by providing enhanced capabilities in a coherent and unified fashion. The model consists of a place-transition net with a predicate-action extension attached to transitions. Predicates correspond to decision statements. Actions correspond to updating of those variables, which affect synchronization, and accessing of shared global variables between two transitions. The predicate-action extension represents addition of information of program semantics to the model. It allows the detection and pruning of infeasible paths and helps in detecting synchronization errors caused by incorrect predicates in decision statements. The action of accessing shared data is represented by augmenting transitions with read and write sets of global variables for detecting anomalies of race conditions.

The analysis is preformed on a reachability graph derived from the augmented predicate-action net (APrAN). The nodes of the reachability graph are augmented with sets of global variables. The paths in the augmented reachability graph (ARG) are generated or pruned depending upon the boolean values of the predicates attached to transitions. Detection of deadlock errors and potential race conditions is done by searching the ARG state nodes.

The APrAN model of concurrent software is used here for Ada's rendezvous model of interprocess communication. However, the modeling and analysis approach is not language-dependent.

The APrAN-based approach for testing concurrent software is explained in the following subsections. APrAN and the modeling procedure are presented first, then the analysis is illustrated.

4.1 PREDICATE-ACTION NET MODEL OF TASKING PROGRAMS

A Predicate-Action net (PrAN) model, introduced by Keller [Keller 76] for the formal verification of parallel programs, consists of a PT net [Peterson 81, Murata 89b] with predicates and actions incorporated in the enabling conditions and firing rules of transitions.

Definition: A Predicate-Action Net is an 8-tuple, PrAN = (P, T, I, O, M_O, V, PR, ACT) where

V	= $\{v_1, ..., v_k\}$ is a set of program variables and constants,
PR	= $\{pr_1, ..., pr_n\}$ is a set of predicates,
pr:	EXP --> {TRUE, FALSE} is a (partial) function,
EXP	= set of expressions, where an expression is defined over V. The grammar defining the expressions has the usual arithmetic and relational operators as terminal symbols.
ACT	= $\{act_1, ..., act_n\}$ is a set of actions,
act	= V --> EXP is a (partial) function
P T	= 0 and PUT = 0,

and the other symbols are as explained in subsection 3.1.

It should be noted that V is the subset of the program variables that affect the synchronization behavior, as will be discussed later in this subsection, and that is assumed that the weight on every arc is 1 and that the maximum capacity of a place is 1. A graphical representation is depicted in Figure 4.1. The enabling conditions and the firing rules are modified as follows.

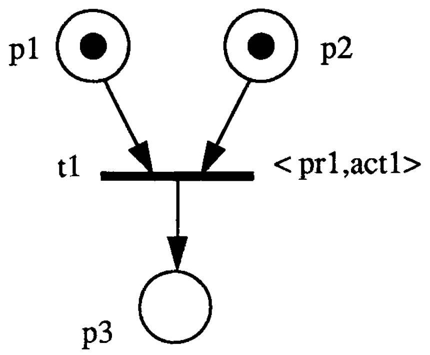


Figure 4.1 Predicate-Action Net.

Enabling Conditions

- A transition t_i is enabled if each of its input places contain a token and the associated predicate is true, i.e.
  for all $p_j \varepsilon I(t_i)$, $M(p_j) = 1$,
  for all $p_j \varepsilon O(t_i)$, $M(p_j) = 0$
  and $pr_i(V) = \text{TRUE}$

Transition Firing Rules

- When a transition t_i fires, tokens are removed from input places and placed in output places and the associated action is invoked to update the relevant program variables, i.e.
  for all $p_j \varepsilon I(t_i)$, $M(p_j) = M(p_j) - 1$,
  for all $p_j \varepsilon O(t_i)$, $M(p_j) = M(p_j) + 1$,
  and $act_i(V)$ is invoked.

Figure 4.2 shows an example of a PrAN before and after firing a transition, assuming the predicate evaluates to TRUE. The state of a PrAN is given by the marking of the places, M, and by the state of the subset, V, of the program variables.

Translation of Program into PrAN

An Ada tasking program can be transformed to a PrAN model by translating its statements into PrAN subnets and then connecting them together. The statements of interest are the tasking statements and the control statements that affect the tasking bahavior by including rendezvous statements within their direct scope of control. Both types of statements determine the structure of a corresponding PrAN and directly determine the movement of tokens. They are henceforth referred to as tasking-related (TR) statements. Other statements

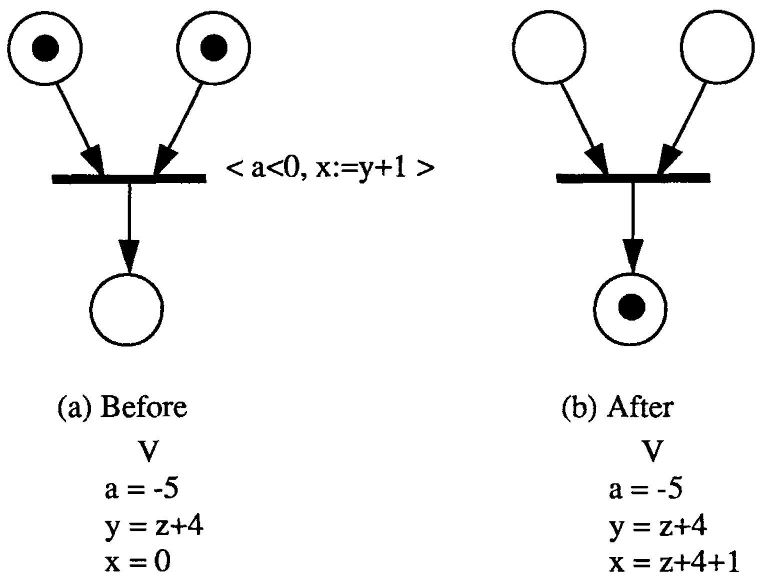
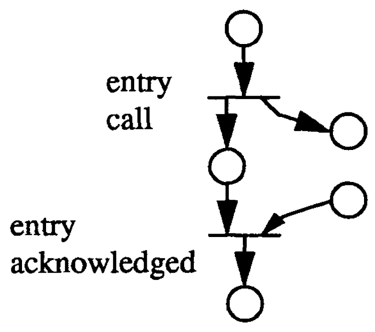


Figure4.2 A PrAN before and after firing a transition.

