TIG-based Petri Nets for Modeling Ada Tasking

Kari Forester
Lori Clarke
Matthew Dwyer

COINS Technical Report 91–69
June 1991

Software Development Laboratory
Computer & Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by National Science Foundation grant CCR-8704478 in cooperation with Defense Advance Research Projects Agency (ARPA Order No. 6104, ARPA Program Code No.7E20)

TIG-based Petri Nets for Modeling Ada Tasking

Kari Forester
Lori Clarke
Matthew Dwyer

Department of Computer and Information Science
University of Massachusetts, Amherst *

June 24, 1991

Abstract

This paper presents a Petri net model for concurrent programs that is based on an internal representation for concurrent programs called *task interaction graphs* (TIGs). Comparisons are made between these TIG-based Petri nets and an existing Petri net model for Ada tasking.

*This work was supported by National Science Foundation grant CCR-8704478 in cooperation with Defense Advanced Research Projects Agency (ARPA Order No.6104, ARPA Program Code No.7E20)

Contents

1 Introduction 3

2 Petri nets 4

3 Task Interaction Graphs 9

4 TIG-based Petri net model 18

5 Comparisons 23

6 Reachability analysis 29

7 Conclusion 29

1 Introduction

As hardware becomes increasingly inexpensive, parallel computing is becoming more feasible and promises to be the predominant computational model of the future. Because more and more critical software will be implemented as concurrent systems, it is important that we be able to ensure that such systems are highly reliable.

Analyzing concurrent software is a complex and difficult task. Asynchronously running processes produce an exponential number of interleavings that are hard to analyze with static methods. Non-deterministic communication patterns can produce erroneous interleavings, such as deadlock or race conditions, that can be difficult to discover or reproduce. Analysis and verification of concurrent programs is an active area of research. Many different models for static analysis exist, including Petri nets [MR87, MSS89, SMBT90], state based models [Tay83b, LC89, YTFB89, McD89, YY90], data flow [TO80, DS91, LC91], and constrained expressions [ADWR86, ABC+90]. Work has also been done in the area of dynamic testing [HL85, TK86, Tai85]. Still others have focused on the formal specification and verification of concurrent programs [OG76, Lam83, Dil90]. A more recent area of research is the analysis of concurrent real-time systems [FGM89, ACDW90]. In this paper, we will look at two models for tasking programs: a well developed Petri net model and a recently proposed model called *task interaction graphs.*

Petri nets have been used successfully to model and analyze concurrent systems [MR87, SC88, MZGT85]. Murata classifies all analysis methods on Petri nets into three groups: 1) reachability tree method, 2) matrix-equation approach, and 3) reduction or decomposition techniques [Mur89]. The reachability graph of a Petri net has been used to perform state-space analysis on concurrent systems [SC88, MR87]. More recently, invariants have been used to detect deadlocks in Ada tasking programs [MSS89] and in [TST] a set of reduction rules are given that preserve liveness properties, allowing deadlock detection to be performed on reduced Petri nets.

Another approach to modeling concurrent systems has been introduced by Taylor [Tay83b]. In this and subsequent work [LC89, YTFB89], tasks are modeled as individual flow graphs. We will refer to this family of models as *task flow models.* Taylor uses *task flowgraphs* to generate *control flow currency graphs.* These concurrency graphs contain all of the possible concurrency states of the system and are analogous to the reachability graph derived from Petri nets. Long and Clarke have proposed a more optimal model for representing tasking programs called *Task Interaction Graphs* (TIGs) [LC89]. TIGs can be used to generate more compact reachability graphs called *Task Interaction Concurrency Graphs* (TICGs). Reachability graph generation has been shown to be intractable as the number of possible program states grows exponentially with the number of tasks involved [Tay83a]. Addressing this problem, recent work has been done in reducing the size of a reachability graph by collapsing states into *clans* [McD89]. Also a compositional approach that uses a divide and conquer method allowing portions of a large system to be analyzed independently of one another is proposed by Yeh and Young [YY90].

The focus of this paper is to present a new Petri net model for tasking programs. This model is obtained via a translation from *task interaction graphs.* We compare our TIG-based model with an existing Petri net model and show that the TIG-based model offers some benefits in terms of the size of the Petri net and the reachability graph that is derived from it.

In the following two sections we review Petri nets and *task interaction graphs.* Section 2 briefly defines Petri nets and discusses their applicability to concurrency analysis. We then examine an existing Petri net model for Ada [ALR83] tasking called Ada-nets. Section 3 reviews *task interaction graphs* and shows how they have been useful in modeling and analyzing concurrent programs. Section 4 defines TIG-based Petri nets and section 5 makes comparisons between Ada-nets and TIG-based Petri nets. Section 6 compares both models in terms of the reachability graphs that they generate and section 7 concludes.

2 Petri nets

Petri nets are a graphical formalism useful for specifying concurrent systems. A Petri net is a directed graph with two node kinds: *places* and *transitions* where places are drawn as circles and transitions as bars. The directed arcs of the graph are either from a place to a transition or from a transition to a place. A *marking* is an assignment of an integer to each place in the net representing the number of *tokens* at that place. Tokens are drawn by placing zero or more black dots in each place. A marking is given by a m-vector, M, where m is the number of places in the net and M(p) denotes the number of tokens at place p. The initial marking of the net is written as M_0. Arcs can be labeled with a positive integer representing their *weight.* A k-weighted arc can be equivalently represented as k parallel 1-weighted arcs.

Definition

A formal definition of a Petri net, taken from [Mur89], is given below.

A Petri net is a 5-tuple, PN = (P,T,F,W,M_0) where:

P = p_1, p_2,...,p_m is a finite set of places,
T = t_1, t_2,...,t_n is a finite set of transitions,
F $\subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
W: F $\rightarrow$ 1 2,3,... is a weight function,
M_0: P $\rightarrow$ 0,1,2,3,... is the initial marking,
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

Petri nets can be used to model system behaviors using the concepts of conditions and events, where places represent conditions and transitions represent events. Tokens represent the truth of a one of the conditions associated with a place. Each transition has a fixed number of input and output places that represent the pre and post conditions of the event. The state of the net changes according to the following firing rule:

A transition is enabled if each input place of the transition is marked with at least as many tokens as the weight given on the associated input arc. A transition may not fire unless it is enabled. A firing of a transition t removes w(p_i,t) token from each input place p_i and adds w(t,p_o) tokens to each output place p_o.

Reachability graphs

Given any Petri net, we can derive a reachability graph that enumerates all possible marking of the net. Each node in the reachability graph represents a marking of the Petri net and each edge represents the firing of a transition. Starting from the initial marking, M_0, we can generate all possible next markings by considering all enabled transitions. For each enabled transition, we create a new node that represents the marking of the net once that transition has fired. An edge, labeled with the fired transition, is added from the initial state to each new state. From the each new marking we can again generate a set of markings and repeat until no new markings exist. Reachability graphs are useful in analyzing Petri net models of concurrent systems, because they enumerate all possible states of the system. They can be used to detect deadlock and trace concurrency histories.

