9-1990

A Petri Net-Based Tool for Detecting Deadlocks and Race Conditions in Concurrent Programs

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

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

SU-CIS-90-31

A Petri Net-Based Tool for Detecting Deadlocks and Race Conditions in Concurrent Programs

Amrit L. Goel and Nashat Mansour

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-90-31

A Petri Net-Based Tool for Detecting Deadlocks and Race Conditions in Concurrent Programs

Amrit L. Goel and Nashat Mansour

September 1990

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

(315) 443-2368

A PETRI NET-BASED TOOL FOR DETECTING DEADLOCKS AND RACE CONDITIONS IN CONCURRENT PROGRAMS

by

Amrit L. Goel[*]
Nashat Mansour[**]

September 1990

[*] Professor, Department of Electrical and Computer Engineering and the School of Computer and Information Science, Syracuse University, Sryacuse, NY 13244.

[**] Researsch Assistant, School of Computer and Information Science, Syracuse University, Syracuse, NY 13244.

ABSTRACT

A static analysis tool for detecting deadlocks and potential race conditions on shared variables in concurrent programs is presented. It is based on Petri Net modeling and reachability analysis, where a concurrent program is modeled as an augmented Petri net and a reachability graph is then derived and analyzed for desired information. Place-Transition subnets representing programming language constructs are described. Transitions in these subnets are augmented with sets of shared variables that occur in sections of the program, called concurrency zones, related to the transitions. The tool consists of four modules. The modeling module employs the augmented subnets as building blocks in translating only the synchronization-related statements of a concurrent program and connects the subnets to yield the total model. The second module produces an augmented reachability graph for the augmented Petri net. The analyzer module searches the augmented reachability graph for deadlocks, race conditions and other useful analysis information requested by the user about the underlying program. The user interface is provided by an X-window based module. Ada is used as a representative of concurrent languages that adopt the rendezvous model of interprocess communication and synchronization. The validation of the tool, its applicability and limitations are also discussed.

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

1. INTRODUCTION AND BACKGROUND

Software testing is a 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]. Testing concurrent software is more difficult than sequential software mainly because in a concurrent program a number of processes are considered. 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 software. The main factors are different processor speed, unpredictable scheduling of multiple processes and nondeterministic language constructs, in addition to different processor speeds. These factors can lead to nondeterministic sequences of execution and cause the reproducibility or replay problem [Tai 85, 89a, 89b], where different executions of the program may yield different results. Moreover, concurrent processes may enter a race condition, if shared variables are allowed in the programming language.

In addition to the usual computational and domain errors, concurrent programs may include synchronization and concurrency errors and anomalies. The most important types 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 concurrent programs have been discussed in [Bristow 79].

The approaches for testing cocnurrent programs can be divided into static analysis and dynamic analysis. In static analysis, the program code is often transformed into a model and the model is then analyzed for detecting specific error states. Static analysis has the advantage that it is independent of the characteristics of the target machine and can be performed in inexpensive and conveneint environments. However, it suffers from a lack of program semantics that may lead to spurious error reports. In dynamic analysis, the program is executed 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].

Several approaches have been proposed for testing concurrent software. Most of the dynamic testing work has been based on deterministic execution testing (DET) [Tai 89a, 89b, 86]. The DET approach is geared towards solving the reproducibility problem. An input test case in DET consists of data and a synchronization sequence, S. A control task is added to the program to force its execution according to S. Hence, results can be reporduced and their validity can be checked. In [Taylor 86], structural testing is proposed based on a concurrency state graph, which is derived by static analysis of the program. The use of a controllable scheduler to force the execution of a path is suggested.

The first static analysis approach appeared in [Taylor 83a]. This approach is based on flowgraph models of concurrent processes or tasks. A directed graph of concurrency states is then derived from the flowgraphs where a state represents the control state of the concurrent tasks, including synchronization information. 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. A similar analysis approach to that of Taylor's appears in [Shatz 88a, 89] but within the 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 for analyzing the reachability graph. The analysis results include information about deadlock states and the synchronization behavior of the program. Within the Petri net framework, [Murata 89a] presents algorithms based on place and transition invariants to guide a selective generation of the reachability graph. A task interaction graph (TIG) is proposed in [Long 89] as a model for tasks or processes. A TIG represents a task as a set of regions and a set of interactions between regions. 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 synchronization event. Deadlock is detected if a task is unable to complete a synchronization activity. [McDowell 89] 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 concurrency in the program is a result of parallel execution of multiple copies of the same task. In this approach, deadlocks and the anomaly of parallel update of shared variables can be detected. In [Dillon 88], symbolic execution is used in the formal verification of Ada tasking programs.

The above approaches to concurrent software testing exhibit the complexity of the testing problem. Based on some of these approaches, tools have been reported for dynamic testing [Tai 89a] and static analysis [Shatz 89, McDowell 89]. This small number of tools and the limited experiences reported do not provide sufficient confidence in the feasibility of automatable testing methodologies. Further, each tool has limited applicability as pointed out in these studies.

The work presented in this paper is a contribution to the testing research, which aims at demonstrating the feasibility, although limited, of automatable approaches for testing concurrent programs. The approach is based upon static analysis using a Petri net model. It is concerned with the concurrency and synchronization behavior of concurrent programs, namely with the detection of deadlocks and potential race conditions. Like other static analysis approaches, this work assumes that the sequential behavior of individual processes is tested by other relevant means independent of testing the concurrency features. The model of synchronization considered here is the rendezvous type. Shared global variables are also allowed. Ada is chosen as a representative of the class of programming language notations that adopt this model of synchronization and concurrency. However, other languages in this class can be easily incorporated in the implementation.

Our approach is based upon Petri net modeling and reachability analysis, which has been previously adopted for deadlock detection in [Shatz 89, Murata 89a]. In this work, both modeling and analysis capabilities of Petri nets are offered in a unified and coherent framework within which a tool has been developed. Furthermore, detection of race conditions on shared variables has been incorporated in a coherent way, without post-processing, by augmenting the Petri net model with shared variables. The tool consists of four modules: A modeling module that translates a concurrent program into a Petri net model augmented with sets of shared variables, a module that generates the reachability graph of the Petri net, augmented with shared variables, a module that performs analysis on the reachability graph, and a user interface module that presents the analysis results and allows user choices in a user-friendly fashion using X-window display facilities. The complexity of the tool, its validation and applicability are also discussed in this paper.

This paper is organized as follows. In the next section, preliminary concepts are presented. The augmented Petri net-based approach is described in Section 3. The tool design is presented in Section 4, and its validation and applicability are discussed in Section 5. A discussion of the work presented here as well as some suggestions for extending this work are presented in Section 6.

2. PRELIMINARIES

In this section, we describe some relevant concepts utilized throughout this paper. These include Petri nets, the rendezvous model of synchronization and selected Ada programming constructs.