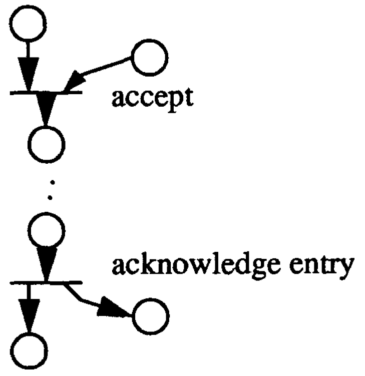
of interest are assignment statements that affect the tasking behavior by updating program variables and control statement that do not include TR statements in their scope of control but include relevant assignment statements. These are also considered, although in a different way, and will be referred to as indirectly tasking-related (ITR) statements. Specifically, the TR statements to be translated into PrAN subnets are rendezvous statements (entry call, or entry call accept), nondeterministic select statements and control statements (if, loops) with rendezvous statements within their body. The conditions in the if-statements appear as 'predicates' associated with transitions. The ITR assignment statements that follow a TR statement, until the next TR statement, appear as 'action' associated with the transition corresponding to the first TR statement. The ITR control statements are also translated into PrAN subnets like the TR control statements. The PrAN subnet models are defined in a semi-formal way in Figure 4.3. S, S1, S2 and S3 in Figure 4.3 are assumed to be a collection of ITR assignment statements, included for illustration purposes. The terminal components of all subnets, as shown in Figure 4.3, must be places. All places within a task are called sequential places. Places extending to other tasks, in rendezvous statements, are called synchronization places. Compatible terminal places in subnets are merged to form a PrAN model for the tasking behavior of a concurrent program. In the total model, subnets may be nested or combined in any way that reflects the structure of the program.

The PrAN model of a tasking program is finite since it is constructed by components (subnets) equivalent to TR statements in the program. Thus, the size of the model is linearly proportional to the number of TR statements. The PrAN model of each task is connected because the consecutive subnets can always be connected by merging terminal sequential places. The PrAN model is safe since the arcs weight is one, the place capacity is one token and none of the subnet structures allows an accumulation of tokens that exceeds the capacity of the places.

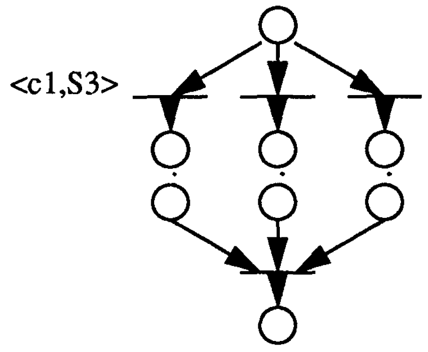
The correspondence between PrAN models and concurrent programs is not one-to-one. In spite of this, we argue that the PrAN model is suitable to represent the structure of a



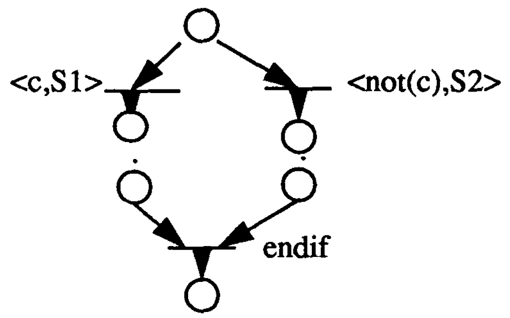
(a) Rendezvous request



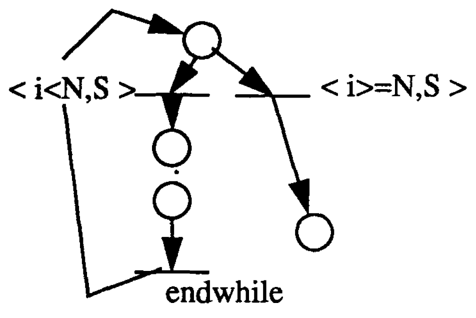
(b) Rendezvous accept



```
(c) select
          when c1 accept1 ; S3 ; ...
       or accept2 ....
       or accept3 ....
    end select
```



(d) if c then S1;..else S2;.. endif



(e) while i<N do S; ..... endwhile
(S is expected to include i:=i+k)

Fig 4.3 PrAN subnets for programming constructs.

concurrent program. The argument about the correctness of the PrAN model within this framework is supported by the validation results of the implementation of previous Petri net-based approaches [Shatz 89, Goel 90]. A similar modeling technique has previously been used to demonstrate the equivalence of a Petri net and a Turing machine in terms of computational power [Petersen 81].

4.2 AUGMENTING THE PrAN MODEL WITH USAGE OF GLOBAL VARIABLES

The PrAN model is augmented with the usage of global variables so that its analysis will also reveal the anomalies of conflicting access of shared variables by more than one task concurrently. The resulting model is henceforth referred to as augmented PrAN (APrAN).

A transition in the net is augmented with a Read set and a Write set of global variables in the transition's concurrency zone, which is defined as follows.
Definition: A concurrency zone of a transition is a sequence of program statements that includes and follows the statement corresponding to the transition. The last statement in the zone is that preceding the statement corresponding to the next transition in the net.

A Read set (RS) contains the global variables that occur on the right hand side of assignment statements in the concurrency zone. The Write set (WS) consists of the global variables that are updated.

Each task is divided into concurrency zones. Zones in one task succeed each other. Concurrency zones in different tasks may be concurrent or not depending upon their position with respect to the rendezvous (synchronization) points in the tasks. Zones in different tasks are said to be concurrent if the statements lying in these zones can be executed concurrently. for example, if two tasks T1 and T2 synchronize at point S1 (referring to the two corresponding program statements), a zone in T1 before S1 cannot be concurrent with a zone in T2 after S1. For illustration, a program may be represented by a graph. The nodes of the graph represent zones, vertical edges refer to the sequencing relationship between two