Ada-net model

A Petri net model for Ada tasking is found in Shatz's toolkit for static analysis of tasking behavior [SC88]. Ada tasking programs are parsed and translated to a Petri net represen-

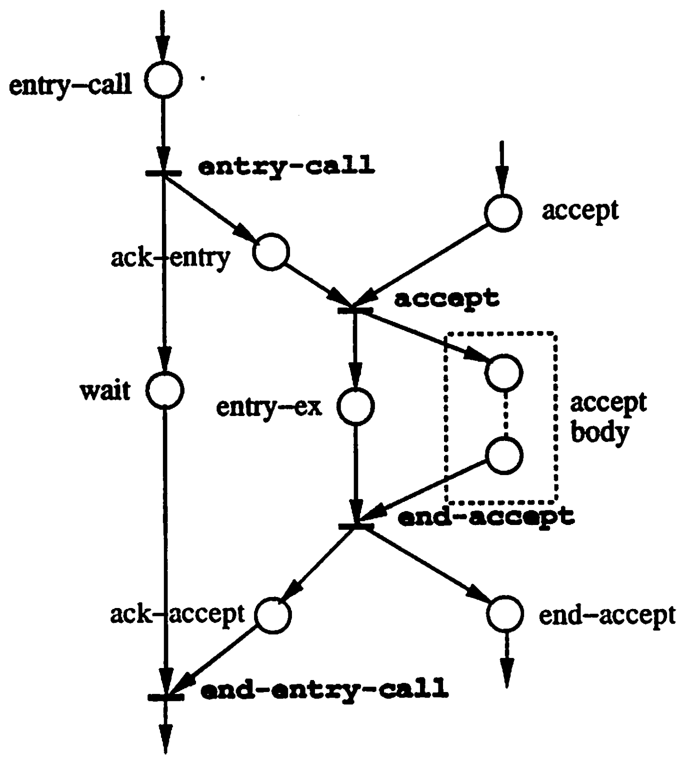

Figure 1: Petri net representation for Ada rendezvous

tation of the system using a set of translation templates. The resulting Petri nets are called Ada-nets. A reachability graph is generated from the resulting Petri net that is then used to analyze tasking behavior properties.

The representation for an Ada rendezvous in this Petri net model is illustrated in figure 1. With this approach, control flow constructs that can alter the program's tasking behavior, such as loops, if-then-else clauses and select statements, are modeled explicitly. Consider the simple Ada tasking program from figure 2. This example is given in [SC88] and an illustration of the corresponding Ada-net representation is found if figure 3. In order to understand the model we must be able to interpret the meaning of the places and transitions of the Petri net. We attempt to do this by assigning a condition to each place and an event to each transition:

places (conditions):

P1: task T1 has begun

```
task body T1 is
begin
    T2.E
end T1;

task body T2 is
begin
    accept E do
        ...
    end E;
end T2;

task body T3 is
begin
    T2.E
end T3;
```

Figure 2: Simple example

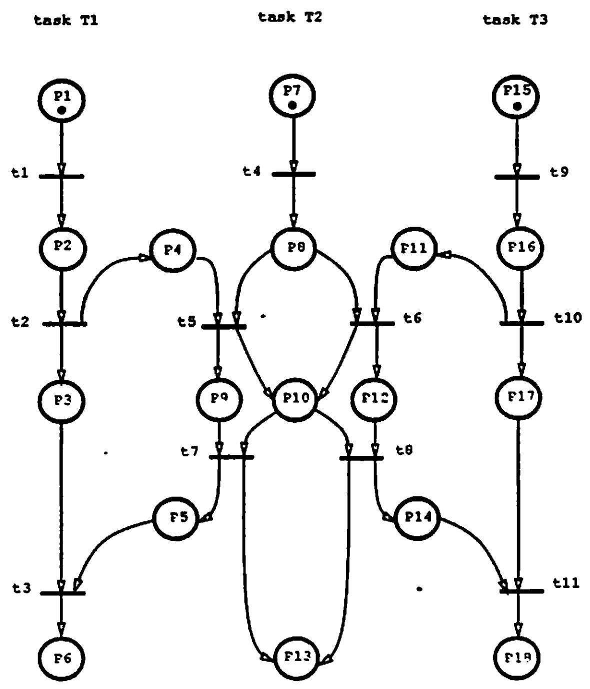

Figure 3: Ada-net representation for tasks in figure 2

P2: task T1 wishes to make an entry call to T2.E
P3: task T1 is waiting for an ackn. that the rendezvous is complete
P4: task T1 is queued to rendezvous with T2.E
P5: task T2 signals T1 that the rendezvous is complete
P6: task T1 continues with next statement after rendezvous
P7: task T2 has begun
P8: task T2 is ready to accept an entry call on E
P9: task T1 is rendezvousing with task T2
P10: task T2 is executing the accept body for E
P11: task T3 is queued to rendezvous with T2.E
P12: task T3 is rendezvousing with task T1
P13: task T2 continues with next statement after rendezvous
P14: task T2 signals T3 that the rendezvous is complete
P15: task T3 has begun
P16: task T3 wishes to make an entry call to T2.E
P17: task T3 is waiting for an ackn. that the rendezvous is complete
P18: task T3 continues with next statement after rendezvous

transitions (events):

t1: task T1 completes code before first task interaction
t2: task T1 makes entry call on T2.E
t3: task T1 exits rendezvous with T2.E
t4: task T2 completes code before first task interaction
t5: task T2 begins accept E with task T1
t6: task T2 begins accept E with task T3
t7: task T2 exits accept E with task T1
t8: task T2 exits accept E with task T3
t9: task T3 completes code before first task interaction
t10: task T3 makes entry call on T2.E
t11: Task T3 exits rendezvous with T2.E

3 Task Interaction Graphs

Task interaction graphs (TIGs) have been proposed by Long and Clarke [LC89] as a compact representation for tasks that is amenable for analysis. TIGs provide a useful abstraction by dividing tasks into maximal sequential regions, where such task regions define all of the possible behaviors of a task from one task interaction to the next. A task interaction is defined as any point where the behavior of one task may be influenced by the behavior of another task. TIGs offer a natural representation for interacting tasks that we believe will provide a basis for several different analysis techniques.

TIG definition

We now provide a brief description of *task interaction graphs* that is sufficient in allowing the reader to understand the examples presented in this paper. For a more complete description of TIGs and their derivation from Ada tasking code, the reader is advised to refer to [LC89].

The nodes of a *task interaction graph* represent a maximal sequential region of the task. An explicit code representation for each region is associated with a corresponding node of the graph. Such a representation must be able to indicate the entry points and exit points of a region. A region has a single entry point and potentially many exit points. *Pseudocode* is used to represent the code in task regions where the pseudocode is Ada plus two non-executable statements, ENTER and EXIT.

The edges of a *task interaction graph* are labeled with the tasking interactions that cause transitions from one region to the next. The tasking interactions we will consider are confined here to entry calls and accept statements. There are four distinct kinds of tasking interactions: starting an entry call, ending an entry call, starting an accept statement, and ending an accept statement. It is necessary to model both the start and end of a rendezvous explicitly because information may be exchange between tasks via parameters. At the start of a rendezvous, the accepting task may be passed information from the calling task, thus changing its environment and necessitating a new task region. Likewise, at the end of a rendezvous the calling task may be passed information, also causing a new task region. Although the start of a rendezvous does not change the environment of the calling task and the end of a rendezvous does not change the environment of the accepting task, both tasks are divided into two regions at both the start and the end of a rendezvous in order to facilitate keeping track of the synchronization between the calling and accepting tasks. There are instances, however, where a simplified representation can be used as we show later.

A TIG is defined formally as a tuple (N,E,S,T,L,C), where N is the set of nodes in the graph, E is the set of edges, S is the start node representing the region where the task begins execution, T is a set of terminal nodes representing regions where a task may potentially

$$
\begin{aligned}
N &= \{1,2,3\} \\
E &= \{(1,2),(2,3)\} \\
S &= 1 \\
T &= \{3\} \\
L(1,2) &= T2.E_S \\
L(2,3) &= T2.E_E
\end{aligned}
$$

```
C(1) = ENTER(TASK_ACTIVATE);
       task body T1 is
       begin
          EXIT(CALL_START(T2.E), 2);

C(2) = ENTER(CALL_START(T2.E));
       EXIT(CALL_END(T2.E), 3);

C(3) = ENTER(CALL_END(T2.E));
       end T1;
       EXIT(TASK_TERMINATE, nil);
```

Figure 4: Formal Task Interaction Graphs for task T1 of figure 2

finish execution, L is a function that assigns a label to each edge, and C is a function that assigns the pseudocode for a tasking region to each node.