2.1 PETRI NETS

A system can be modeled by a Petri net (PN), which becomes its mathematical representation [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, which is defined below. Description of their analysis is integrated with other material in subsection 3.3.

Definition: A PT net is a 5-tuple, PN = (P, T, I, O, M_O), where P is a finite set of places, T is a finite set of transitions, I is a set of transition input arcs, O is a set of transition output arcs and M_O is the initial marking.

For purposes of this paper, it is assumed that the weight on every arc is 1 and that the maximum capacity of a place is 1. Graphically, a PN is a directed bipartite graph with bars representing transitions and circles representing places (see Figure 2.1).

Enabling Conditions: A transition t_i is enabled if each of its input places contain a token.

Transition Firing Rules: When a transition t_i fires, tokens are removed from input places and placed in output places.

Figure 2.1, 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.

2.2 THE RENDEZVOUS MODEL OF SYNCHRONIZATION AND ADA

The rendezvous is a message-passing mechanism for interprocess communication and synchronization. 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.

Ada [DoD 81] 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 and the communication/synchronization among tasks is referred to as the tasking behavior. The Ada constructs for rendezvous request and accept are illustrated in a simple example in Figure 2.2.

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

3. PETRI NET-BASED TESTING APPROACH

The testing approach presented here is of the static analysis type, aimed at revealing deadlock errors and race condition anomalies. It is based on Petri net modeling of concurrent programs that use rendezvous for interprocess communication and synchronization. The concurrent program to be tested is translated into a Petri net model augmented with sets of shared variables. From this augmented Petri net (APN), an augmented reachability graph (ARG) is derived which is used for detecting deadlock errors and potential race conditions. This approach is further explained in this section and details about its implementation are presented in Section 4.

The correspondence between a concurrent program and its PN model is not one-to-one. Yet, such a model is a suitable representation of the static structure of the concurrent program. The argument about the correctness of the PN model within this framework is supported by the validation results of the implementation, presented in Section 5. This modeling technique has previously been used to demonstrate the equivalence of Petri nets and Turing machine in terms of computational power [Petersen 81]. However, it should be noted that the PN model is syntax-based and ignores predicates in decision statements and conditional loops. This leads to shortcomings and limitations, which will be discussed in Section 6.

Although, Ada programming constructs are used in the implementation, the approach is not language-dependent. It is applicable to all design notations that employ the rendezvous model for synchronization and communication, such as CSP [Hoare 78] and its variants.

3.1 PETRI NET MODELING OF CONCURRENT PROGRAMS

A concurrent program can be transformed to a Petri net (PN) model by translating its statements into PN subnets and then connecting them together. The statements of interest are the rendezvous (synchronization or tasking) statements and the control statements that affect the tasking behavior. Both types of statements determine the structure of the corresponding PN model and directly determine the movement of tokens in the PN. They are henceforth referred to as tasking-related (TR) statements. Statements which do not influence the tasking behavior do not contribute to the PN mdoel. Specifically, the TR statements to be translated into PN subnets are rendezvous statements (entry call, accept), nondeterministic select statements and control statements (if, loop) in which rendezvous statements occur within their direct scope of control. These PN subnet models are defined in a semi-formal way in Figure 3.1. The terminal components of all subnets, as seen in Figure 3.1, 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 total PN model for the tasking behavior of the program being analyzed as described in Section 4. It should be noted that in the total PN, subnets may be nested or combined in any way that reflects the structure of the program.

A Petri net model for the program of Figure 2.2 is shown in Figure 3.2. The augmenting extensions shown here are explained in the next subsection. This PN shows the TR statements, in addition to task-begin and task-end, being modeled by the corresponding subnets defined in Figure 3.1. Also, note that places p11 and p12 are synchronization places, whereas all others are sequential places.

Now we examine some properties of the PN model of a concurrent program obtained as above. Such a model is finite since it is constructed by components (subnets) equivalent to finite TR statements in the program. Thus, the size of the model is linearly proportional to the number of TR statements. Further, the PN model of each task is connected because consecutive subnets will always be connected by merging terminal sequential places. Finally, the PN model is safe] since the weight of each arc is one, the place capacity is one token, and none of the subnet structures allows an accumulation of tokens that exceed the capacity of the places.

3.2 AUGMENTING THE PN MODEL WITH USAGE OF SHARED VARIABLES

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

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. 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. A 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 or may not be concurrent depending upon their position with respect to the rendezvous (synchronizaiton) points in the respective tasks. Zones in different tasks are said to be concurrent if the statements lying in these zones can be excuted concurrently. For example, if two tasks, T1 and T2, communicate and synchronize at point S1 (referring to the two matching rendezvous statements), a zone in T1 before S1 cannot be concurrent with a zone in T2 after S1. For illustration, a concurrent program may be represented by a graph. The nodes of the graph represent zones, vertical edges refer to the sequencing relationship between two 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 given in Figure 3.3, which shows the concurrency zones of the program in Figure 2.2. 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 3.3 next to the graph nodes are RS and WS sets in the respective concrurency zones. These RS and WS sets are shown in Figure 3.2 augmenting the PN's transitions that correspond to the zones. To express the augmentation of the PN, the following is added to the firing rules:

When a transition t_i fires, the shared variables in t_i's concurrency zone are accessed (read or written) and hence the sets RS and WS are formed. The definition of a state of an APN at an instant will also include the sets RS and WS of all tasks at that instant.

With these additions to the firing rules and the definition of APN state, the formation of RS and WS sets is incorporated in a coherent way in the program modeling procedure. The implementation will be described in Section 4.

3.3 ANALYSIS OF APN MODEL

The APN model of a concurrent program is executed to generate a reachability graph augmented with sets of shared variables, referred to as augmented reachability graph (ARG). The ARG is then analyzed to examine the tasking behavior of the underlying parallel program and its usage of shared variables. 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 of the definitions related to PT nets [Peterson 81], adapted here for APN. Then, ARG generation and analysis is illustrated.

An APN state M is defined by a token marking of the net and a collection of pairs of RS and WS sets, with one pair for every concurrent task. 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 APN is reached at which the enabling conditions for the immediate successive transition in FS are satisfied.

By virtue of the addition to the firing rules in subsection 3.2, firing a transition alters not only the marking of the net, but also the sets RS and WS.

A reachability set RS(M) is the set of all states reachable from state M connected by transitions t_i ε FS such that if M_1 ε RS then M_2 ε RS for for some transitions in FS. An augmented reachability graph (ARG) is the set of all reachability sets RS(M_O) for all possible firing sequences FS, where M_O is the initial state of the net. Graphically, a state node in ARG is represented by a token 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. rendevzous, in the concurrent program. The procedure for generating an ARG for APN is the same as that for the PT nets, with RS and WS sets taken into account. The ARG generation procedure starts at an initial state M_O, and repeats a basic step until no more nodes, i.e. state nodes, can be generated. The basic step in the generation procedure is the determination of all enabled transitions at a given state. 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 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 empty, augmenting the generated node.