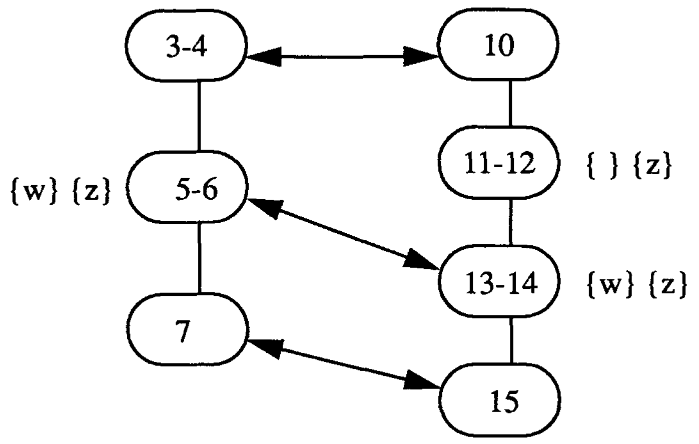


Figure4.4 Graph of concurrency zones for the program in Fig. 3.2.
(RS and WS sets are shown next to the relevant nodes)

contiguous successive zones in one task and horizontal edges refer to potential concurrency between two zones in different tasks. An example of such a graph is shown in Figure 4.4, which shows the concurrency zones of the program given in Figure 3.2. Note, for example, that since task SENDER is suspended at statement 5 until task RECEIVER executes statement 13 (acknowledging end of rendezvous), zones 5-6 and 11-12 are not concurrent and hence no horizontal edge is shown in the graph between them. The sets of variables shown in Figure 4.4 next to the graph nodes are RS and WS sets in the respective concurrency zones. RS and WS sets are shown in Figure 4.5 augmenting the PN's transitions that correspond to the zones.

The access of shared variables is considered part of the 'action' associated with a transition in APrAN. The following additions to the PrAN model are required.
Definition: An Augmented PrAN is a 10-tuple

$$APrAN = (P, T, I, O, M_O, V, SV, PR, ACT, SACT)$$

where SV V is a set of shared variables,
SACT = $\{sact_1, ..., sact_n\}$ is a set of actions on shared variables and $sact_i$ is a (partial) function on SV that places a shared variable either in RS or WS of t_i.

Graphically, APrAN appears in Figure 4.6.

The following is added to the firing rules:

When a transition t_i fires, $sact_i(SV)$ is invoked. That is, the shared variables in t_i's concurrency zone are accessed (read or write) and hence the sets RS and WS are formed.

The definition of a state of an APrAN at an instant also includes the sets RS and WS of all tasks at that instant.

With these additions to the firing rules and the definition of APrAN state, the formation of RS and WS sets is incorporated in a coherent way in the program modeling procedure.

4.3 ANALYSIS OF APrAN MODEL

The APrAN model of a tasking program is executed to generate a reachability graph

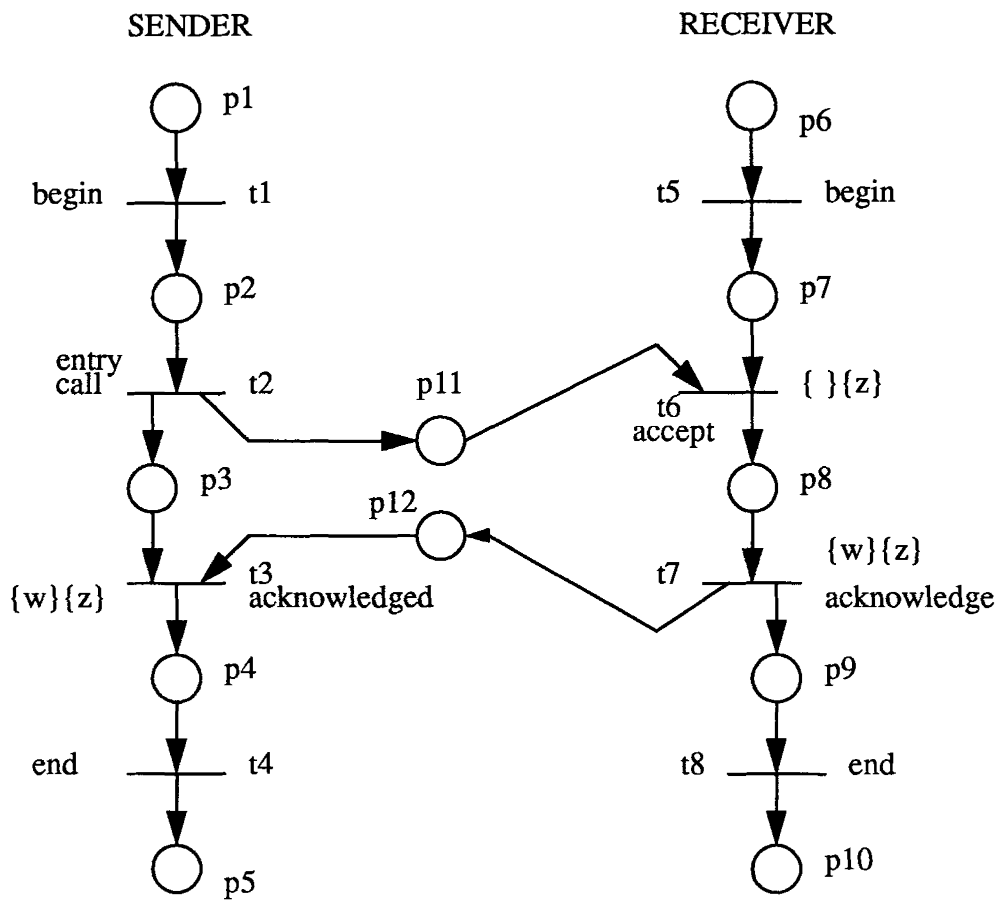


Figure 4.5 Augmented Petri net model for the program in Figure 3.2.
(Augmenting sets are in the order : RS , WS)

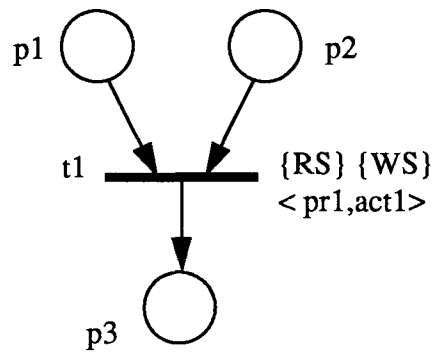


Figure 4.6 Augmented Predicate-Action Net

augmented with sets of shared variables, referred to as augmented reachability graph (ARG). This graph is then analyzed to examine the tasking behavior of the underlying program. Most importantly, deadlocks and potential race conditions on shared variables are considered. The concepts involved in the generation and analysis of the ARG are briefly presented in this subsection. First, some definitions are given informally. They are simple extensions to the definitions related to PT nets [Peterson 81], adapted here for APrAN. ARG generation and analysis is then illustrated.