Examples

Now we will look at some Ada tasking examples to demonstrate some features of TIGs. First we will consider the simple tasking example of figure 2. This example has three tasks: T1, T2, and T3. Formal TIG descriptions of T1 and T2 are found in figures 4 and 5 respectively. The TIG for task T3 is essentially identical to that of T1.

In the pseudocode, entry calls and accepts statements are replaced by the pseudostatements ENTER(*interaction*) and EXIT(*interaction*, *next*), where *interaction* refers to the type of the interaction that causes a transition from one region to another and *next* is the region that is entered after an exit. The four possible interaction types are: CALL_START, CALL_END,

$$\begin{aligned}
N &= \{4,5,6\} \\
E &= \{(4,5),(5,6)\} \\
S &= 4 \\
T &= \{6\} \\
L(4,5) &= E_S \\
L(5,6) &= E_E
\end{aligned}$$

$C(4) =$

```
ENTER(TASK_ACTIVATE);
task body T2 is
begin
    EXIT(ACCEPT_START(E), 5);
```

$C(5) =$

```
ENTER(ACCEPT_START(E));
...
end E;
EXIT(ACCEPT_END(E), 6);
```

$C(6) =$

```
ENTER(ACCEPT_END(E));
end T2;
EXIT(TASK_TERMINATE, nil);
```

Figure 5: Formal Task Interaction Graphs for task T2 of figure 2

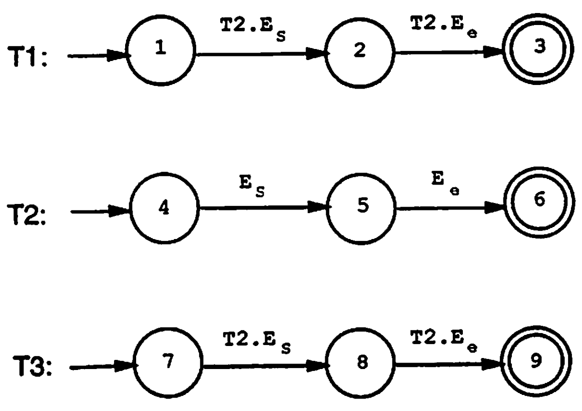

Figure 6: Task interaction graphs

ACCEPT_START and ACCEPT_END. TIG edges represent tasking interactions and are labeled with the type of interaction that causes the transition from one region to the next. In this example TIG labels are abbreviated to E_S, E_E, $T2.E_S$ and $T2.E_E$ where the S and E subscripts stand for start and end respectively.

Drawings of the TIGs for tasks T1, T2, and T3 of figure 2 are given in figure 6. An unrooted arrow points to the start node of a TIG and a double circle around a node indicates a terminal node.

We now consider a more complicated example that contains both branching and looping constructs. As mentioned earlier, TIG regions can have multiple exit points. This may occur when there is a task interaction on a branch of a conditional statement. Consider the example tasks given in figure 7. The pseudocode for task T1 is given in figure 8. Task T1 is divided into 5 sequential regions. Region 1 enters at the beginning of the task and exits at the select statement. There are two exits out of this region. The first exit is on the start of the accept for E1 and the second is on the start of the accept for E2. Region 2 enters at the start of the accept for E1 and exits at the end of the accept for E1. This region is empty because the accept statement for E1 has no accept body. Region 3 is similar. Region 4 enters at the end of the accept for E1 and exits at the select statement. Like region 1, it also has two exits, depending on which branch of the select is chosen. Region 5 is similar. It is interesting to note that task interactions within loops always result in some duplication of code in the regions. Any code from the beginning of the loop to the first task interaction will appear in two regions: the region between the interaction that precedes the loop and the interaction embedded within the loop and the region between two executions of the interaction within the loop. Drawings of the TIGs for both tasks T1 and T2 are given in figure 9.

```
task body T1 is
begin
    loop
        select
            accept E1;
        or
            accept E2;
        end select;
    end loop;
end T1;

task body T2 is
begin
    loop
        T1.E1;
        T1.E2;
    end loop;
end T2;
```

Figure 7: Ada tasking example

```
C(1)   =   ENTER(TASK_ACTIVATE);
           task body T1 is
           begin
               loop
                   select
                       EXIT(ACCEPT_START(E1), 2);
                   or
                       EXIT(ACCEPT_START(E2), 3);
                   end select;
               end loop
           end T1;

C(2)   =   ENTER(ACCEPT_START(E1));
           EXIT(ACCEPT_END(E1), 4);

C(3)   =   ENTER(ACCEPT_START(E2));
           EXIT(ACCEPT_END(E2), 5);

C(4)   =       loop
                   select
                       EXIT(ACCEPT_START(E1), 2);
                       ENTER(ACCEPT_END(E1));
                   or
                       EXIT(ACCEPT_START(E2), 3);
                   end select;
               end loop
           end T1;

C(5)   =       loop
                   select
                       EXIT(ACCEPT_START(E1), 2);
                   or
                       EXIT(ACCEPT_START(E2), 3);
                       ENTER(ACCEPT_END(E2));
                   end select;
               end loop
           end T1;
```