The generation procedure terminates and yields a finite ARG because ARG corresponds to a finite APN, the reachability graphs of the component subnets of APN are finite, and loops will simply yield a previously generated state so that the same state node is never generated more than once.

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 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 states and potential race conditions on shared variables. Race conditions are detected when more than one task may conflict over the access of shared variables in the same state.

An ARG is depicted in Figure 3.4 for the APN of Figure 3.2. It shows that no deadlock occurs in the program under consideration. Instead, a valid termination state is reached, where tokens reach the end-place p5 and p11 of the two tasks. The ARG also shows that the two tasks may conflict in one of the states in attempting to update the value of the shared variable z.

4. CONCURRENT PROGRAM STATIC ANALYSIS TOOL

A Concurrent Program Static Analysis (CPSA) Tool, which implements the approach explained in the previous section, is described in this section. The CPSA tool consists of four modules which are shown in Figure 4.1. The Modeling (MOD) module produces an APN model of the tasking-related program statements whereas the Augmented Reachability Graph Generator (ARGG) module constructs its augmented reachability graph (ARG). The Reachability Graph Analyzer (RGA) module is composed of various procedures which analyze the information offered by the ARG about the underlying concurrent program. The User Interface (UI) module uses X-Windows software to facilitate interaction with users. The UI module offers 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. Such a user interface facilitates and speeds up the process of isolating and locating errors. The tool has been developed under UNIX environment running on a SUN 3/50 workstation. the MOD, ARGG and RGA modules are written in the C language. The overall size of the tool is about 11,000 lines of code.

The design of the CPSA tool is structured and modular. The UI module has access to files and outputs produced by the other three modules to provide error reports and visual support for analysis; thus facilitating the debugging of the concurrent program being analyzed.

The modular design makes the tool suitable for several programming languages. Language dependency occurs only in a small number of sections of the MOD module and, hence, simple substitutions are sufficient to accommodate different languages. In the present implementation of the CPSA tool, a subset of the Ada language constructs which is sufficient to illustrate the approach is considered. This subset includes the entry call statement, accept-end, if-then-else-end, case-end, loop constructs, select statement and begin-end. Loop conditions are ignored, to avoid combinatorial explosion.

The procedures employed and some implementation considerations for each module of the CPSA tool are described below. Some comments about the complexity of the tool are also presented.

4.1 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, and a table of concurrency zones involving shared variables in different sections of the tasks.

Translation of source code into APN considers only tasking-related statements, that is IP. The translation strategy consists of using Ada subnets as templates or building blocks and connecting these subnets based on either the sequential location of the corresponding statement or information derived from the rendezvous tables. The Ada templates have the same structure as shown in Figure 3.1. Translation is done by scanning the IP statements in sequence, fetching the 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 sets of shared variables in respective concurrency zones, connecting the subnets by combining compatible sequential and synchronization places, and building necessary tables and data structures.

The MOD module consists of three phases. In phase 1, the source code is scanned and filtered to produce an IP. Also NSL is 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 APN model of the underlying program. An outline of the three phases is given below.

<u>Phase 1</u>. While scanning the source code do:

- 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 statement numbers to which they belong and determine whether they occur as Read or Write variables.

<u>Phase 2</u>. While scanning IP do:

- 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. Each row in the table 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.

<u>Phase 3</u>. While scanning IP do:

- For each statement, look up the corresponding template subnet.
- 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.

The program shown in Figure 2.2 with statement numbers is an example of NSL. For this program, IP would retain statements 3, 5, 7, 10, 11, 13, and 15. A task table, a rendezvous table and a concurrency zone table for this example are given in Figure 4.2.

Labels attached to synchronization places in phase 3 are useful in error reports and in analyzing ARG. Examples of important labels are <<request-called task ID-request statement no.>> and <<acknowledge-accepting task ID-request statement no.>>.

4.2 AUGMENTED REACHABILITY GRAPH GENERATOR MODULE

As explained in Section 3.3, an ARG consists 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 permutations at any given state of the APN. A breadth-first generation procedure is presented below. Nodes of the ARG are assigned unique node IDs, a level (with respect to the root) number, the IDs of the input and output arcs (i.e. APN transitions) and pointers to RS and WS sets for all tasks. Other useful data structures are a list of unexplored ARG nodes, UNEXPLORED, and a list of enabled transitions, TRENABLED. A valid termination node is determined by the presence of tokens in the end-places of all tasks.

Procedure

- The root node of the 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:
  - Find the first node in the list, UNEXPLORED.
  - For the new state, search in the neighborhood of places with tokens for enabled transitions (That is, the entire APN need not be searched). Create TRENABLED; in case of structural conflict (if-then-else), add both transitions to TRENABLED.
  - Fire all the enabled transitions in TRENABLED successively, each time starting from the same parent state node.
  - Whenever a transition fires, change the number of tokens in the input and output places and update RS and WS corresponding to the fired transition in the specified task (by using table of concurrency zones with transition ID as index).
  - Add new child state nodes created by firing transitions to UNEXPLORED.
  - Delete nodes from UNEXPLORED if all their transitions in TRENABLED have been fired or if they enable no transitions.

end-repeat

4.3 REACHABILITY GRAPH ANALYZER MODULE

Analysis in the Reachability Graph Analyzer (RGA) module is done on the ARG which is language-independent. It is initiated when requests are made by the user through the UI module. Analysis of ARG aims at providing error reports and some performance information, which may provide insights into factors such as workload balancing and bottelnecks in the concurrent program. The present analysis capabilities of the RGA module are described below. Because of the modular design of the CPSA tool, additional analysis capabilities can be easily added.

Deadlock Detection: Deadlock here refers to any type of blocking or infinite wait encountered by a task. A deadlock state is defined as a non-valid termination state from which no transition can be enabled. Detection of deadlock is performed by searching the ARG for (non-valid-termination) leaf nodes.

A terminal leaf node in the ARG represents either a deadlock state or a valid termination state. A valid termination node represents a marking of the APN where tokens are present in the end-places of all tasks. This marking entails that the list of unexplored nodes is empty and that no tokens are present in any place other than the task end-places.