Definition: An APrAN state (m, VS, SVS) is defined by a marking M of the net, a state VS of the subset of program variables V and a collection of pairs of RS and WS sets, denoted as SVS, with one pair for every concurrent task.

Definition: A firing sequence FS (subset of T) is an ordered sequence of transitions t_i, t_d, ..., t_k such that after firing t_b ε FS, a new state of APrAN is reached at which the enabling conditions for the immediate successive transition in FS are satisfied.

It should be emphasized that firing a transition, with associated predicate-action, alters not only the marking of the net, but also the state of V and the sets RS and WS. The conjunction of predicates associated with transitions in a firing sequence is what is known in symbolic execution literature as path condition.

Definition: A reachability set RS(M, VS, SVS, FS) is the set of all states reachable from state (M, VS, SVS) connected by transitions t_i ε FS such that if (M_1, VS_1, SVS_1) ε RS then (M_2, VS_2, SVS_2) ε RS for for some transitions in FS.

Definition: An augmented reachability graph (or tree) ARG is the set of all reachability sets RS(M_0, VS_0, SVS_0, FS) for all possible firing sequences FS. M_0 is the initial marking of the net. VS_0 is given by initial values of variables. SVS_0 is given by empty RS and WS sets. Graphically, a state (or node) in ARG is represented by a marking augmented with RS and

WS sets for all tasks. An arc between two nodes is labeled by the corresponding fired transition.

It should be noted that a path in ARG corresponds to a sequence of synchronization events, i.e. rendezvous, in the program. The procedure for generating an ARG for APrAN is similar to that for PT nets. However, it takes into account the sets of shared variables and makes use of symbolic evaluation to minimize infeasible paths and to account for conditional loops. The ARG generation procedure starts at an initial state (M_O, VS_O, SVS_O), as defined above, and repeats a basic step until no more nodes, i.e. states, can be generated. The basic step in the generation procedure is the determination of all enabled transitions at a given state. The enabling conditions of a transition include both the availability of tokens in the input places and the (symbolic) evaluation of the associated predicate to 'TRUE' in a given state of the variables in V. The enabled transitions will then be fired in all possible permutations. Each time a transition, which belongs to a task subnet, is fired a new token marking is reached, an update of variables in V may take place and a new concurrency zone in the relevant task may be entered. A new concurrency zone for a task yields new RS and WS, possibly empyt, augmenting the generated node. A transition whose predicate evaluates to 'FALSE' cannot be fired and hence the corresponding path in the ARG is pruned. Such paths are infeasible and would have been allowed in the reachability graph of a PT net.

The generation procedure terminates and yields a finite ARG because ARG corresponds to a finite APrAN and the reachability graphs of the component subnets of APrAN are finite. APrAN is finite since it models finite tasking operations in the program. Of particular interest are subnets of conditional loops, as translated by APrAN. Subnets representing the body of a loop can be executed only once. This is sufficient to detect deadlocks resulting from misordering of rendezvous statements. In addition, a symbolic comparison is performed on the loop counters to ensure a match between the number of entry calls and rendezvous accepts. Therefore, a finite number of nodes in ARG is produced by loops. Handling of loops is explained in Section 5.

A terminal node in ARG corresponds to either a valid termination state or to a deadlock state. Valid termination indicates that all tasks have performed their synchronization and communication operations and are no longer active. Its determination in terms of net markings is an implementation issue. A deadlock state is a terminal state that does not represent valid termination.

The analysis of ARG is carried out by searching all nodes for deadlock errors and anamolies of shared data usage. A deadlock error is reported when a deadlock state is found. Potential race conditions on shared data are reported when more than one task may conflict over the access of shared variables in one state. A race condition on a variable x occurs when pairwise comparisons of RS_i and WS_i sets, augmenting a node, for all tasks i detect the membership of x in RS_i and WS_j or WS_i and WS_j of at least two different tasks (i.e. $i \neq j$).

To illustrate the basic concepts in this approach, an example is given by the program in Figure 4.7 and its corresponding PrAN and RG in Figures 4.8 and 4.9. Shared variables are not included in the program because it is not possible to show them in the figures.

```
task A is
end A;

task B is
  entry E;
end B;

task body A is
begin
  if a < b then
    B.E;
  else
    c := b + 1;
  endif;
  if a>=b then
    B.E;
  else
    b := a;
  endif;
end A;

task body B is
begin
  accept E;
end B;
```

Figure4.7 An Ada program that does not contain a deadlock.