Figure 8: Pseudocode for task T1 of figure 7

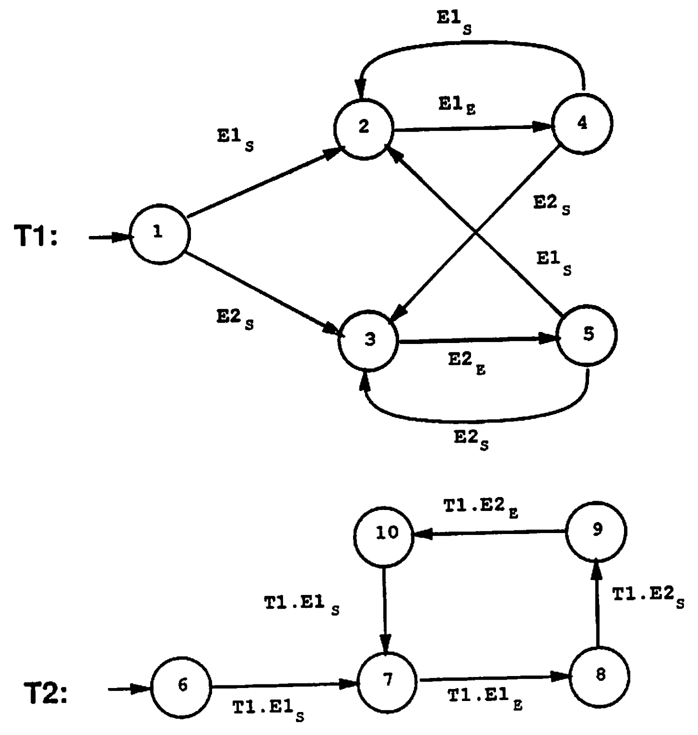

Figure 9: TIGs for tasks T1 and T2 of figure 7

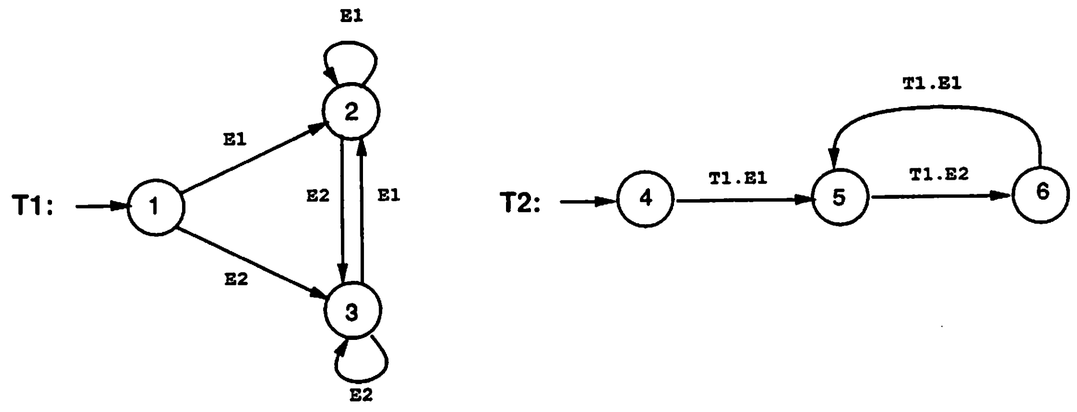

Figure 10: Reduced TIGs for tasks T1 and T2 of figure 7

Reduced task interaction graphs

Finally, as we mentioned earlier, there are instances where accept statements and entry calls can be modeled by a reduced representation. If the accept statement of a rendezvous has no accept body then it can be modeled by two nodes instead of three. Only a single interaction, comprising both the start and the end of the rendezvous is needed to model such an accept statement and any entry calls made on it. Two new task interaction types are introduced: CALL_START_END and ACCEPT_START_END. For TIG labels on edges that represent such interaction types, the S and E subscripts are omitted. Since the accept statements given in task T1 of figure 7 have no accept bodies, the TIGs for tasks T1 and T2 can be reduced as shown in figure 10.

Task interaction concurrency graphs

TIGs have been shown to offer a good basis for reachability analysis. Unlike previously proposed representations, such as Taylor's task flowgraphs which model dataflow information explicitly, TIGs abstract away control flow information providing a more compact representation. Reducing the number of nodes necessary to represent a task significantly reduces the size of the resulting concurrency graph.

An algorithm for constructing a concurrency graph from a set of task flowgraphs was first presented by Taylor in [Tay83b]. A similar algorithm is used to construct a TICG from a set of TIGs. A TICG node represents the state of the entire system. Given a set of n TIGs,

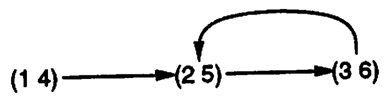

Figure 11: TICGs for TIGs of figure 10

a TICG node is an ordered n-tuple, $< t_1, t_2, ..., t_n >$, where t_i corresponds to the state (or current tasking region) of TIG_i. The nodes and edges of a TICG are defined by a successor relationship. The initial TICG state is obtained by creating an n-tuple where each element, t_i, corresponds to the start node of TIG_i. Node $< s_1, s_2, ...s_n >$, is a successor of node $< t_1, t_2, ...t_n >$ if and only if:

1. there exists i and j such that (t_i, s_i) and (t_j, s_j) are edges in TIG_i and TIG_j respectively and the labels on these edges represent a potential task interaction.

2. for all $k \neq$ i or j, $t_k = s_k$.

From each node we create a directed TICG edge to each of its successors. A TICG is constructed by taking the transitive closure of the successor relationship on the initial TICG node. As an example, the TICG constructed from the set of TIGs in figure 10 is illustrated in figure 11.

4 TIG-based Petri net model

Using a TIG-based representation for the same simple tasking example, we can produce a Petri net model that abstracts away much of the control flow and Ada tasking mechanisms modeled explicitly by Shatz. We propose a model where each place in the Petri net has a one-to-one correspondence with a TIG tasking region and each transition represents a task interaction. A TIG-based Petri net representation for an Ada rendezvous is given in figure 12. As with TIGs, if there is no accept body, the transitions can be combined to a single *rendezvous-start-end* event and the intermediate regions can be removed.

Translation

Translating TIGs into Petri nets is simple and straight forward. Given a set of TIGs, create a unique place for each TIG region. For each pair of edges that represents a potential

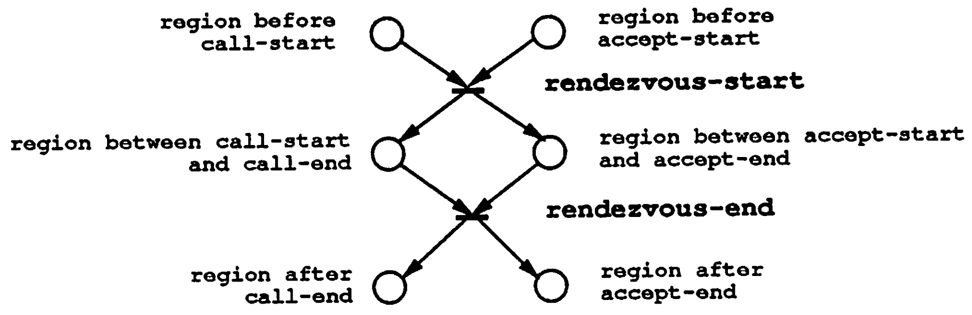
region before call-start
region before accept-start
rendezvous-start
region between call-start and call-end
region between accept-start and accept-end
rendezvous-end
region after call-end
region after accept-end

Figure 12: TIG-based Petri net representation for Ada rendezvous

tasking interaction, create a transition whose input places are the places corresponding to the regions that exit on those edges and whose output places are the places corresponding to the regions that enter on those edges. For the initial marking, M_0, of the Petri net, we put a single token at each place that corresponds to a start region in the set of TIGs. A similar algorithm is presented in [PTY].

As an example, recall the simple tasking program of figure 2, whose TIGs is given in figure 6. Using the above algorithm we would construct the TIG-based Petri net illustrated in figure 13. Places P1–P3 correspond to regions 1–3 in the TIG representing task T1, places P4–P6 correspond to regions 4–6 in the TIG representing task T2, and places P7–P9 correspond to regions 7–9 in the TIG representing task T3. The conditions and events associated with the places and transitions of this Petri net are more straightforward than those associated with the places and transitions of the Ada-net given in figure 3. Place Pn models the condition that region n has completed execution. The transitions of the Petri net signify the following events:

t1:	Task T1 starts rendezvous with T2.E
t2:	Task T3 starts rendezvous with T2.E
t3:	Task T1 ends rendezvous with T2.E
t4:	Task T3 ends rendezvous with T2.E

Since the accept statements in this example have no accept bodies we can construct an even more compact Petri net representation from a reduced TIG representation. The resulting reduced Petri net is drawn in figure 14.

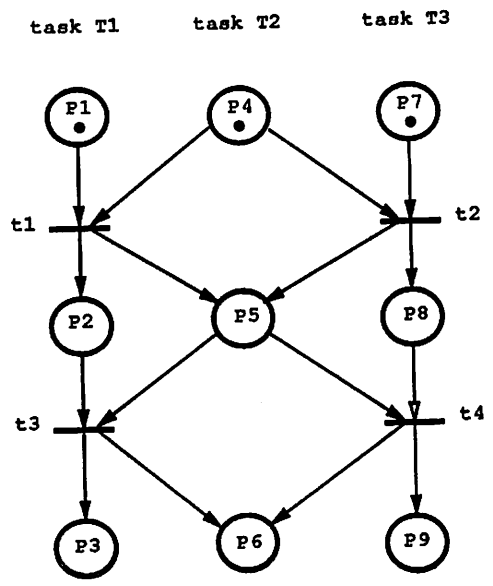

Figure 13: TIG-based Petri net representation for tasks in figure 2

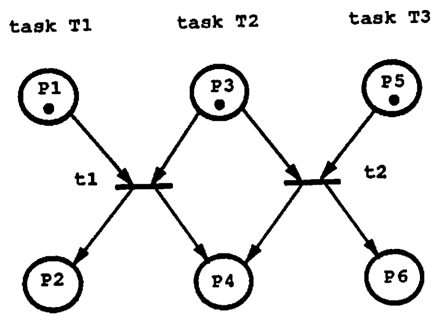

Figure 14: Reduced TIG-based Petri net representation for tasks in figure 2

Dead transitions

A inherent problem with any Petri net model for tasking synchronization is that a there exists a certain degree of added complexity due to the nature of static analysis. Sometimes synchronizations are explicitly represented that will never execute at runtime. For example, when building Ada-nets, a rendezvous is constructed for any call-accept pair that matches syntactically. It is possible however that such a pair may never interact at runtime. In such a case, there will be a number of non-fireable (or dead) transitions in the representation.