Concurrent Update of Shared Variables: This refers to the anomaly called potential race condition. It is detected by traversing the nodes of the ARG and performing, for each node, pairwise comparisons for all tasks T_i, where $i = 1, 2, ..., n$, on all $\{RS_i\}$ and $\{WS_i\}$. A race condition on shared data may occur if there exist elements in the intersection of $\{RS_i\}$ and $\{WS_j\}$ or $\{WS_i\}$ and $\{WS_j\}$ for $i = 1, 2, ..., n$, $j = 1, 2, ..., n$ and i different from j. The number of comparisons is not as large as may first seem to be since most of the sets are normally empty and comparisons with empty sets can be dispensed with. Moreover, comparisons need only be performed between the new RS and WS sets, resulting from firing a transition in a task, and the sets for the other tasks. That is, comparisons among unchanged RS and WS sets need not be repeated.

Rendezvous of a Task: The number of rendezvous a task T makes and the identity of the tasks with which rendezvous takes place is determined by searching the ARG nodes for markings where tokens appear in synchronization places with labels "acknowledge-T-statement no". This label refers to places acknowledging acceptance by task T for a task requesting a rendezvous in statement number SN. The statement numbers SN are reported to the user to indicate entry call statements that may result in a rendezvous with task T.

<u>Maximum Number of Rendezvous Queued for a Task</u>: The maximum number of rendezvous that are queued for a task T is determined by searching the nodes of the ARG. In each node, the marking is inspected for tokens that appear in synchronization places with labels "request-T-statement no". The number of rendezvous requests indicated in each node is recorded and the maximum over all nodes is determined.

4.4 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 or redesigning the program. This module enables the user to request analysis information, displays the results produced by the RGA module in a convenient format and allows the user to inspect important data structures. All these facilities are provided with a button-click style of operation in an X-Window environment, which hides the complexity of the tool. The currently implemented facilities are adequate for our objectives. However, they can easily be extended and made more user-friendly.

A typical X-Window screen display is shown in Figure 4.3. Three windows are used in addition to a menu of the available functions. As shown in the figure, the source code filename has to be specified first. Then a number of facilities and analysis functions become available. The first row of functions in the screen display offers general convenient facilities, list, save, etc.

Three windows are used, that allow simultaneous display of different results and the display of a fair amount of information in each window by scrolling it up and down. Window 1 displays status messages, error reports and other analysis information. The status messages are messages about current operation of the tool, such as << Generate Petri net >> and << Generate reachability graph >>. The error and anomaly reports provide results received from the RGA about deadlocks and race conditions on shared variables, if any. Reports about other analysis functions refer to the rendezvous a task can make and the maximum number of rendezvous queued for a task.

Window 2 displays NSL and IP, which can be inspected in association with the error and analysis reports in window 1. In window 3, the important data structures produced by the tool are displayed (by "list"). These are APN, ARG, rendezvous table and concurrency zones table. Viewing these structures simultaneously with error reports and NSL is essential for locating errors and anomalies. The list of places of APN includes, for each place, the label, its ID, and IDs of input and output transitions. An APN transition is viewed by its label, its ID, IDs of input and output places, and RS and WS sets of variables. The ARG is displayed as a list of nodes. The concurrency zones list is also displayed as a table.

4.5 COMPLEXITY OF THE TOOL

The algorithms employed in different modules have different complexities. The complexity of the modeling algorithms is linear in the size of the source code and specifically in the number of the tasking-related statements in the concurrent program. The generation and analysis of the reachability graph activities are exponential in the number of tasks. This complexity tends to set an upper bound on the utility of the tool. However, this complexity can be reduced by collecting analysis information, such as the identification of deadlock states, during the ARG generation and making this information available on demand to the user. The detection of race conditions on shared variables does not add another dimension of exponentiality in the number of tasks as it may seem to be, since the comparison of the elements of RS and WS sets is performed only on updated sets in a node of the ARG. Clearly, only one pair of sets is updated per transition from one node to the next in the ARG. The use of X-window displays in the UI module does not add significant time delay to the total execution time of the tool. It is worth noting that the complexity or the algorithms discussed here is comparable to that for typical general static analysis approaches, such as [Taylor 83a], [Shatz 88a] and [Dillon 88].

5. EXPERIMENTS WITH THE TOOL

Several test programs were employed to assess the correctness of the CPSA tool in detecting of deadlocks and potential race conditions. These test cases include most of the examples that have been used in the literature to illustrate other approaches or to demonstrate their validity.

The test programs for deadlock detection can be broadly classified according to the condition that leads to deadlock. Some test programs involve a mismatch in the number of rendezvous requests and rendezvous accepts in the communicating tasks. This class includes a simple example, which has been employed in [Shatz 88], in which two tasks make entry calls to a task that can only accept one entry. A second class of test programs involves misordering of entry call and accept statements in rendezvous-ing tasks. For example, task T1 may make two entry calls to task T2 with consecutive entry points E1 then E2, whereas T2 accepts rendezvous with entry point E2 first then E1. An example program of the producer-consumer type employed in [Murata 89a] has been included in this class of tests. A third class of test programs involves circular deadlocks which are caused by a set of rendezvous statements, each in a different task, mutually suspending each other and, thus, blocking their respective tasks. A typical example of a circular deadlock is that occurring in the dining philosophers system when each philosopher picks up his left fork and no one picks up the right fork. In this class of test cases we have included, a consumer-producer program from [Murata 89a] and the known gas station problem from [Helmbold 85]. Another set of test programs is deadlock-free, such as an example from [Dillon 88a].

The tool has been tested with test cases from these four classes of concurrent programs. In addition, test cases which include potential race conditions on shared variables have been employed, where global variables are inserted in various concurrency zones. The analysis results of the tool have been correct for this collection of test programs. That is, the tool reports a statically detectable deadlock or a race condition when such anomalies and errors are known to be present and does not when they are known to be absent in these specific programs.

Three example sets are presented below only in enough detail to explain our approach. Example set 1 describes deadlock detection. Example set 2 describes detection of potential race conditions. The two features are integrated in the tool and are separated in these examples only for clarity purposes. Example set 3 is included to point out the limitations of this tool as well as of static analysis based approaches for testing concurrent programs.

Example Set 1 An Ada program, shown in Figure 5.1, is chosen to represent this set. This program results in a deadlock because task A is making two rendezvous requests to task B, whereas task B is accepting only one rendezvous. The Petri net produced by the modeling module is shown in Figure 5.2 and its reachability graph is depicted in Figure 5.3. In both figures, the RS and WS sets are omitted since they are empty in this example. A sample user display is given in Figure 4.3, where the error report, the numbered program statements (i.e. NSL), and parts of the PN list (places and transitions) are shown.

Example Set 2 The Ada program given in Figure 2.2 is used to represent this set. It involves the shared variables w and z. The anomaly of race conditions may occur over the global variable z used in the respective concurrency zones of tasks SENDER and RECEIVER. this is illustrated in Figures 3.2, 3.3 and 3.4.

Example Set 3 A number of programs of various sizes and task interaction rates have been examined. Programs with more than two tasks require either a large amount of memory storage or a long analysis time by the tool. This will be discussed in the next section.