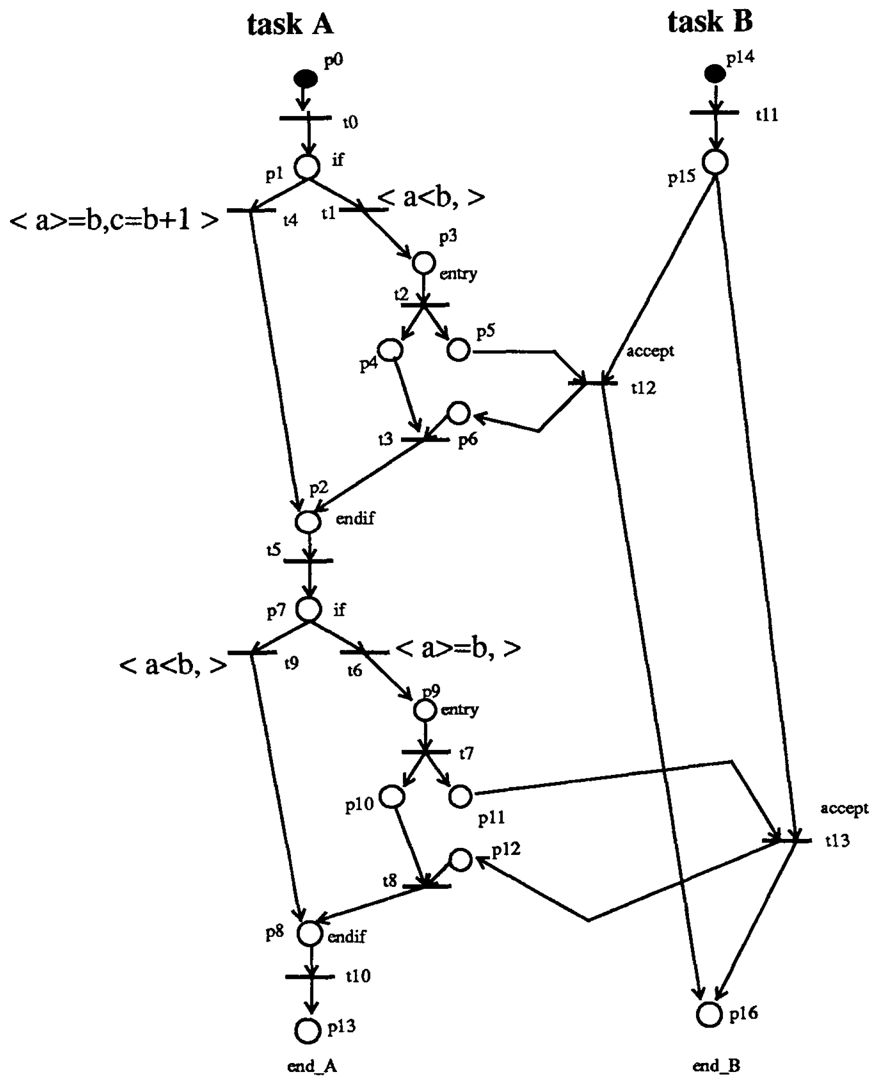


Figure4.8 APrAN model of the program in figure 6.

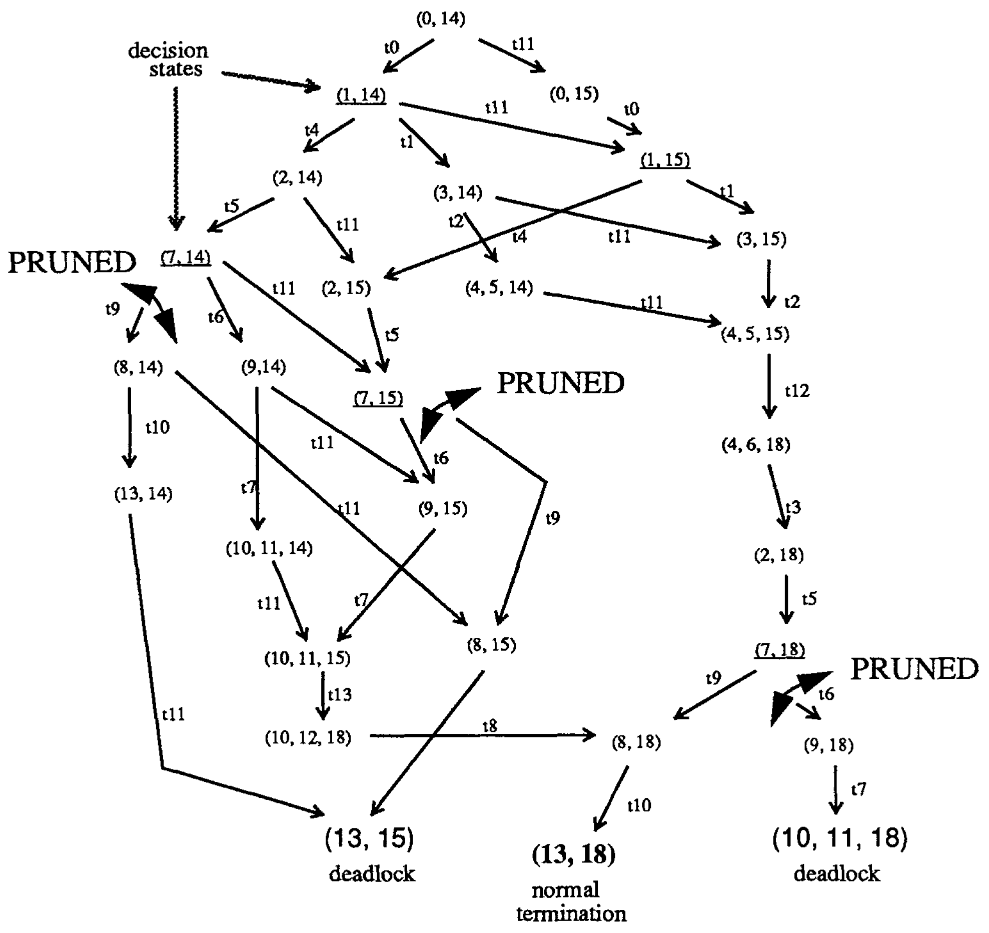


Figure4.9 Reachability graph for PrAN in Fig. 7.

5. IMPLEMENTATION ISSUES

A block diagram of a structured tool system implementing the APrAN-based testing approach is shown in Figure 5.1. The system may consist of four modules. The Modeling (MOD) module produces an Augmented Petri Net (APN) model of the tasking-related program statements. The Augmented Reachability Graph Generator (ARGG) module constructs the augmented reachability graph (ARG) of the APN. The Reachability Graph Analyzer (RGA) module may be composed of various procedures that analyze the information offered by the ARG about the underlying concurrent program. The User Interface (UI) module may use X-Windows software to facilitate interaction with users. The UI module may offer a menu-driven user friendly environment, where a user can select one of several analysis options by clicking a mouse and can view multiple results simultaneously. Implementation issues for the four modules and for dealing with conditional loops are illustrated in the following subsections.

5.1 MODELING MODULE

The MOD module translates an Ada source code into an APN model. It also yields useful byproducts which are a source program with line numbers, referred to as numbered statement list (NSL), and a list of tasking-related statements, referred to as intermediate program (IP). Other useful data structures are a table of subnets corresponding to Ada language constructs, a table of task names and identification numbers (ID), a table of rendezvous information involving all synchronization points, a table of concurrency zones involving shared variables in different sections of the tasks, a predicate dependence tree (from which path conditions can be extracted) and trees of symbolic expressions for variables in TR and ITR statements.