For TIG-based Petri nets, this problem is exacerbated by the fact that in order to preserve tasking regions, a single call or accept may be represented on multiple TIG edges. This means that multiple transitions may be constructed to model the same tasking interaction. This in itself is not problematic because the semantics of a TIG-based transition firing is not only that a tasking interaction has occurred, but that the two tasks involved have just completed identifiable regions and will continue executing in their next respective regions. So for a single tasking interaction, we construct a transition for every pair of *regions* that syntactically communicate at that interaction.

However, duplicated edge labels can still present a problem. For Ada-nets, dead transitions only exist if a task interaction is not reachable or if there is deadlock in the system. But for TIG-based Petri nets, dead transitions are also formed when a pair of *regions* can never communicate at an interaction. Take, for example, the set of TIGs found in figure 10. Although the edges from region 2 to itself and from region 4 to region 5 can syntactically interact, clearly this is not a potential interaction in the system. Using the above algorithm, we would produce a Petri net with the following transitions, where the numbers before the arrow represent the input places of the transition and the numbers after the arrow represent the output places of the transition. Again, place numbers correspond to TIG regions.

t1: $1,4 \Rightarrow 2,5$
t2: $2,4 \Rightarrow 2,5$
t3: $3,4 \Rightarrow 2,5$
t4: $1,5 \Rightarrow 3,6$
t5: $2,5 \Rightarrow 3,6$
t6: $3,5 \Rightarrow 3,6$
t7: $1,6 \Rightarrow 2,5$
t8: $2,6 \Rightarrow 2,5$
t9: $3,6 \Rightarrow 2,5$

As noted above, transition t2 does not represent a valid potential tasking interaction and

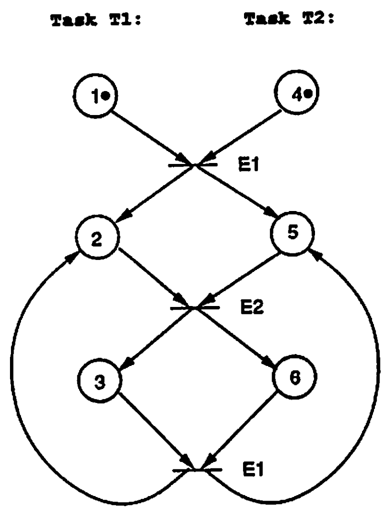

Figure 15: TIG-based Petri net representation for tasks in figure 7

therefore can never fire. In fact only transitions t1, t5 and t9 are fireable.

Ideally we would like to be able to prune all dead transitions. A perfectly pruned TIG-based petri net for the same set of TIGs is given in figure 15. Note that the behaviors of both Ada-nets and TIG-based Petri nets are not affected by these dead transitions. Their reachability graphs will be identical to that of a perfectly pruned net as long as the initial markings are correct. For Ada-nets, this means that there is a single token in the beginning place of each task and for TIG-based Petri nets, it mean that there is a single token at each place corresponding to a TIG start node.

There are two ways in which we can obtain perfectly pruned TIG-based petri nets. One is to use a translation algorithm that generate pruned nets from the start. Such an algorithm would be equivalent building a TICG. Since each edge of the TICG represents an communication between two regions, we simply build a transition for each unique pair of communicating regions over all TICG edges. Another alternative is to generate a reachability graph from a non-pruned Petri net. Any transitions that are not shown to fire in the reachability graph can be pruned. This solution, of course, applies to any Petri net model. Since both solutions are based on using a reachability graph they are essentially equivalent. Neither of these solutions are satisfactory, however, because both require an exponential amount of work in the worst case.

	seq	cond	seq-loop	cond-loop
seq	$2n$			
cond	$2n$	$2n$		
seq-loop	$2n+1$	$2n+1$	$2n+3$	
cond-loop	n^2+2n	n^2+2n	n^2+3n+1	n^3+2n^2+2n

Table 1: Number of transitions for TIG construct combinations

5 Comparisons

The fundamental difference between the Ada-net model and the TIG-based model is that Ada-nets model concurrent programs by explicitly representing potential tasking interactions, control flow information, and Ada tasking semantics, whereas the TIG-based Petri nets only model the synchronization behaviors of the system. Let's consider the example of figure 7 whose corresponding reduced TIGs were given in figure 10. The Ada-net for this program is given in figure 16 and the pruned TIG-based representation was given in figure 15. The Petri net models information such as: both tasks contain a loop, task T1 contains a select statement, and a rendezvous can occur if the calling task has made an entry call and the accepting task is ready to accept the call. The TIG-based Petri net models the fact that first tasks T1 and T2 execute sequential regions 1 and 4 in parallel then synchronize at entry E1. The system then alternates executing regions 2 and 5 in parallel, synchronizing on entry E2 then executing regions 3 and 6 in parallel, synchronizing on entry E1.

In the remainder of this section we attempt compare Ada-nets and TIG-based Petri nets in terms of the number of places and transitions they produce. We will do this by considering a few somewhat contrived examples. Suppose that we have two tasks: one that contains n accepts and another that contains n entry calls. Assume that there is a one-to-one correspondence between calls and accepts. Now let each task take the form of one of the four basic TIG constructs given in figure 17 where s labels denote rendezvous-start edges and e labels denote rendezvous-end edges. One nice property of TIG-based petri nets is that there is a fixed upper bound on the number of places: the number of TIG regions in a set of TIGs. Each construct shown in figure 17 produces $2n + 1$ nodes. This means that then number of places in a TIG-based Petri net for two such tasks is always $4n + 2$. Table 1 gives the number of transitions created from the various combinations of TIG constructs. Here we see that the number of transitions becomes exponential when at least of the of the tasks has a conditional within a loop. This is because a looping conditional TIG construct with n branches creates n^2 back looping edges. For example, consider a loop that contains select