6. DISCUSSION AND EXTENSIONS

A static analysis tool has been described for providing reports about deadlocks and potential race conditions on shared variables. It also provides other useful analysis information in Ada concurrent programs. In the tool, a concurrent program is first translated into an augmented Petri net model. Then, upon user requests, an augmented reachability graph is derived and analyzed to yield desired information about the original concurrent program. User requests are made through an X-window-based interface.

Although Ada has been chosen as a rendezvous-based programming notation, the approach upon which the tool is based is not language dependent. Simple substitutions in the modeling module of the tool can accommodate other rendezvous-based notations, such as CSP [Hoare 78] and its variants.

The utility of the tool is expressed in its ability to report important syntax-based deadlocks and anomalies and to provide useful analysis information. Our experience with the tool and its validation has shown its usefulness in capturing many deadlock errors and cases of race conditions on shared variables. We feel that its use in analyzing concurrent programs does enhance our confidence about the correctness of these programs with respect to such errors and anomalies.

The tool suffers from some of the limitations shared by all approaches based on pure static analysis and exhaustive search. These limitations can be described as lack of program semantics and combinatorial explosion of the reachability graph. The lack of program semantics may lead to infeasible paths in the search space which can produce spurious error reports. However, such reports can be minimized by adding features that capture the effects of decision control statements leading the to pruning of the infeasible paths. The exponential growth of the reachability graph can be satisfactorily overcome either by selective generation and analysis of the graph or by Petri net reduction. Net reduction techniques have been proposed in [Berthelot 86] and [Lee 87]. Based on these techniques, algorithms can be worked out and then incorporated in the CPSA tool. Work is underway to enhance the Petri net model for minimizing spurious error reports, and to reduce the search space.

REFERENCES

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).

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 Workshop on Software Testing, Verification and Analysis, pp. 114-122 (1988).

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

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).

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

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, 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, "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).

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).

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

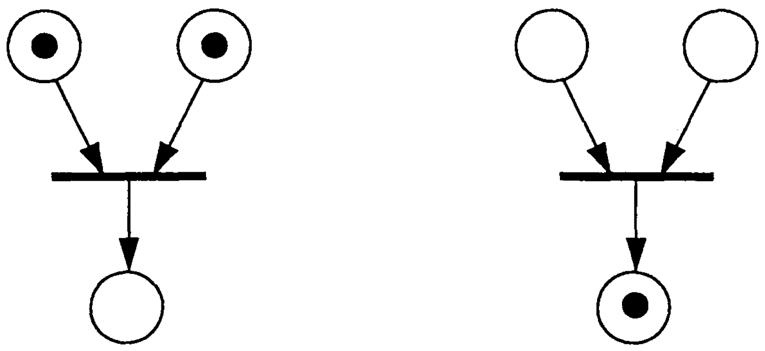


Figure 2.1 A PT net before and after firing a transition.

```
Task body SENDER is
   story  : integer;
. . .
begin
   create (story);
   RECEIVER. takemessage (story);
   z := story + w,
end SENDER

Task body RECEIVER is
   y  : integer;
. . .
begin
   accept takemessage (message : in integer) do
   z   : = message + y;
   end,
   z := message - w;
end RECEIVER
```

Figure 2.2 An example illustrating Ada constructs for rendezvous. (Variables z and w are assumed to be global)

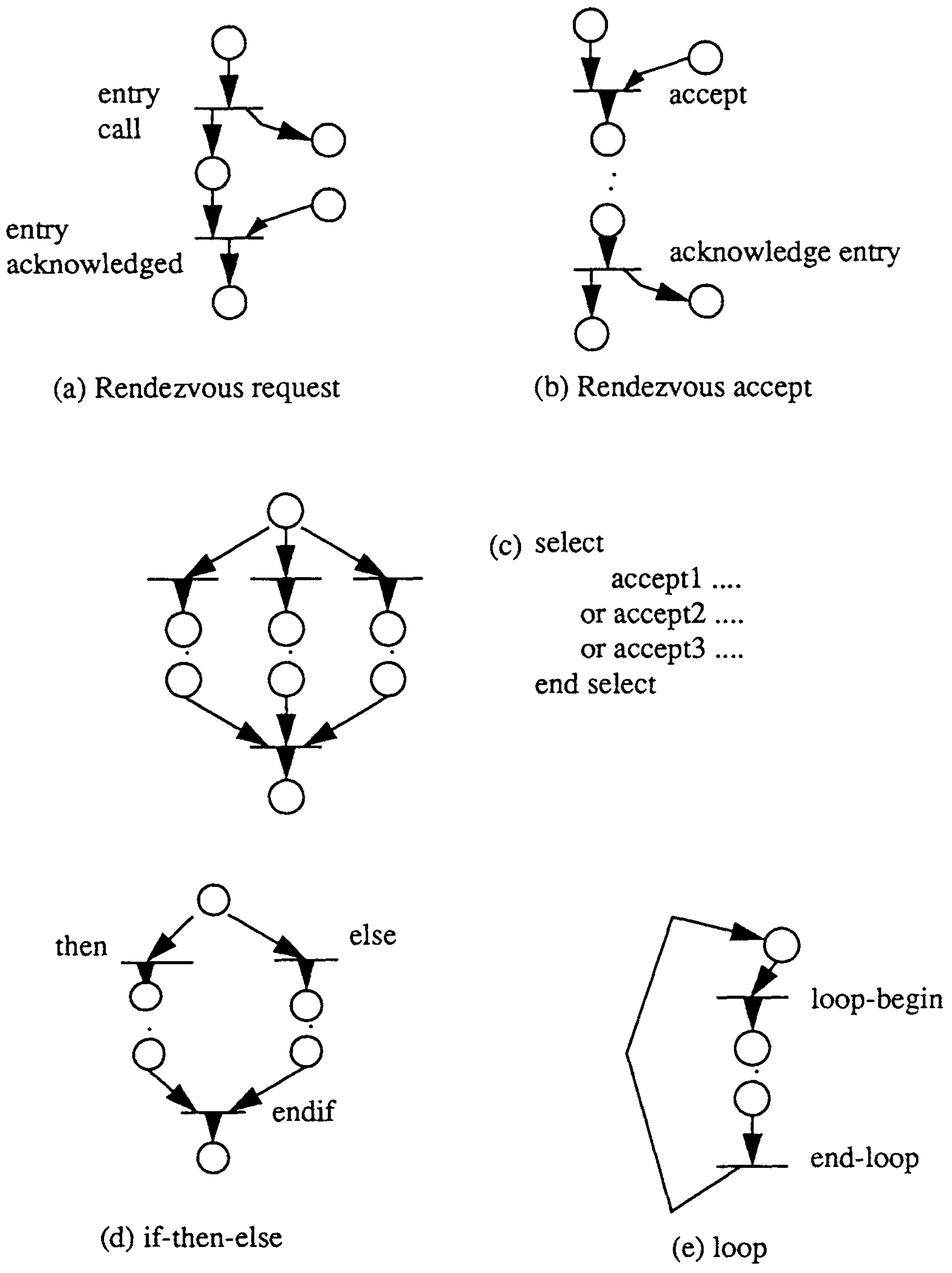


Figure 3.1 PN subnets corresponding to TR statements.