Translation of source code into APN considers directly only TR statements, that is IP. ITR statements are utilized, when necessary, through the symbolic expression trees. The translation strategy consists of using Ada subnets as templates or building blocks and

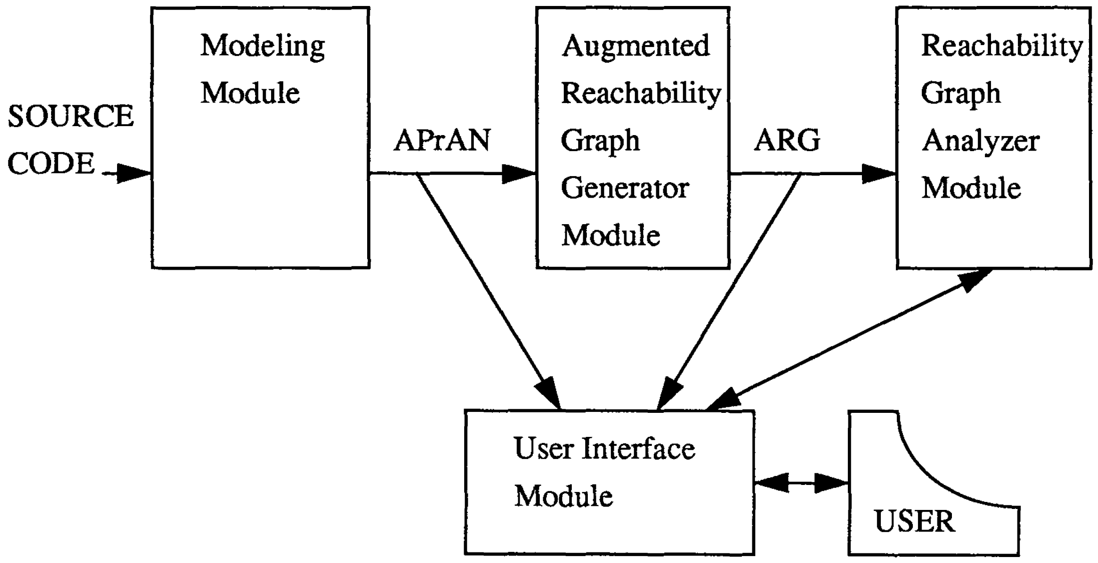

Modeling Module
SOURCE CODE
APrAN
Augmented Reachability Graph Generator Module
ARG
Reachability Graph Analyzer Module
User Interface Module
USER

Figure 5.1 A block diagram of a tool system for implementing APrAN-based testing approach.

connecting these subnets based on either the sequential location of the corresponding statement or information derived from rendezvous tables. Translation can be done by scanning IP statements in sequence, fetching corresponding templates in a table look-up fashion, labeling the places and transitions of the subnets with identification information for later analysis, augmenting the subnets with pointers to predicates and actions, connecting the subnets by combining compatible sequential and synchronization places, and building necessary tables and data structures.

The MOD module may consist of three phases. In phase 1, the source code is scanned and filtered to produce an IP. Also NSL may be produced for later reference in error reporting. In phase 2, IP is scanned to construct tables and data structures needed for the next phase. In phase 3, another pass through IP is made to build the APrAN model of the underlying program. An outline of the three phases is given below.

Phase 1:

- Read the source code and assign line numbers to statements for producing NSL.
- Identify statements that are tasking-related and construct IP.
- Identify global variables, by differentiating them from locally declared variables, in each task with the numbers of the statements to which they belong and determine whether they occur as Read or Write variables.
- Identify variables in the predicates of the control statements affecting tasking, and the numbers of these statements.
- Build a data dependency graph (DDG).
- Build a predicate dependency tree (PDT), where a node consists of the statement number and the variables involved.

Phase 2:

- Scan IP.

- Create a Task Table, which is a list of all tasks in the program with an assigned unique integer ID.
- Construct a Rendezvous Table, which consists of IDs of tasks requesting rendezvous, IDs of tasks accepting rendezvous, entry points in the accept statements and the line numbers of these statements.
- Construct a Concurrency Zones Table. A concurrency zone corresponds to statements in the source program that lie between two statements in IP, including the IP statement that occurs first and excluding the second one (which becomes the first statement in the next zone). Each row in the table corresponds to a concurrency zone in a task. A row consists of the task ID, the start statement number of the zone, the finish statement number of the zone, the Read set of global variables in the zone and the Write set of global variables. The number of the start statement of a concurrency zone is used as the index of the table.
- Using DDG and PDT, build trees of symbolic expressions for variables occurring in TR predicates. Each variable can have several indices to expressions, where each index will actually be related to a path condition. These trees of expressions will be used for the symbolic evaluation of predicates, within a path, in the next module.
- Using DDG, build trees of symbolic expressions for varialbes in statements that occur between 2 successive TR predicates. These trees will be short and are used as actions when firing transitions.

<u>Phase 3</u>:

- Purge DDG.
- Scan IP.
- For each statement, look up the corresponding template subnet.
- Link predicates and actions, as specified by trees of expressions built in phase 2, to transitions by pointers/indices.
- Augment transitions with Read and Write sets of shared variables determined from the

corresponding row in the table of concurrency zones.

- Label synchronization places with the name of the task involved and the synchronization status (e.g. entry, accept, end). Also, label transitions with the type and line number of the statement it corresponds to (in NSL). The labels are used in connecting subnets and in error reporting by UI module. This step uses the rendezvous table.
- Store in the data structures of places (resp. transitions) unique IDs, the number of input and output transitions (places) and the number of tokens (initially zero).
- Connect subnets by merging compatible terminal sequential places in consecutive subnets, within the same task, and by merging compatible synchronization places of rendezvous subnets in different tasks. This step uses the rendezvous table and place labels (to detect compatibility).
- Finally, assign single tokens to the begin-places of all tasks to prepare APN for the construction of the reachability graph.

5.2 AUGMENTED REACHABILITY GRAPH GENERATOR MODULE

As explained in Section 4.3, an ARG is formed of nodes and arcs. A state node represents a marking of the net and is augmented with RS and WS of shared variables. An arc represents a fired transition which leads to a new state. The ARG generation strategy is based upon firing all enabled transitions in all possible combinations at any given state of the APrAN. A depth-first generation procedure is presented below. The input to the procedure is an APrAN and its output is an ARG. Nodes of the ARG can be assigned unique node IDs, a level (from the root) number, the IDs of the input and output arcs (i.e. APrAN transitions) and pointers to RS and WS sets for all tasks. Other useful data structures are a list of unexplored ARG nodes, UNEXPLORED, a list of token-enabled transitions, TRENABLED, and a stack of predicates, PREDSTACK. A valid termination node is determined by the presence of tokens in end-places of all tasks.