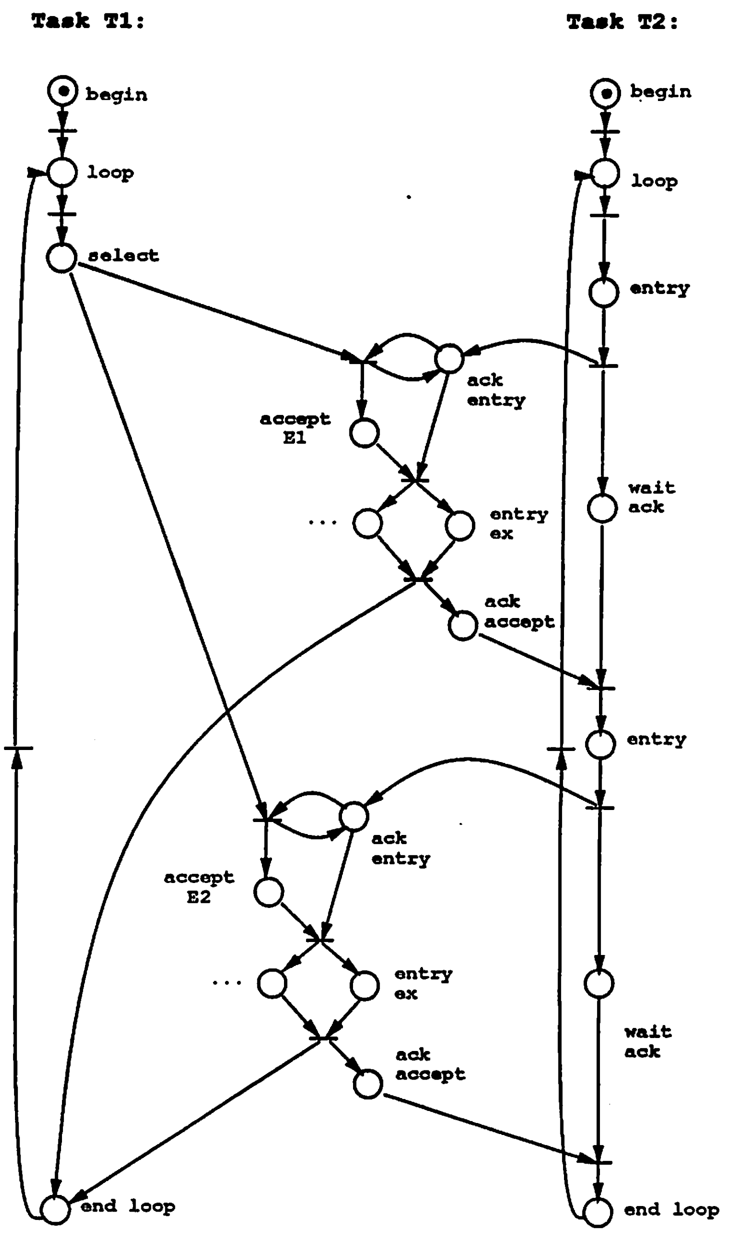

Figure 16: Ada-net representation for tasks in figure 7

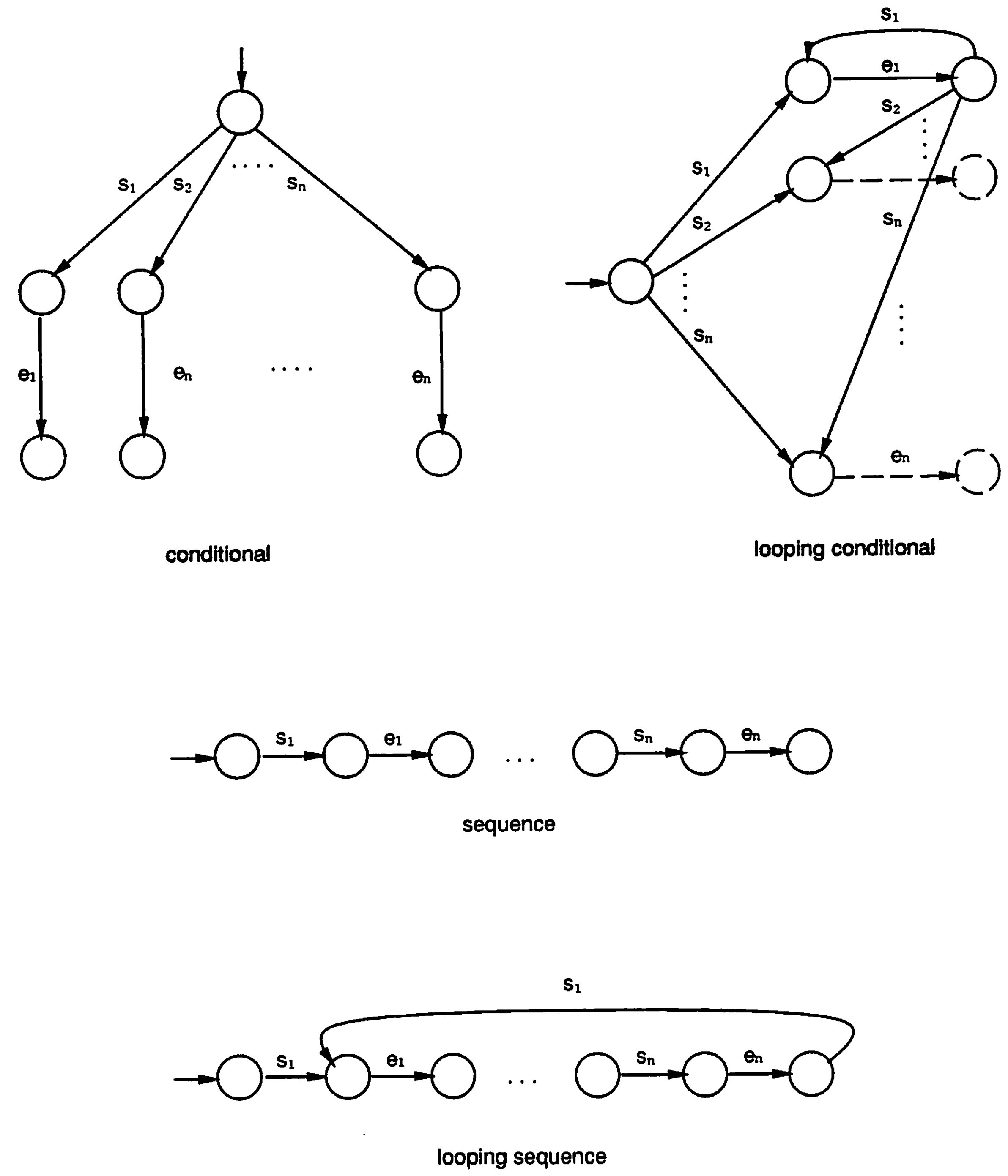

Figure 17: Four basic TIG constructs

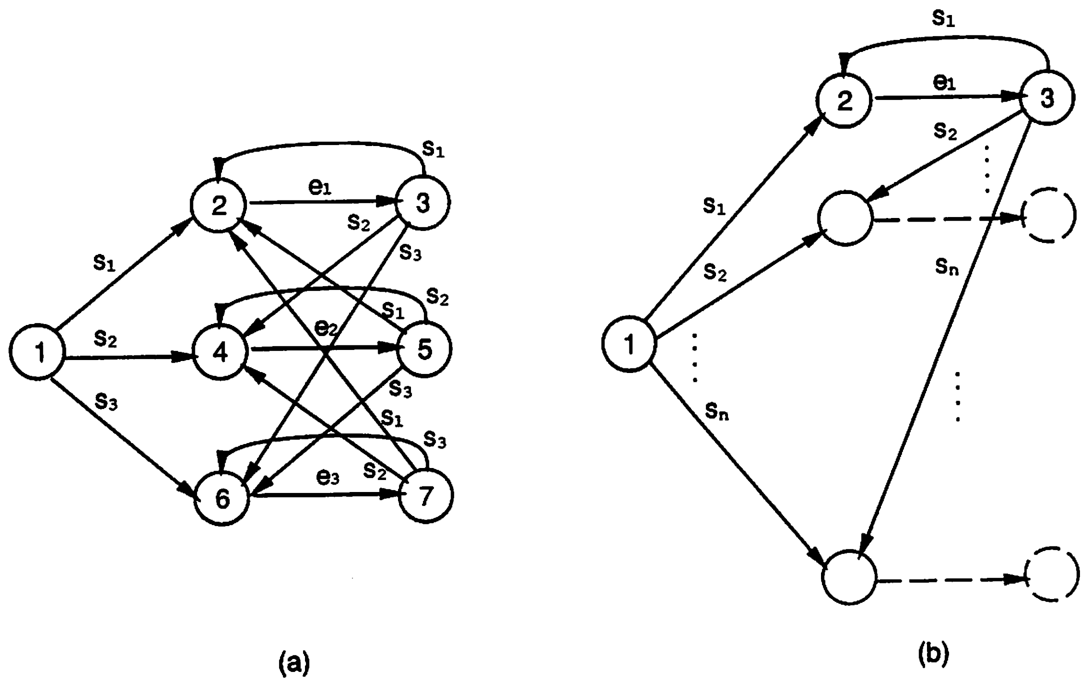

Figure 18: TIG representation for looping conditional with (a) 3 and (b) n branches

statement with 3 branches. A TIG for such a construct is drawn in figure 18(a). Figure 18(b) generalizes over n branches.