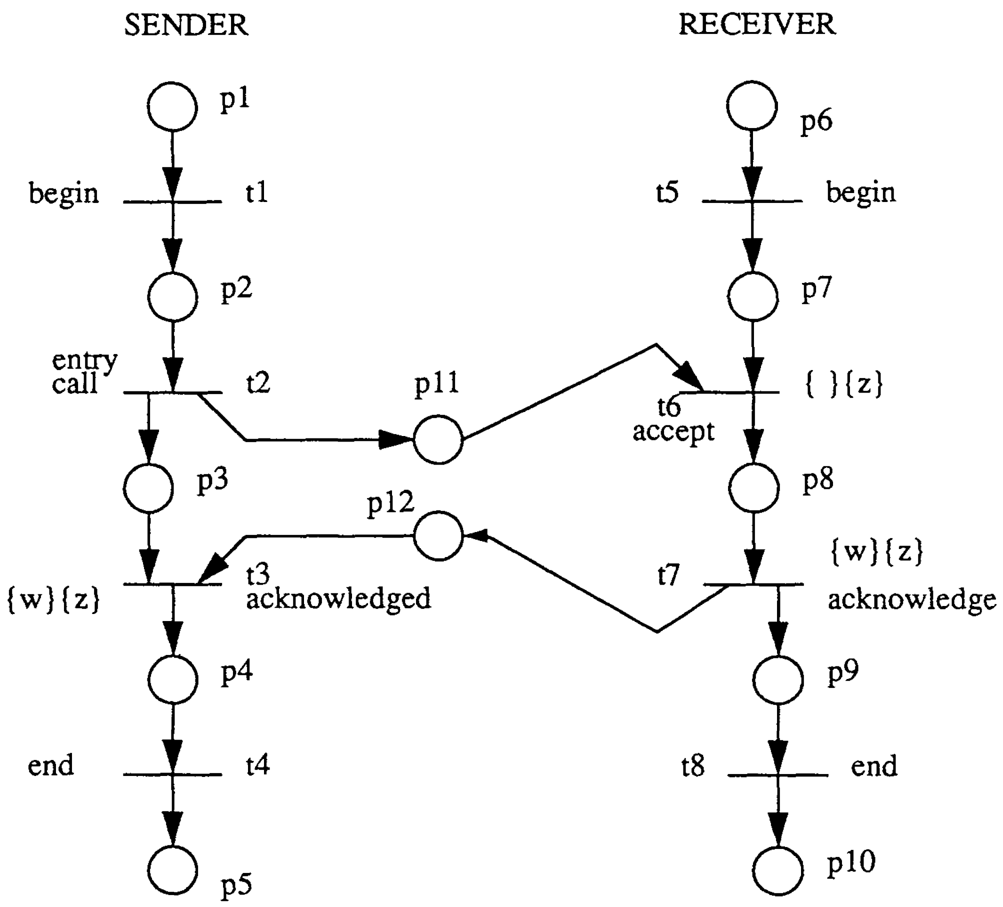


Figure 3.2 Augmented Petri net model for the program in Figure 2.1.
(Augmenting sets are in the order : RS , WS)

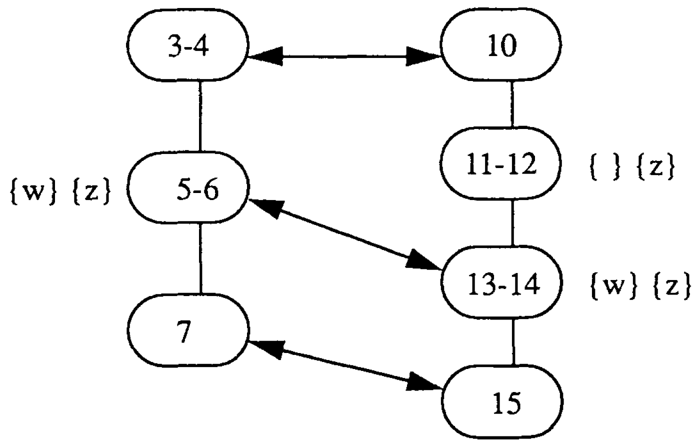


Figure 3.3 Graph of concurrency zones for the program in Fig. 2.1.
(RS and WS sets are shown next to the relevant nodes)

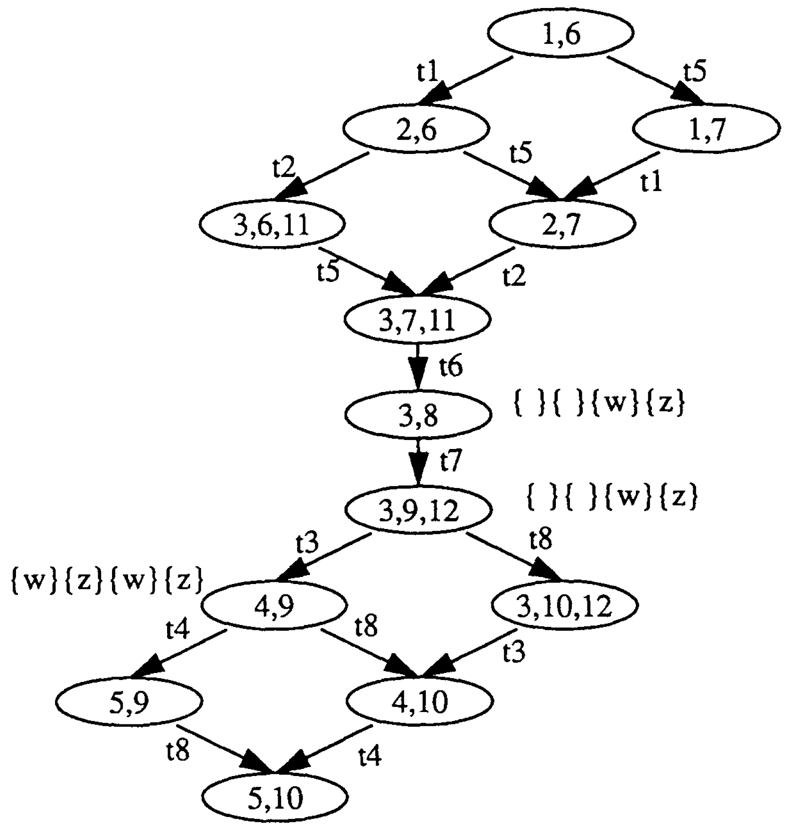


Figure 3.4 Augmented reachability graph for the APN in Figure 3.2.
(Augmenting sets are in the following order:
RS of SENDER, WS of SENDER, RS of RECEIVER,
WS of RECEIVER)