Procedure

- Root node of ARG corresponds to the state resulting from the presence of tokens in the begin-places of all tasks and from the augmenting RS and WS sets of the first concurrency zone in all tasks. Initially UNEXPLORED contains only the root node.
- Repeat until no more nodes in UNEXPLORED:
  - (a) Find the first node in the list, UNEXPLORED.
  - (b) For the new state, search in the neighborhood of places with tokens for enabled transitions (That is, not all APN needs to be searched). Create TRENABLED. In case of structural conflict (if-then-else) add both transitions to TRENABLED.
  - (c) For each enabled transition, evaluate its predicate with respect to the path condition (i.e. conjunction of predicates) recorded so far. If it evaluates to 'TRUE' fire the transition and push the index of the predicate onto the path condition stack, PREDSTACK, otherwise the path is pruned. This step requires symbolic evaluation (e.g. use simplex method) using the trees of expressions of variables constructed in the modeling module.
  - (d) Add the new child state node, created by firing the transition to UNEXPLORED.
  - (e) Go to (b).
  - (f) When it is no longer possible to pursue a path any further (due to pruning, deadlock, etc.), pop PREDSTACK (i.e. backtrack one step along the path), search TRENABLED for transition token-enabled in this state and go to step (c).
  - (g) Whenever a transition fires, change the number of tokens in the input and output places, update RS and WS corresponding to the fired transition in the specified task (by using table of concurrency zones with transition ID as index) and symbolically evaluate the variables specified by the attached action.
  - (h) Delete nodes from UNEXPLORED if all their transitions in TRENABLED have been fired or if they enable no transitions.

end-repeat

5.3 REACHABILITY GRAPH ANALYZER MODULE

Analysis in the Reachability Graph Analyzer (RGA) module is done on the ARG. Most of the analysis information can be collected during the ARG generation, otherwise analysis is initiated when requests are made by the user through the UI module. Analysis of ARG may provide reports either about location of errors, namely deadlock and concurrent updating of shared data, or for performance information, such as the number of rendezvous per task and the maximum possible number of rendezvous for a task. Performance analysis may provide insights into factors such as workload balancing and bottlenecks in the concurrent program.

The inputs to the RGA module are ARG and user requests through the UI module. Its output is error and analysis reports directed to the user via the UI module.

5.4 USER INTERFACE

The User Interface (UI) module, in conjunction with the RGA module, indicates to the user the location and type of detected errors and anomalies and provides information that may be used for debugging and redesigning the program. The UI module may enable the user to request analysis information, display the results produced by the RGA module in a convenient format and allow the user to inspect important data structures. All these facilities can be provided with a button-click style of operation in an X-Window environment, which hides the complexity of a tool and makes it user-friendly.

User requests can be menu-driven, where the user selects a function by clicking on the relevant entry. Analysis results and associated information can be displayed in multiple windows, so that complimentary information may be viewed simultaneously and different displays may be inspected or manipulated independently. As shown in Figure 5.1, the UI module interacts with the RGA module and has access to important data structures, such as APN, ARG and NSL.

5.5 HANDLING OF CONDITIONAL LOOPS

A simple strategy to partially handle conditional loops is presented in this subsection. This strategy precludes the generation of large numbers of nodes in ARG and is guided by the following:

- Break the branch back arc in the loop template (Figure 4.3(e)) when generating ARG. This accounts for the tasking statements inside the loop once and hence guarantees the detection of misordering in the corresponding tasking statements or of the absence of a matching reciprocal statement.
- Use the symbolic evaluator procedure to symbolically evaluate the number of iterations of loops enclosing the pair of reciprocal tasking statements (accept, entry). compare the number of iterations for the pair. If they do not match, we can be confident that this will generate a deadlock. If the symbolic comparator can not decide, this condition can either be ignored or reported to the user depending on the desired accuracy.

6. CONCLUSIONS

An approach to testing Ada tasking programs has been presented in this paper. It is based on modeling a program by a Predicate-Action net augmented with sets of shared variables (APrAN). An augmented reachability graph (ARG) is then derived from APrAN so that its nodes can be searched for deadlocks and potential race conditions on shared variables.

The predicate-action extension of the net captures some aspects of the dynamic behavior of a concurrent program. Therefore, it leads to a minimization of spurious error reports encountered in pure static analysis. This is achieved by pruning infeasible paths in the ARG when the conjunction of the predicates along a path becomes false. Other advantages of the APrAN-based approach are the detection of errors due to erroneous decision program statements and the partial handling of finite conditional loops in a simple and practical way.

The APrAN-based approach suffers from combinatorial explosion for non-small scale programs. Possible ways to circumvent this problem are discussed in another paper [Goel 90]. Moreover, the extended features of the APrAN model are not penalty-free. The boolean evaluation of predicates and the invocation of actions require expensive symbolic evaluation. However, the full cost of symbolic evaluation is not incurred because it can be used on demand basis and is limited to the evaluation of 'predicates' and 'actions' related to tasking. On the other hand, practical results will not be as good as the theoretical approach suggests due to the inherent imperfection of symbolic evaluation. More research is needed to weigh the advantages of the APrAN-based approach versus the costs incurred by symbolic evaluation.

REFERENCES

G. Avrunin, et al., "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems," IEEE Trans. Software Eng., pp. 278-291 (February 1986).

G. Berthelot, "Checking Properties of Nets Using Transformations," in Lecture Notes in Computer Science, Vol. 222, pp. 19-40 (1986).

G. Bristow, et al., "Anomaly Detection in Concurrent Programs," Proc. 4th Int. Conf. Software Eng., pp. 265-273, (September 1979).

R.H. Carver and K.C. Tai, "Reproducible Testing of Concurrent Programs Based on Shared Variables," Proc. 6th Int. Conf. on Distributed Computing Systems, pp. 428-433 (1986).

R.H. Carver and K.C. Tai, "A Semantics-Based Approach to Analyzing Concurrent Programs," Proc. 2nd Workshop on Software Testing, Verification and Analysis, pp. 132-133 (1988).

L.K. Dillon, R.A. Kemmerer and L.J. Harrison, "An Experience with Two Symbolic Execution-Based Approaches to Formal Verification of Ada Tasking Programs," Proc. 2nd Workwhop on Software Testing, Verification and Analysis, pp. 114-122 (1988).