Now let's consider the worst case example, where both tasks contain a looping conditional. Pseudocode for such a pair of tasks is given in figure 19. The TIGs for each task will resemble that of figure 18(b). Each rendezvous-end is represented on only one edge of each TIG and therefore has a single match. A rendezvous-start, however, is represented on $n + 1$ edges of each TIG and therefore allows $(n + 1)^2$ matches. This means that for n branches we get a total of $n[(n + 1)^2 + 1]$ transitions in the resulting TIG-based petri net.

The Ada-net for two tasks with looping conditionals on n branches is given in figure 20. We can see that the Ada-net produces only $7n + 8$ places and $6n + 6$ transitions. In fact, for all combinations Ada-nets will produce on the order of $7n$ places and $6n$ transitions. In all combinations, TIG-based petri nets produce fewer places. However, whenever a conditional loop is involved, TIG-based Petri nets produce far more transitions.

Now we will consider the number of transitions for all of the combinations when the Petri nets are perfectly pruned. The number of remaining transitions is given in table 2. Although

```
begin
    loop
        select
            accept < $entry$ >$_1$
        or
            accept < $entry$ >$_2$
                .
                .
                .
        or
            accept < $entry$ >$_n$
        end select
    end loop
end
```

```
begin
    loop
        if (cond)
            call < $entry$ >$_1$
        elsif (cond)
            call < $entry$ >$_2$
                .
                .
                .
        elsif (cond)
            call < $entry$ >$_n$
        end if
    end loop
end
```

Figure 19: Example for worst case

	seq	cond	seq-loop	cond-loop
seq	$2n$			
cond	2	$2n$		
seq-loop	$2n$	2	$2n+1$	
cond-loop	2	$2n$	2	n^2+2n

Table 2: Number of transitions for TIG construct combinations after pruning

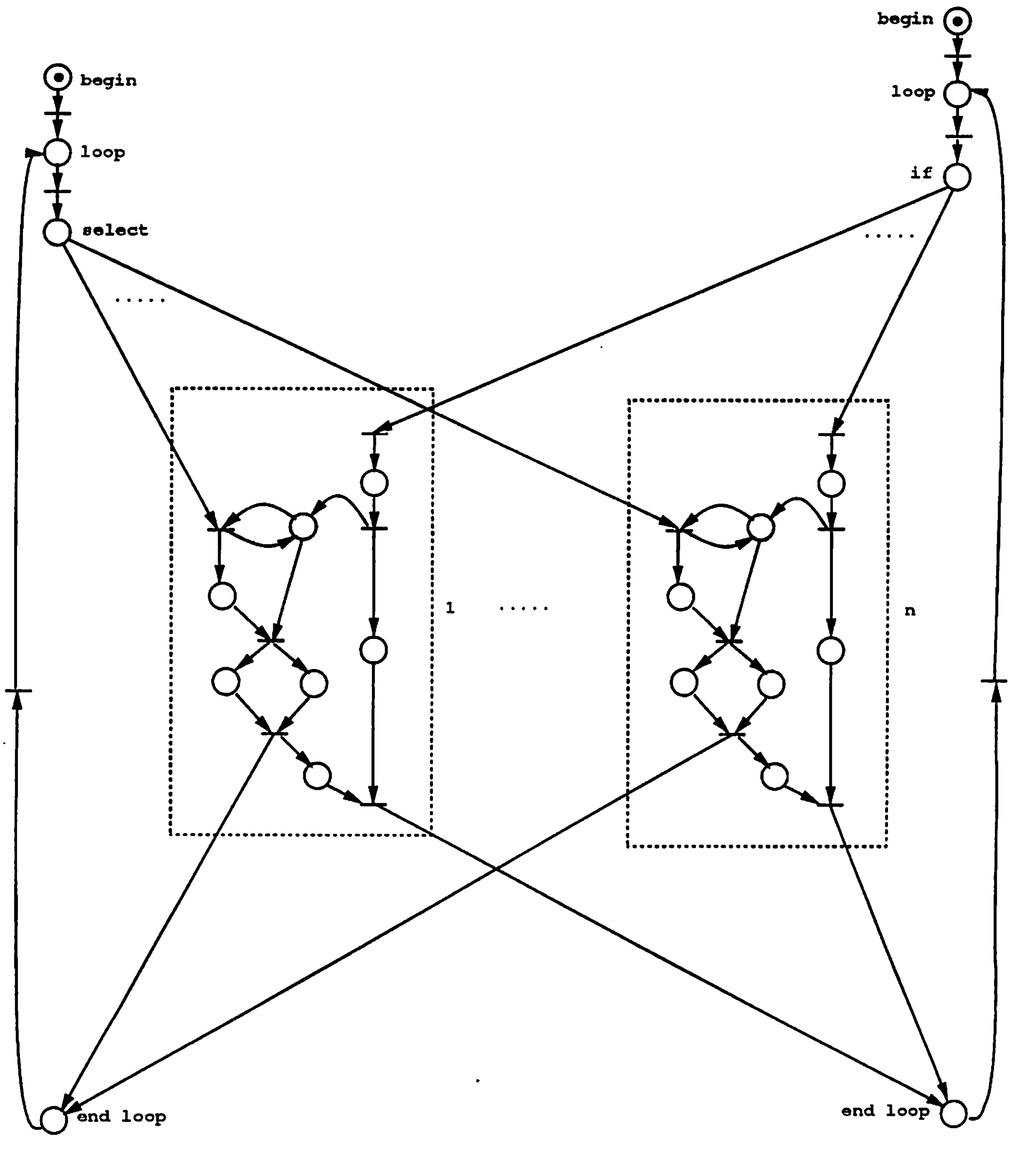

Figure 20: Ada-net representation looping conditional with n branches

pruning significantly reduces the number of transitions for several of the combinations, the case involving two tasks with looping conditionals still produces order n^2 transitions. So even if we could do perfect pruning, looping conditionals will still present a problem. We believe that this uncovers a flaw in the TIG model and propose that the TIG representation for looping conditionals should be reconsidered.

6 Reachability analysis

The Ada-net model represents control flow as explicit places in the net. Some examples of these control flow places are *begin*, *loop*, *if*, and *select*. Because TIG-based petri nets are built from TIGs which abstract away explicit control flow from the representation, they have fewer places than their Ada-net counterparts. In fact, as we saw in section 5, even in the worst case, TIG-based petri nets produce fewer places. Fewer places in a Petri net generally means fewer possible markings and thus smaller reachability graphs. For example, recall the simple tasking example of figure 2. The Ada-net for this example was given in figure 3. The reachability graph for the Ada-net, taken from [SC88], is drawn in figure 21. The non-reduced TIG-based Petri net for this example was given in figure 13 and the reachability graph for that Petri net is drawn in figure 22.

The reachability graph of a TIG-based Petri net is always isomorphic to the corresponding TICG derived from the same set of TIGs. To demonstrate this, we will show that it is trivial to translate between TIG-based reachability graphs and TICGs. One property of TIG-based Petri nets is that for any marking of the net, there is always exactly one token per task that marks the current state of that task. Each node of the reachability graph represents a marking of the net. Given a TIG-based Petri net derived from a set of n TIGs, let t_i be the token that represents the current state of the i^{th} task. Each node of the net's reachability graph can be translated directly to its corresponding TICG node by creating an n-tuple where the i^{th} element of the tuple is the TIG region corresponding to the place where token t_i resides.