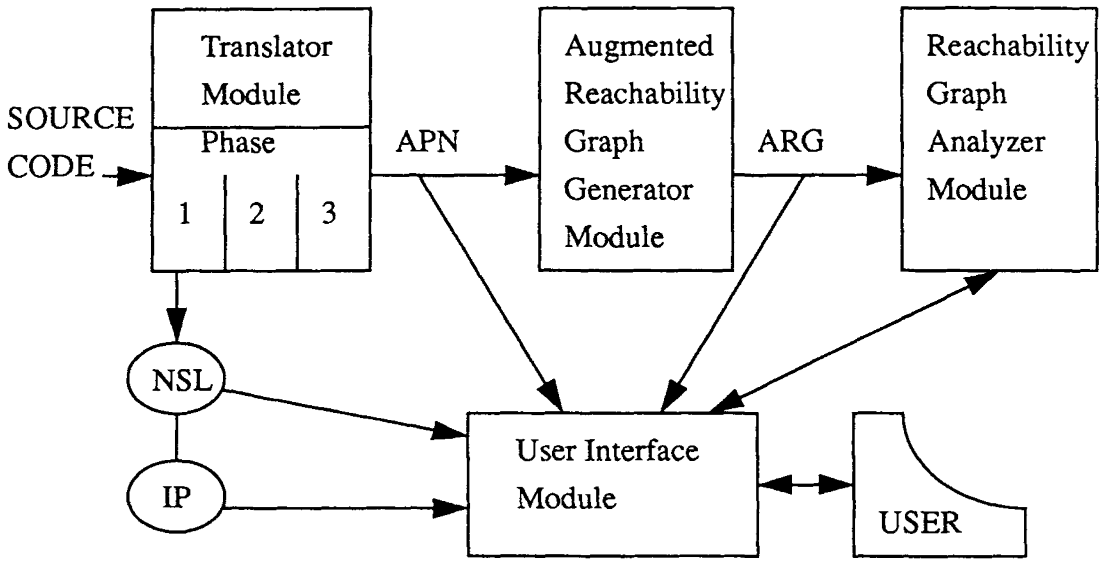


Fig. 4.1 Concurrent Program Static Analysis (CPSA) Tool System.

Task Name	ID
SENDER	1
RECEIVER	2

(a) Task Table

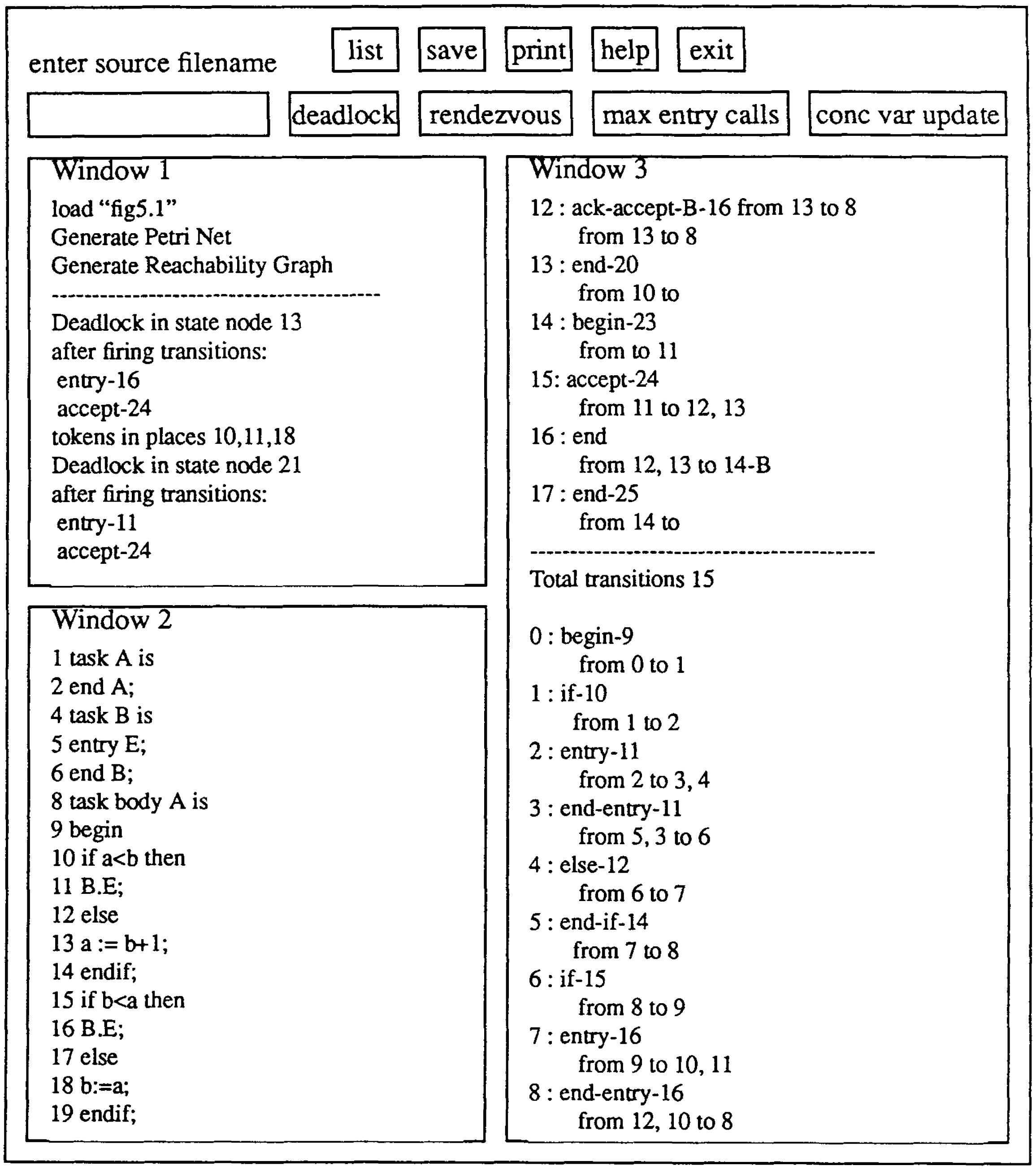
Calling Task	Accepting Task	Entry Point	Line No.
1	2	takemessage	5

(b) Rendezvous Table

Task ID	Start #	Finish #	RS	WS
1	3	4	-	-
1	5	6	w	z
1	7	7	-	-
2	10	10	-	-
2	11	12	-	z
2	13	14	w	z
2	15	15	-	-