L.K. Dillon, "Symbolic Execution-Based Verification of Ada Tasking Programs," Proc. 3rd Int. IEEE conf. on Ada Applications and Environments, pp. 3-14 (May 1988).

Department of Defense. The Programming Language Ada: Reference Manual: Proposed Standard Document, U.S. DoD. Berlin, Springer-Verlag (1981).

J. Gait, "A Probe Effect in Concurrent Programs," Software Practice and Experience, pp. 225-233 (March 1986).

C. Ghezzi, et al., "Symbolic Execution of Concurrent Systems Using Petri Nets," Technical Report, Politecnico di Milano, Dipartimento de Elettronica (1988).

A.L. Goel and N. Mansour, "A Petri Net-Based Tool for Detecting Deadlocks and Race Conditions in Concurrent Programs," Submitted for publication (1990).

C.A.R. Hoare, "Communicating Sequential Processes," Comm. of the ACM, 21 (8), pp. 666-677 (August 1978).

H-L. Hausen, "Comments on Practical Constraints of Software Validation Techniques," in Software Validation, edited by H-L. Hausen, North Holland (1984).

L.J. Harrison and R.A. Kemmerer, "An Interleaving Symbolic Execution Approach for the Formal Verification of Ada Programs with Tasking," Proc. 3rd Int. IEEE Conf. on Ada Applications and Environments, pp. 15-26 (May 1988).

D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," IEEE Software, Vol. 2, No. 2, pp. 47-57 (March 1985).

W.E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Trans. Software Eng., Vol. 2, No. 3, pp. 208-214 (September 1976).

R.M. Keller, "Formal Verification of Parallel Programs," Comm. of the ACM, Vol. 19, No. 7, pp. 371-384 (July 1976).

D.L. Long and L.A. Clarke, "Task Interaction Graphs for Concurrency Analysis," in Proc. Int. Conf. Software Engineering, pp. 44-52 (May 1989).

K-H. Lee and J. Favrel, "Hierarchical Reduction Method for Analysis and Decomposition of Petri Nets," IEEE Trans. Systems, Man and Cybernetics, Vol. 15, No. 2, pp. 272-281 (March/April 1985).

K-H. Lee, J. Favrel and P. Baptiste, "Generalized Petri Net Reduction Method," IEEE Trans. Systems, Man and Cybernetics, Vol. 17, No. 2, pp. 297-303 (March/April 1987).

C.E. McDowell, "Viewing Anomalous States in Parallel Programs," Proc. Int. Conf. Parallel Processing, Vol II, pp. 54-57 (1988).

C.E. McDowell, "A Practical Algorithm for Static Analysis of Parallel Programs," J. of Parallel and Distributed Computing 6, pp. 515-536 (1989).

T. Murata, B. Shenker and S. Shatz, "Detection of Ada Static Deadlocks Using Petri Net Invariants," IEEE Trans. Software Engineering, Vol. 15, No. 3, pp. 314-325 (March 1989).

T. Murata, "Petri Nets," Proc. IEEE, VOl. 77, No. 4, pp. 541-580 (April 1989).

S. Morasca and M. Pezze, "Validation of Concurrent Ada Programs Using Symbolic Execution," Technical Reprot, Politecnico di Milano, Dipartimento de Elettronica (1989).

E.T. Morgan and R. Razouk, "Interactive State-Space Analysis of Concurrent Systems," IEEE Trans. Software Eng., Vol. 13, No. 10, pp. 1080-1091 (October 1987).

L. Osterweil, "Integrating the Testing, Analysis and Debugging of Programs," in Software Validation, edited by H-L. Hausen, North Holland (1984).

J.L. Peterson. Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs, N.J. (1981).

S.M. Shatz and W.K. Cheng, "A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior," J. Systems and Software 8, pp. 343-359 (1988).

S.M. Shatz, "Towards Complexity Metrics for Ada Tasking," IEEE Trans. Software Eng., Vol. 14, No. 8, pp. 1122-1127 (August 1988).

S.M. Shatz, K. Mai, D. Moorthi and J. Woodward, "A Toolkit for Automated Support of Ada Tasking Analysis," in Proc. Int. Conf. Distributed Computing Systems, pp. 395-402 (1989).

K.C. Tai, "On Testing Concurrent Programs," Proc. COMPSAC, pp. 310-317, (October 1985).

K.C. Tai and E.E. Obaid, "Reproducible Testing of Ada Tasking Programs," Proc. IEEE 2nd Int. Conf. on Ada Applications and Environments, pp. 69-79 (April 1986).

K.C. Tai and S. Ahuja, "Reproducible Testing of Communication Software," Proc. COMPSAC, pp. 331-337 (1987).

K.C. Tai and R.H. Carver, "Testing and Debugging of Concurrent Software by Deterministic Execution," In Proc. 7th Pacific Northwest Software Quality Conf. (1989).

K.C. Tai, "Testing of Concurrent Software," Proc. COMPSAC, pp. 62-64, (September 1989).

R.N. Taylor and L.J. Osterweill, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Trans. Software Eng., Vol. 6, No. 3, pp. 265-277 (May 1980).

R.N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," Comm. of the ACM, 26(5), pp. 362-376 (May 1983).

R.N. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs," Acta Informatica 19, pp. 57-84, (1983).

R.N. Taylor and C.D. Kelly, "Structural Testing of Concurrent Programs," in Proc. Workshop on Software Testing, IEEE Comp. Soc. Press, pp. 164-169 (July 1986).

S.N. Weiss, "A Formal Framework for the Study of Concurrent Program Testing," Proc. 2nd Workshop on Software Testing, Verification and Anaysis, pp. 106-113 (1988).

J. Wileden and G. Avrunin, "Toward Automating Analysis Support for Developers of Distributed Software," Proc. 8th Int. Conf. on Distributed Computing Systems, pp. 350-357 (June 1988).

R.G. Willson and B.H. Krogh, "Petri Net Tools for the Specification and Analysis of Discrete Controllers," IEEE Trans. Software Eng., Vol. 16, No. 1, pp. 39-46 (January 1990).

M. Young and R.N. Taylor, "Combining Static Concurrency Analysis with Symbolic Execution," IEEE Tr. Software Engineering, Vol. 14, No. 10, pp. 1499-1511 (October 1988).