7 Conclusion

In this paper, we have presented a new Petri net model for tasking programs based on *task interaction graphs*. We would like to evaluate this model in terms of the three classifications of Petri net analysis: reachability, reduction and matrix-equation approaches. In section 6

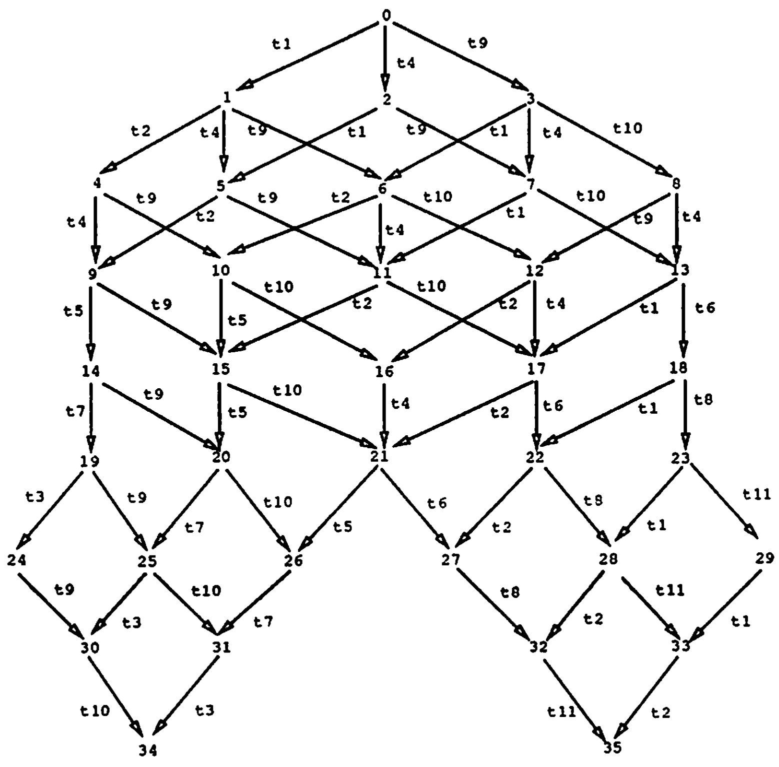

Figure 21: Reachability graph generated from Ada-net of figure 3

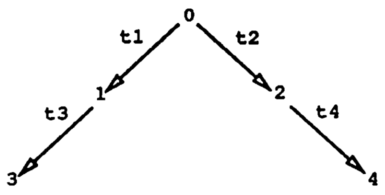

Figure 22: Reachability graph generated from TIG-based Petri net of figure 13

we showed that one benefit given by TIG-based Petri nets is that they tend to produce more compact reachability graphs than Ada-nets.

In [TST], a number of rules are given for Petri net reduction that can significantly reduce the size of Ada-nets while preserving properties of liveness, pseudo-liveness, and proper termination and thus maintain their suitability for deadlock detection. The example of figure 2 is considered in this paper and after a series of reduction applications the reduced Ada-net is isomorphic to the TIG-based Petri net given in figure 14. Although, in general, TIG-based Petri nets are not as small as reduced Ada-nets, they are not limited to deadlock detection. In addition, the general (non-Ada-net specific) reduction rules are still applicable to TIG-based Petri nets. Further comparisons between TIG-based Petri nets and reduced Ada-nets are needed.

We also plan to apply Murata, Shenker and Shatz's method of using Petri net invariants for deadlock detection [MSS89] to TIG-based Petri nets. We believe that TIG-based Petri nets may by well suited for the circular deadlock detection described in that paper.

We are currently looking into methods for pruning TIG-based Petri nets. This involves performing a dataflow analysis technique on the TIGs to obtain ordering information on TIG edges. We can use this information to rule out edge interactions and thus reduce the number of dead transitions in the Petri net. The dataflow framework that we have been considering is a modification to the technique described in [DS91].

We would also like to explore other types of concurrency analysis that are well suited for TIG-base Petri nets. For example, TIG-based Petri nets are ideal for determining parallel regions. A fireable transition in a TIG-based Petri net indicates two pairs of sequential tasking regions that can execute in parallel.

REFERENCES

[ABC+90] George S Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions of Software Engineering*, December 1990. To appear. Available as Technical Report 90-116, Department of Computer and Information Science, University of Massachusetts.

[ACDW90] George S. Avrunin, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated constrained expression analysis of real-time software. COINS Technical Report 90-117, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, December 1990.

[ADWR86] George S. Avrunin, Laura K. Dillon, Jack C. Wileden, and William E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions of Software Engineering*, SE-12(2):278–292, February 1986.

[ALR83] American National Standards Institute. *Military Standard Ada Programming Language (ANSI/MIL-STD-1815A-1983)*, January 1983.

[Dil90] Laura K. Dillon. Verifying general safety properties of ada tasking programs. *IEEE Transactions of Software Engineering*, 16(1):51–63, January 1990.

[DS91] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the 4th Workshop on Software Testing, Analysis, and Verification.* ACM Sigsoft, 1991. To appear.

[FGM89] A. Fuggetta, C. Ghezzi, and D. Mandrioli. Some consideration on real-time behavior of concurrent programs. *IEEE Transactions of Software Engineering*, 15(3):356–359, March 1989.

[HL85] David P. Helmbold and David C. Luckham. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering*, pages 44–52, Pittsburgh, May 1989.

[LC91] Douglas Long and Lori A. Clarke. Data flow analysis and the rendezvous model of concurrency. In *Proceedings of the 4th Workshop on Software Testing, Analysis, and Verification*. ACM Sigsoft, 1991. To appear.

[McD89] C. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, 6(3):515–536, 1989.

[MR87] E. Timothy Morgan and Rami R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Transactions of Software Engineering*, 13(10):1080–1091, 1987.

[MSS89] T. Murata, B. Shenker, and S.M. Shatz. Detection of ada static deadlocks using petri net invariants. *IEEE Transactions of Software Engineering*, 15(3):314–326, 1989.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(44):541–580, April 1989.

[MZGT85] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the ada task system by petri nets. *Computer Languages*, 10(1):43–61, 1985.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[PTY] Mauro Pezze, Richard N. Taylor, and Michal Young. Reachability analysis of concurrent systems. In preparation.

[SC88] S. M. Shatz and W. K. Cheng. A petri net framework for automated static analysis. *The Journal of Systems and Software*, 8:343–359, 1988.

[SMBT90] Sol M. Shatz, Khanh Mai, Christopher Black, and Sengru Tu. Design and implementation of a petri net based toolkit for ada tasking analysis. *IEEE Transactions on Parallel and Distributed System*, 1(4):424–441, October 1990.

[Tai85] K. C. Tai. Reproducible testing of concurrent Ada programs. In *Proceedings of SoftFair II*, pages 49–56, December 1985.

[Tay83a] Richard N. Taylor. Complexitly of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TK86] Richard N. Taylor and Cheryl D. Kelly. Structural testing of concurrent programs. In *Proceedings of the Workshop on Software Testing*, pages 164–169, Banff, Canada, July 1986. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[TO80] Richard N. Taylor and Leon J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions of Software Engineering*, SE-6(3):265–278, 1980.

[TST] S. Tu, S.M. Shatz, and T.Murata. Theory and application of petri net reduction for ada-tasking deadlock analysis. Technical report, Software Systems Laboratory, Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL.

[YTFB89] Michal Young, Richard N. Taylor, Kari Forester, and Debra Brodbeck. Integrated concurrency analysis in a software development environmnet. In *Proceedings of the 3rd Workshop on Software Testing, Analysis, and Verification*, pages 200–209, Key West, Florida, December 1989. ACM Sigsoft.

[YY90] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. Technical report, Software Engineering Research Center, Department of Computer Sciences, Purdue University, September 1990.