(c) Concurrency Zone Table

Figure 4.2 Task table, Rendezvous table and Concurrency zone tables for the Program of Figure 2.1



```
load "fig5.1"
Generate Petri Net
Generate Reachability Graph
----------------------------------------
Deadlock in state node 13
after firing transitions:
 entry-16
 accept-24
tokens in places 10,11,18
Deadlock in state node 21
after firing transitions:
 entry-11
 accept-24
```

```
1 task A is
2 end A;
4 task B is
5 entry E;
6 end B;
8 task body A is
9 begin
10 if a<b then
11 B.E;
12 else
13 a := b+1;
14 endif;
15 if b<a then
16 B.E;
17 else
18 b:=a;
19 endif;
```

```
12 : ack-accept-B-16 from 13 to 8
      from 13 to 8
13 : end-20
      from 10 to
14 : begin-23
      from to 11
15: accept-24
      from 11 to 12, 13
16 : end
      from 12, 13 to 14-B
17 : end-25
      from 14 to
------------------------------------------
Total transitions 15

0 : begin-9
      from 0 to 1
1 : if-10
     from 1 to 2
2 : entry-11
      from 2 to 3, 4
3 : end-entry-11
      from 5, 3 to 6
4 : else-12
      from 6 to 7
5 : end-if-14
     from 7 to 8
6 : if-15
      from 8 to 9
7 : entry-16
      from 9 to 10, 11
8 : end-entry-16
      from 12, 10 to 8
```

Fig. 4.3 X-Window screen for CPSA tool.

```
task A is
end A;

task B is
  entry E;
end B;
```

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

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

Figure 5.1 An Ada program that contains a deadlock.

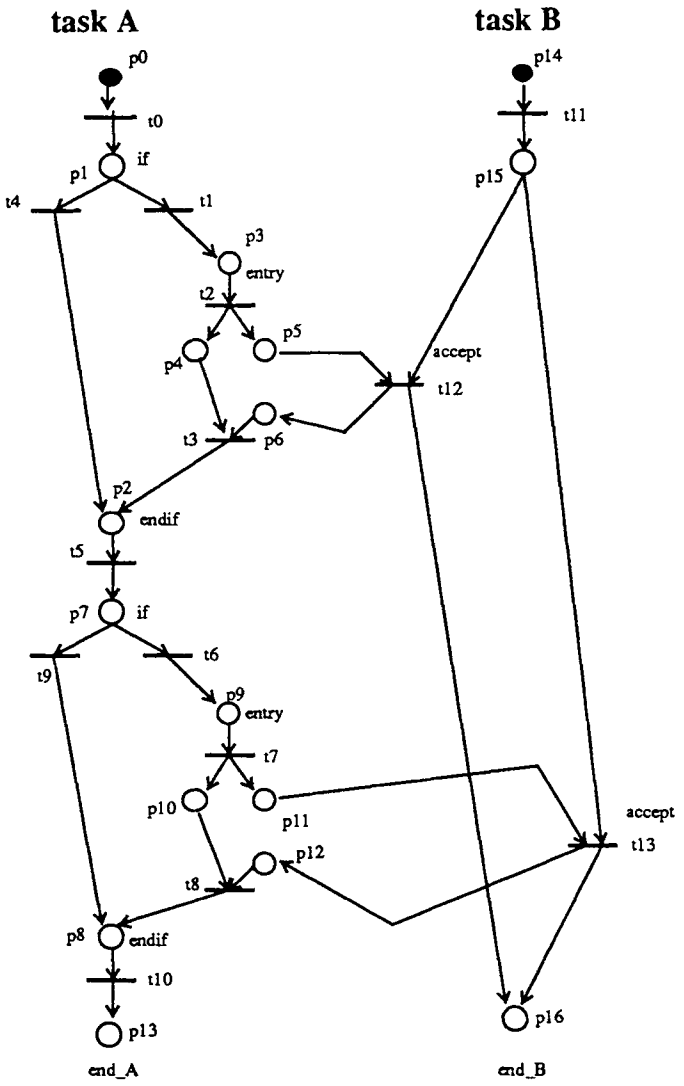


Figure 5.2 Petri net model of the program in figure 5.1.

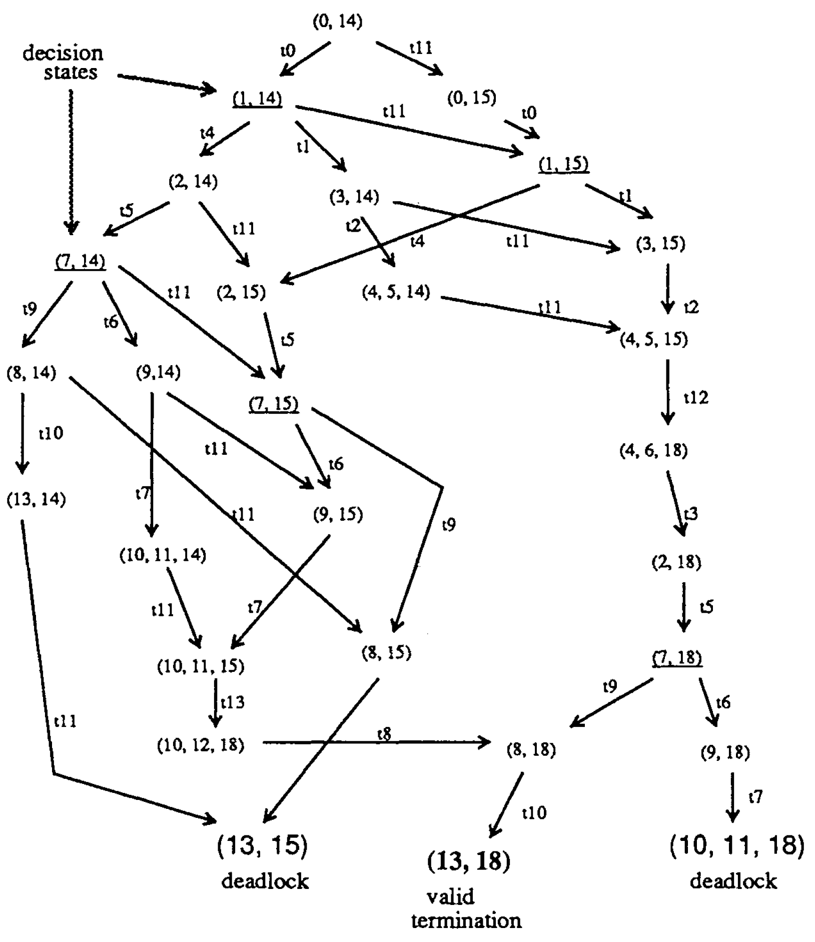


Figure 5.3 Reachability Graph of the Petri Net in figure 5.2.